PROCESAMIENTO PARALELO

CÓMPUTO PARALELO EN UNA ARQUITECTURA HETEROGÉNEA

Año 2000

ALUMNOS:
CARLOS OMAR RIVARES
ADRIÁN EDUARDO EVANGELISTA

DIRECTOR:
FERNANDO GUSTAVO TINETTI

A mis viejos y hermano, por el aliento y
ser los pilares de mi vida; y también
a mis amigos/as por ser la familia
que supe elegir.

CARLOS OMAR RIVARES

A toda mi familia: abuelos, padres,
hermanos, hermanas, cuñada, y especialmente
a mis sobrinos Juan Martín y María Victoria,
que me dieron la fuerza necesaria
para que esto sea posible.

ADRIAN EDUARDO EVANGELISTA

Agradecimientos

Nuestro primer agradecimiento es a nuestro director por el apoyo, paciencia y empeño invertido en esta tesis. Al profesor Fernando Romero por habernos aguantado todos los días y todo el día en el laboratorio de paralelo y habernos enseñado a soldar "cablecitos" para el hipercubo. También agradecemos las infinitas veces que Alejandra Pizarro nos posibilitó la llave del laboratorio de paralelo como así también la recepción de la misma. No nos olvidamos tampoco de Andrés Barbieri que nos brindó sus amplios conocimientos en Linux en momentos de total incertidumbre. También por los conocimientos en procesamiento paralelo y especialmente en transputers a Laura De Guisti y Diego Tarrío.

También queremos agradecer a nuestras familias y amistades que nos apoyaron en los buenos y malos momentos y que gracias a ellos fue posible nuestra continuidad.

Por último, y no por eso menos importante, agradecemos al director del laboratorio LIDI, Ingeniero Armando De Giusti, y al laboratorio mismo por la posibilidad de utilizar las instalaciones a nuestro gusto y brindarnos las herramientas y bibliografía para el desarrollo de este trabajo.

Indice

1. INTRODUCCION

La utilización de las computadoras para el cálculo científico y aplicaciones que procesan gran cantidad de datos, demandan un procesamiento con mejor rendimiento, bajo costo y una buena producción. Por estos motivos se está incrementando la aceptación y adopción del cómputo paralelo. Los mayores desarrollos en el procesamiento paralelo tuvieron lugar en los procesadores paralelos (PP) y el cómputo distribuído.

Uno de los procesadores paralelos más utilizados son las multicomputadoras basadas en transputers, que combina un conjunto de procesadores RISC con capacidad de interconexión flexible. En el cómputo distribuído un conjunto de computadoras, no necesariamente homogéneas, que pueden ser conectadas a una red son utilizadas para resolver un problema en forma conjunta. El sistema PVM (Parallel Virtual Machine) por ejemplo utiliza el modelo de pasaje de mensaje para permitir a los programadores realizar cómputo distribuido en una amplia variedad de tipos de computadoras, incluyendo PP.

En el campo del procesamiento paralelo, la multiplicación de matrices (MM) es uno de los benchmarks más conocidos, estudiados y aceptados. Tiene aplicaciones en diversas áreas, como por ejemplo el procesamiento de imágenes y aplicaciones que incluyen la resolución de problemas de álgebra lineal.

Los algoritmos más reconocidos para la multiplicación de matrices en forma paralela requieren cómputo intensivo y gran capacidad de almacenamiento. Una grilla de transputers provee un medio propicio para distribuir la carga de cómputo y obtener un rendimiento razonable. Además, como posee memoria distribuida, puede procesar una carga mayor de datos en forma paralela.

Uno de los beneficios del cómputo distribuido es la utilización de hardware existente para reducir costos. La integración de la red de transputers a una red de estaciones de trabajo permite la participación de éstos en un cálculo intensivo, como lo es la multiplicación de matrices.

La presente tesis tiene dos objetivos:

- Integrar una red de computadoras (estaciones de trabajo) entre las que se encuentra un hipercubo de 32 transputers (T805) conectados en forma de grilla. De esta manera se obtiene una máquina paralela "global".
- Desarrollar una aplicación correspondiente al ámbito científico en la máquina paralela. La aplicación elegida fue la multiplicación de matrices.

A partir del presente trabajo se podrán obtener distintas medidas de performance de la multiplicación de matrices en la máquina paralela y tiempos de comunicación requeridos para la integración de los distintos tipos de computadoras.

1.1. Procesamiento Paralelo

En el amplio espectro de problemas resueltos por computadoras, existen dos en particular: los inherentemente paralelos que necesitan cómputo paralelo y aquellos que requieren procesar grandes cantidades de datos. Entre los inherentemente paralelos podemos encontrar productor y consumidor, lectores y escritores, entre otros. Entre los que requieren el procesamiento masivo de datos podemos encontrar aplicaciones para el manejo de datos enviados por un satélite, verificación y prueba de modelos de aviones y astronomía, entre otros.

El vertiginoso desarrollo de la tecnología describe un avance exponencial en la velocidad de procesamiento en los procesadores. Pero existe un límite: la velocidad de la luz en el vacío. Ninguna computadora por más velocidad de procesamiento que posea no podrá igualar a la velocidad obtenida en una colección de procesadores utilizados en forma conjunta para resolver un problema en particular.

De este modo llegamos a la conclusión de que la única forma de tratar algunos problemas es por medio del procesamiento paralelo. Si varias operaciones pueden ser ejecutadas simultáneamente, el tiempo total de procesamiento se verá reducido, aún cuando cada una de las operaciones no se lleve a cabo más rápidamente [Tin98]. En el procesamiento paralelo, esta simultaneidad se logra ejecutando al mismo tiempo un conjunto de pequeñas tareas que resuelven un problema de gran escala [Gei94].

El éxito del procesamiento paralelo se debe a dos de los mayores desarrollos en esta área: los procesadores paralelos (PP) y el cómputo distribuido.

Los procesadores paralelos pueden combinan cientos de procesadores en un único gabinete y conectados a cientos de gigabytes de memoria. Ofrecen un enorme poder de cómputo y son las computadoras más poderosas del mundo. Todos los procesadores poseen las mismas características. O sea se tiene una colección homogénea de procesadores.

En cómputo distribuido un conjunto de computadoras conectadas por una red son usadas colectivamente para resolver un problema de gran escala. La combinación de varias estaciones de trabajo en una red de alta velocidad puede tener mayor poder de procesamiento que una única supercomputadora.

Empezaremos definiendo la operación multiplicación de matricess y la forma de paralelizarla y para luego pasar al estudio del procesamiento paralelo con un conjunto de definiciones que facilitarán el estudio de los siguientes capítulos. Luego estudiaremos las arquitecturas de procesamiento paralelo para obtener una programación óptima ante una computadora paralela dada.

1.2. Multiplicación de matrices

Se quiere obtener la multiplicación de matrices [Gei95]

$$C = AB$$

Asumimos que cada matriz X tiene las dimensión $m^X \times n^X$, X e $\{A,B,C\}$. Se tienen las restricciones en estas dimensiones para que la multiplicación este bien formada: asumiremos que las dimensiones de C son $m \times n$, mientras que la otra dimensión es k.

Asumimos la siguiente asignación de datos a los nodos: dada una matriz $m^X \times n^X$, X e $\{A,B,C\}$, y una grilla lógica $r \times c$ de nodos, particionaremos de la siguiente manera:

$$X = \left(\begin{array}{c|c|c} X_{00} & \cdots & X_{0(c-1)} \\ \hline \vdots & & \vdots \\ \hline X_{(r-1)0} & \cdots & X_{(r-1)(c-1)} \end{array}\right)$$

asignando X_{ij} al nodo $\mathbf{P}_{ij}$. La submatriz X_{ij} tiene las dimensiones $m_i^X \times n_j^X$ con ? $m_i^X = m$ y ? $n_i^X = n$. Si a_{ij}, b_{ij}, c_{ij} denotan el elemento (i,j) de las matrices, respectivamente, entonces los elementos de C estan dados por la siguiente operación

$$c_{ij} = \; ? \; a_{ik} \, b_{kj}$$

Las filas de C se calculan de las filas de A, y las columnas de C se calculan de las columnas de B. Consideremos que cálculo se requiere para formar C_{ij}:

$$C_{ij} = \overbrace{\left(\begin{array}{c|c|c|c} A_{i0} & A_{i1} & \cdots & A_{i(c-1)} \end{array}\right)}^{A'_i} \left.\begin{array}{c} \left(\begin{array}{c} B_{0j} \\ \hline B_{1j} \\ \hline \vdots \\ \hline B_{(r-1)j} \end{array}\right) \end{array}\right\} B'^{j}$$

Notar que A'_i está enteramente asignado al nodo fila i, mientras que B'^{j} está asignado al nodo columnna j. Teniendo

$$A'_i = \left(\begin{array}{c|c|c|c} a'^{0}_i & a'^{1}_i & \cdots & a'^{(k-1)}_i \end{array}\right) \quad \text{y} \quad \left(\begin{array}{c} b'^{0}_j{}^{T} \\ \hline b'^{1}_j{}^{T} \\ \hline \vdots \\ \hline b'^{(r-1)}_j{}^{T} \end{array}\right)$$

vemos que

$$C_{ij} = \sum_{l=0}^{k-1} a'^{l}_i \, b'^{l}_j{}^{T}$$

La multiplicación matriz-matriz puede ser formulada como una secuencia de una actualización. Cada una de estas actualizaciones pueden ser paralelizadas. El pseudo-código para esta paralelización se muestra en la figura 1.1, ejecutándose simultáneamente en todos los nodos $\mathbf{P}_{ij}$.

```
C_ij = 0
for l = 0, k - 1
        broadcast de a'_i^l en mi fila
        broadcast de b'_j^l en mi columna
        C_ij = C_ij + a'_i^l b'_j^T
endfor
```

figura 1.1: Pseudo-código para $C = AB$.

1.3. Definiciones Generales

A continuación se detallan un conjunto de definiciones generales [TIN98a] que ayudarán al entendimiento del léxico del informe.

Paralelismo: Ejecución simultánea (en el mismo instante de tiempo) sobre diferentes componentes físicos (procesadores). El paralelismo es un concepto asociado con la existencia de múltiples procesadores ejecutando un algoritmo en forma coordinada y cooperante. Al mismo tiempo se requiere que el algoritmo admita una descomposición en múltiples procesos ejecutables en diferentes procesadores (concurrencia).

Objetivos del Procesamiento Paralelo:

- Disminuir los tiempos de ejecución.
- Incrementar la eficiencia.
- Atender fenómenos del mundo real que suceden en paralelo.

Proceso y Procesador: Un proceso es un bloque de programa secuencial, con su propio seguimiento de control. El concepto de proceso es el concepto básico e inicial de la programación concurrente: si en el sistema existen procesos independientes, existe la concurrencia. Cada proceso puede residir en un procesador independiente o dedicado. También se pueden tener múltiples procesos sobre el mismo procesador. Se debe notar que en este último caso se tiene concurrencia pero no paralelismo, o simultaneidad de ejecución.

Interacción, Comunicación y Sincronización de procesos: N procesos que residen en un procesador o en varios procesadores interactúan para ejecutar los aspectos del algoritmo global que requieran cooperación.

La interacción requiere comunicación para el intercambio de datos entre los procesos. La comunicación entre dos procesos puede ser por memoria compartida, a través de un

mensaje explícito entre los procesos, o de un mensaje implícito por medio de un proceso servidor intermedio.

Cuando dos procesos necesitan ajustar el orden de ejecución de sus secuencias de instrucciones al estado de la ejecución del otro, se deben sincronizar.

Speed-Up (o Factor de Speed-Up): La relación entre el mejor tiempo de ejecución de un algoritmo sobre un procesador (T_1) y el tiempo de ejecución sobre una arquitectura paralela con N procesadores (T_N) se denomina factor de Speed-Up (S).

$$S = T_1 / T_N$$

- El óptimo que se puede esperar para S es N.
- Normalmente, resulta prácticamente imposible alcanzar el óptimo.
- Además, parece razonable pensar que más allá de un cierto N, para un dado problema algorítmico PA, las ineficiencias propias del algoritmo harán inútil el agregado de nuevos procesadores, es decir que S tendrá una cota máxima distinta de N.

Eficiencia: La relación entre el Speed-Up alcanzado S y el óptimo teórico Sop se define como eficiencia (E).

$$E = S / Sop$$

O, lo que es igual,

$$E = S / N$$

tal como se la encuentra definida en (Kum94), y donde N representa la cantidad de procesadores. Es claro que

$$E \leq 1$$

La eficiencia depende normalmente del tamaño del problema, tal como sucede con el factor de Speed-Up. En este factor de eficiencia están englobados varios aspectos, como el balance de la carga computacional. Cuando la carga computacional de la aplicación está balanceada, se obtiene un grado de ocupación similar en cada uno de los procesadores que forman parte de la arquitectura paralela. El desbalance de carga tiende a secuencializar la ejecución de la solución algorítmica.

Para lograr un rendimiento cercano al óptimo, o el óptimo, es necesario que confluyan tres aspectos en la utilización de las computadoras paralelas:

1. Capacidad de procesamiento del hardware.
2. Capacidad de programación paralela (software) sobre el hardware.
3. Aplicación paralelizable.

1.4. Arquitecturas de Procesamiento

A continuación se dará una breve descripción de la clasificación de las arquitecturas de procesamiento paralelo. Esta clasificación fue establecida por Flynn en 1972 [Tin98], aunque no cubre todas las diversidades de computadoras modernas, es aún utilizada porque impone un orden y mantenimiento de un nivel de simplicidad necesario para las arquitecturas de hoy en día. La categorización establecida por Flynn está basada en la forma en que la arquitectura administra el flujo de instrucciones que operan sobre los datos. Define cuatro categorías: SISD, MISD, SIMD, MIMD.

En la categoría SISD (**S**ingle **I**nstruction **S**ingle **D**ata) se encuentra la computadora clásica de procesamiento secuencial, definida por Von Neumann. Las instrucciones se ejecutan una después de otra, en serie. Poseen una sola unidad de control, una sola unidad de procesamiento y una única memoria.

En el caso de la categoría MISD (**M**últiple **I**nstruction **S**ingle **D**ata) un mismo dato es procesado por múltiples instrucciones en distintas unidades de procesamiento. Este tipo de arquitecturas se adapta a una clase de problemas y no se considera de propósito general.

En la categoría SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) las computadoras poseen un conjunto de elementos de procesamiento idénticos, todos controlados por una única unidad de control. Cada elemento de procesamiento procesa dato/s distinto/s de los demás.

Las computadoras de la categoría MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata) se consideran intrínsecamente paralelas y es aceptada como de propósito general. Las computadoras que se incluyen dentro de esta categoría constan de n procesadores, en donde cada procesador pertenece a la categoría SISD. Por esta razón, cada uno de los procesadores puede ejecutar su propia secuencia de instrucciones, y cada secuencia de instrucciones actúa con diferentes datos de los demás procesadores. La forma en que se conectan los procesadores a la memoria y también entre sí permite diferenciar al menos dos subclases: *multiprocesadores* y las *multicomputadoras*.

En los multiprocesadores, todos los procesadores comparten el mismo espacio de memoria, por este motivo también se las conoce como computadoras fuertemente acopladas. La comunicación entre los procesadores se realiza por medio de la memoria compartida, requiriendo para esto último mecanismos de sincronización.

En las multicomputadoras cada procesador posee su propia memoria (exclusiva) y se comunica con los demás procesadores por medio de mensajes explícitos a través de una red de interconexión. También reciben el nombre de computadoras débilmente acopladas, multicomputadoras de pasaje de mensajes y también computadores con arquitectura de memoria distribuida. En la figura 1.2 se muestra una arquitectura MIMD con memoria distribuida.

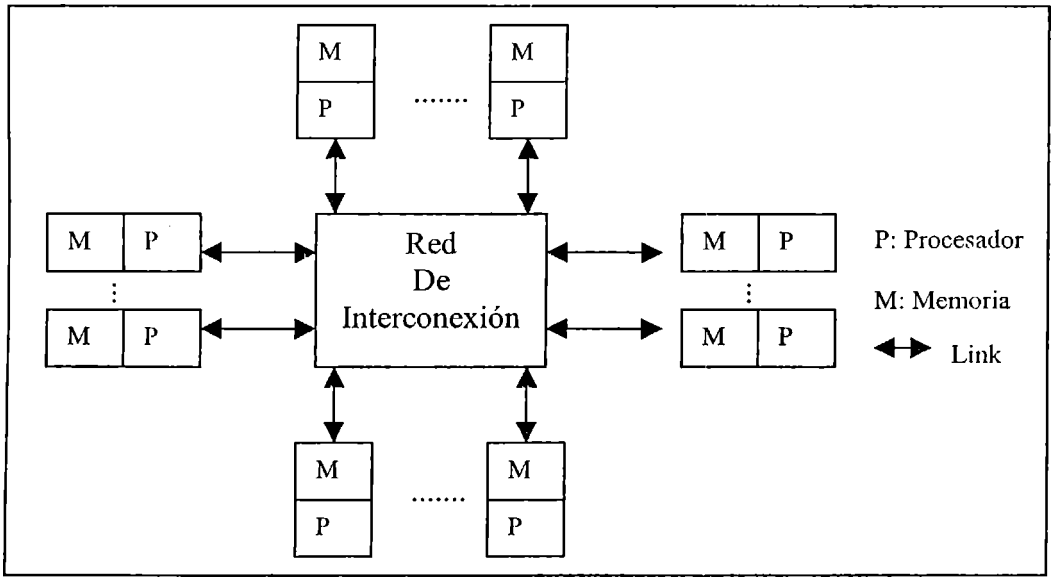


Figura 1.2: Arquitectura MIMD con Memoria Distribuida [Tin98]

1.5. Organización del informe

El informe está organizado de la siguiente manera:

- **Capítulo 1:** Introducción al problema a resolver. Definición del procesamiento paralelo y definiciones asociadas con éste. Breve descripción de las distintas arquitecturas de procesamiento paralelo.

- **Capítulo 2:** Redes de computadoras heterogéneas. Tipos, ventajas y problemas asociados. Sistema PVM.

- **Capítulo 3:** Definición de transputer, su arquitectura y sus características.

- **Capítulo 4:** Algoritmos de multiplicación de matrices. Definición de Single Program Multiple Data (SPMD). Descripción y comparación de los algoritmos SUMMA y DIMMA.

- **Capítulo 5:** Diseño del algoritmo de MM. Diseño del modo SPMD. Descripción de los procesos de ruteo de datos y de procesamiento.

- **Capítulo 6:** Detalles de implementación de los algoritmos de MM, procesos de ruteo de datos y de procesamiento. Deadlock. Buffering. Mensajes. Dimensión de las matrices. Pasaje de parámetros.

- **Capítulo 7:** Integración de la red transputers con PVM. Idea general. Integración de los transputers a la red de computadoras. Pasaje de parámetro en la máquina paralela

"global". Pasaje de parámentros. Entrada y salida del hipercubo a la red. Comunicación con PVM. Algoritmo de MM global.

- **Capítulo 8:** Resultados Obtenidos: tiempos empleados en el algoritmo, speed-up, tiempos de comunicación. Objetivos logrados. Trabajo futuro.

- **Apéndice A:** Lenguaje 'C' paralelo. Rutinas para especificación de concurrencia y semáforos. Manejo de canales físicos y virtuales. Timing y Scheduling.

- **Apéndice B:** Librería de PVM. Rutinas para la administración de procesos y comunicación.

- **Apéndice C:** Configuración del hipercubo en forma de grilla. Placas ***VME-XP***^{TM}. Utilización de los jumpers. Hardwired Link Configuration Array. Configuración de la grilla.

- **Apéndice D:** Instalación del sistema PVM para WIN32 e integración con LINUX.

- **Apéndice E:** Regresión lineal. Parámetros α y β.

2. Redes de computadoras heterogéneas

En la actualidad las empresas o instituciones conectan un conjunto de estaciones de trabajo para compartir recursos, tales como datos o dispositivos caros, aunque no es la única utilidad que se les puede dar. Un conjunto de estaciones de trabajo (*workstations*) interconectadas en una LAN cooperando para resolver un problema tiene un costo más bajo, con respecto al rendimiento que los procesadores paralelos. Esta forma de resolución de problemas de cómputo intensivo se ha denominado cómputo paralelo sobre NOW (**N**etwork **O**f **W**orkstations) [TIN98a].

En un procesador paralelo (PP), cada procesador es exactamente como los demás en capacidad, recursos, software, y velocidad de comunicación. Las computadoras disponibles en una red pueden haber sido hechas por diferentes vendedores o tener diferentes compiladores. Por lo que es necesario tener en cuenta los distintos tipos de heterogeneidad a la hora de programar:

- arquitectura,
- formato de los datos,
- velocidad de procesamiento,
- carga de la máquina, y
- carga de la red.

Cada tipo de arquitectura tiene su propio método óptimo de programación. La máquina paralela virtual puede estar compuesta de computadoras paralelas. Aún cuando la arquitectura sea sólo de workstations seriales, existe el problema de incompatibilidad en los formatos binarios y la necesidad de compilar una tarea paralela en cada máquina diferente.

Los formatos de los datos en computadoras diferentes a menudo son incompatibles, Esta incompatibilidad es un punto importante en el cómputo distribuido porque los datos enviados por una computadora pueden ser ilegibles en la computadora receptora.

En general, los problemas relacionados con el hardware están básicamente resueltos por los protocolos estándares de comunicación como TCP/IP junto con ambientes como PVM e implementaciones de MPI que se encargan de evitar que la heterogeneidad llegue a ser visible desde los programas paralelos.

Aún si el conjunto de computadoras son todas workstations con el mismo formato de datos, se pueden tener diferentes velocidades de procesamiento. La máquina virtual puede estar compuesta de un conjunto idéntico de workstations. Pero la red de computadoras puede tener otros usuarios ejecutando distintos trabajos, por lo que la carga de cada máquina puede variar dramáticamente. Pueden existir estaciones de trabajo que permanecen sin procesamiento útil para ejecutar, lo cual no significa que las computadoras están siempre disponibles, aunque hay períodos en los cuales se puede considerar una disponibilidad completa de las estaciones de trabajo.

Como la carga de la máquina, el tiempo necesario para enviar un mensaje en la red puede variar dependiendo en la carga de la red impuesta por todos los demás usuarios, quienes no utilizan de la misma manera las computadoras de la máquina virtual. Este tiempo de envío es importante cuando una tarea se encuentra detenida esperando un mensaje, y aún más importante cuando el algoritmo depende del tiempo de arribo de los mensajes.

La paralelización de aplicaciones y la caracterización del rendimiento están fuertemente ligadas dado que el beneficio o perjuicio que se obtiene utilizando una máquina paralela se cuantifica en función de los índices de rendimiento que se utilizan. Si los índices de rendimiento son correctos (no ocultan información), el "mejor" programa paralelo dará la justificación para utilizar o no una red de estaciones de trabajo para resolver un problema.

Los índices de rendimiento para carcterizar una NOW están basados en las nociones de factor de *speedup* y MFLOPS (Millones de operaciones de punto flotante por segundo). Esta segunda medida es particularmente útil en el área de los problemas de cómputo intensivo (cálculo numérico).

A pesar de las dificultades causadas por la heterogeneidad, el cómputo distribuido tiene las siguientes caracteristicas:

- Por la utilización de hardware existente, el costo de cómputo puede ser muy bajo.
- El rendimiento puede ser optimizado asignando cada tarea en la arquitectura más apropiada.
- Posibilidad de explotar la naturaleza heterogénea de un cómputo.
- Los recursos de la computadora virtual pueden crecer por etapas y adquirir la última tecnología en cómputo y redes.
- El desarrollo de programas se puede acrecentar usando un ambiente familiar. Los programadores pueden utilizar editores, compiladores, y debuggers que están disponibles en máquinas individuales.
- Las computadoras individuales y workstations son generalmente estables, y se tiene experiencia en un uso confiable.
- La tolerancia a fallas a nivel de usuario y nivel de programa puede ser implementada con poco esfuerzo tanto en la aplicación como en el sistema operativo subyacente.
- El computo distribuido puede facilitar el trabajo colaborativo.

Todos estos factores se trasladan en la reducción del tiempo de desarrollo y corrección, reducción de la disputa por los recursos, costos reducidos, y posiblemente implementaciones más eficientes de una aplicación.

2.1. Introducción a PVM

PVM es un sistema que posibilita desarrollar programas paralelos en forma eficiente y sencilla utilizando un hardware ya existente [Gei94]. Permite que una colección de computadoras heterogéneas sean vistas como una simple máquina virtual

paralela (***Parallel Virtual Machine***). El ruteo de mensajes, la conversión de datos y la administración (scheduling) de tareas es realizada transparentemente por PVM, facilitando el uso de la red conformada por computadoras de diferentes arquitecturas.

El modelo de computación de PVM es simple y general. La interface de programación no es complicada, lo que permite que los programas sean implementados en forma simple e intuitiva. Una aplicación consta de una colección de tareas (*tasks*) que se ejecutarán en la red que conforma la máquina virtual. Los recursos de PVM son accedidos por estas tareas mediante una librería que contiene la interface estándar de las rutinas. Estas rutinas permiten la creación y terminación de las tareas como así también la sincronización entre éstas. Las primitivas de pasaje de mensajes de PVM posibilitan la comunicación de computadoras heterogéneas, usando constructores fuertemente tipados para realizar buffering y la transmisión. Estas primitivas de comunicación incluyen constructores para enviar y recibir datos, así como también primitivas de alto nivel como broadcast, sincronización con barreras, y suma global.

Una tarea, durante su ejecución, puede iniciar o terminar otra tarea o agregar o eliminar computadoras de la máquina virtual. Además, cualquier proceso puede comunicarse o sincronizare con cualquier otro.

2.2. *El Sistema PVM*

PVM es un conjunto integrado de herramientas y librerías que emulan un ambiente de cómputo concurrente en forma heterogénea, flexible y de propósito general sobre computadoras con distintas arquitecturas interconectadas por una red. El objetivo de PVM es permitir que esta colección de computadoras sean usadas cooperativamente para computaciones paralelas o concurrentes. PVM se basa en los siguientes principios:

- "Host pool" configurado por el usuario: Las tareas que forman parte de la aplicación se ejecutan en un conjunto de máquinas determinadas por el usuario para una dada ejecución del programa PVM. Estas máquinas pueden ser multiprocesadores (con memoria compartida o distribuida) o una máquina con una sola CPU. Este "host pool" (colección de computadoras que forman la máquina virtual) puede ser alterado durante la ejecución de una aplicación (una característica importante para la tolerancia a fallas).
- Acceso transparente al hardware: Los programas de aplicación pueden ver el hardware como una colección de elementos de procesamiento virtuales sin atributos particulares (no le interesa los detalles de la arquitectura); o puede aprovechar las características de algunas máquinas, cargando ciertas tareas en la computadora que sea más apropiada para su procesamiento.
- Computación basada en el Proceso: La unidad de paralelismo en PVM es una tarea, un hilo (thread) de control secuencial e independiente que alterna entre comunicación y computación. No necesariamente una sola tarea se puede ejecutar en un procesador, sino que múltiples procesos pueden coexistir en un mismo procesador.
- Modelo de pasaje de mensajes explícito: el conjunto de tareas que forman la aplicación cooperan explícitamente enviando y recibiendo mensajes unas con otras. El tamaño de un mensaje está limitado sólo por la memoria disponible.
- Soporte heterogéneo: PVM soporta heterogeneidad a nivel de máquinas, redes, y aplicaciones. Con respecto al pasaje de mensajes, PVM posibilita que mensajes con

más de un tipo de datos sean intercambiados entre máquinas con distintas representaciones de datos.
- Soporte de Multiprocesadores: PVM utiliza las facilidades de pasajes de mensajes nativas de los multiprocesadores para aprovechar su hardware específico.

El sistema PVM está compuesto por dos partes. La primer parte es un "daemon" llamado *pvmd3* (se abrevia como *pvmd*) que reside en todas las computadoras que conforman la máquina virtual. La segunda parte del sistema es una librería con la interface de las rutinas de PVM. Esta librería contiene un repertorio funcionalmente completo de las primitivas necesarias para la cooperación entre tareas en una aplicación.

El modelo de computación de PVM se basa en la noción de que una aplicación consiste de varias tareas. Cada tarea es responsable de una parte de la aplicación. Una forma de paralelizar una aplicación es según su función; es decir que cada tarea tiene una función distinta, llamado paralelismo funcional. Otro método común de paralelizar una aplicación consiste en que todas las tareas son la misma, pero actúan sobre diferentes datos, llamado paralelismo de datos. Esto también se conoce como el modelo de computación SPMD (Single Program Multiple Data).

PVM actualmente soporta los lenguajes C, C++ y Fortran. Esto se debe a que la mayoría de las aplicaciones que pueden aprovechar las capacidades de PVM están implementadas con C y Fortran, y además se agrega C++ por la tendencia emergente de experimentar con lenguajes y metodologías orientadas a objetos.

Las tareas de PVM son identificadas con un número entero llamado *identificador de tarea (task identifier,* TID). Estos TIDs son suministrados por el pvmd local, y son únicos en la máquina virtual. Los mensajes son enviados y recibidos por medio de estos TIDs, por lo cual PVM provee rutinas para retornar estos valores.

También se pueden generar *grupos de tareas*. Cuando una tarea se une a un grupo se le asigna un número de instancia único en ese grupo. Las funciones de grupo son implementadas por PVM de forma que sean transparentes al usuario. Los grupos pueden superponerse, y las tareas pueden enviar mensajes en forma de broadcast a grupos de los que no son miembros.

El paradigma general para la programación de una aplicación en PVM es el siguiente. El usuario escribe uno o más programas secuenciales en C, C++ o Fotran 77 que contienen llamadas a la librería de PVM. Cada programa corresponde a una tarea que conforma la aplicación. Estos programas son compilados para cada arquitectura en el "host pool", y los archivos ejecutables generados son colocados en un lugar accesible por PVM. Para ejecutar una aplicación, el usuario inicia una tarea (generalmente el "master" o la tarea inicial) manualmente en la máquina del "host pool". Este proceso iniciará otras tareas de PVM, obteniendo una colección de tareas activas que intercambiarán mensajes para resolver el problema.

3. EL TRANSPUTER

Las computadoras basadas en transputers son máquinas MIMD, más precisamente multicomputadoras. Pertenecen a la categoría de las arquitecturas de pasaje de mensajes, en donde los transputers intercambian información unos con otros y no es posible tener acceso remoto a la memoria de otro transputer.

Para poder hacer efectivo el envío de mensajes es necesario poseer una red de interconexión para conectar los transputers. Esta red define la topología de conexión de los transputers. El intercambio de información se realiza por medio de los enlaces de comunicación (links) y en general en las máquinas MIMD son estáticos. Los transputers están conectados directamente y cada transputer posee la misma cantidad de enlaces.

3.1. Características de un transputer

El circuito de un transputer contiene un procesador, una pequeña cantidad de memoria, un coprocesador y cuatro links bidireccionales de alta velocidad [Thi95]. Aunque esta lista no impresiona, el diseño de un transputer provee el ambiente necesario para el procesamiento paralelo.

Procesador Multitasking

El procesador de un transputer soporta multitasking, y fuerza dos niveles de prioridad para tareas concurrentes. Por lo que en una misma aplicación podemos tener procesamiento concurrente y paralelo. El procesamiento concurrente dentro de un mismo transputer y procesamiento paralelo a través de un conjunto de transputers.

Comunicación por medio de links y canales

La comunicación entre transputers es realizada vía links seriales de alta velocidad. El reducido número de alambres de comunicación requeridos y la simplicidad del protocolo hace que la interconexión de los transputers sea simple y en la mayoría de los casos configurable por el usuario. A nivel de software, los links de hardware son definidos como *canales*. Los transputers se comunican por medio de links, las tareas lo hacen por medio de canales. Las primitivas de comunicación son implementadas directamente en microcódigo y el acceso a los links seriales es mapeado a memoria. Por lo que las primitivas para comunicar transputers vecinos son las mismas que para comunicar tareas.

Canales virtuales

Sólo aquellos transputers que están conectados directamente pueden intercambiar mensajes. El lenguaje C paralelo [Log94a][Log94b] posibilita la definición de canales virtuales, que permiten la transferencia de mensajes entre diferentes transputers como si estuvieran conectados por un canal, a pesar de que no estén conectados. Crea un camino virtual que conecta, en forma transparente al usuario, cualquier par de transputers.

3.2. *Arquitectura de un transputer*

El circuito es una computadora (com**puter**) en sí mismo, con un procesador, memoria para almacenar datos y programas, y varios ports para el intercambio o transferencia de información con otros transputers o hacia el exterior. Por el diseño de éstos circuitos que pueden ser conectados de la misma manera como los **trans**itores en una computadora, nació **transputer**. El diagrama del bloque de un transputer se muestra en la figura 3.1.

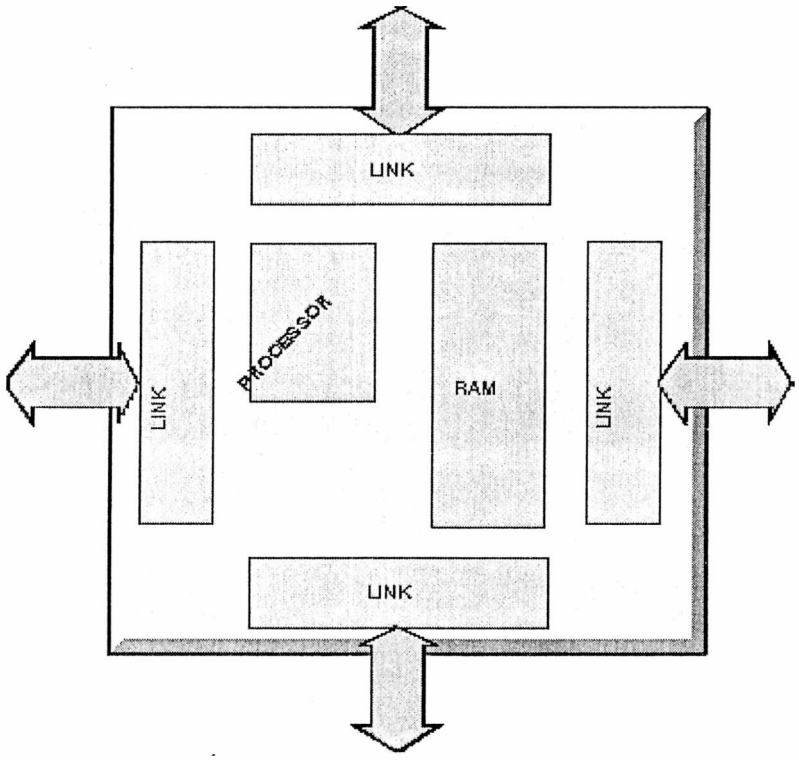


Figura 3.1: Diagrama de bloque del transputer [Thi95]

El término *transputer* realmente abarca una familia de circuitos: algunos con un tamaño de palabra de 16 bits (la serie T2xx), otros con un tamaño de palabra de 32 bits (T805, T800, T425, T414, y el T400). Las series T4xx y T8xx difieren una de otra en varios factores:

- La cantidad de memoria on-chip. El T414 y T400 tienen solo 2Kbytes de memoria interna, mientras que la serie T8xx tiene 4 Kbytes.

- Contienen un procesador de punto flotante. En el T800 y T805 está integrado, mientras que la serie T4xx posee un software que implementa (a una velocidad más lenta) las operaciones de punto flotante.

- El número de ports de E/S disponibles. La serie T4xx tiene dos, mientras que la serie T8xx tiene cuatro, permitiendo redes más complejas.

3.3. *El procesador*

El rendimiento obtenido por los transputers se debe a las siguientes decisiones de diseño:

No tiene registros dedicados a los datos. Posee una pila de registros, la cual permite una selección implícita de los registros. El resultado obtenido es un pequeño formato de instrucción.

Conjunto de instrucción reducido. El transputer adopta la filosofía RISC y dispone de un pequeño conjunto de instrucciones que se ejecutan con unos pocos ciclos de reloj.

Disponibilidad de multitasking en microcódigo. Las acciones necesarias para que el transputer cambie de una tarea a otra son ejecutadas a nivel de hardware, liberando al sistema del programador de esta tarea, y resultando en rápidas operaciones de cambio.

Se estudiarán estas tres características basadas en el Inmos T800. El diagrama de bloque del transputer se muestra en la figura 2.2.

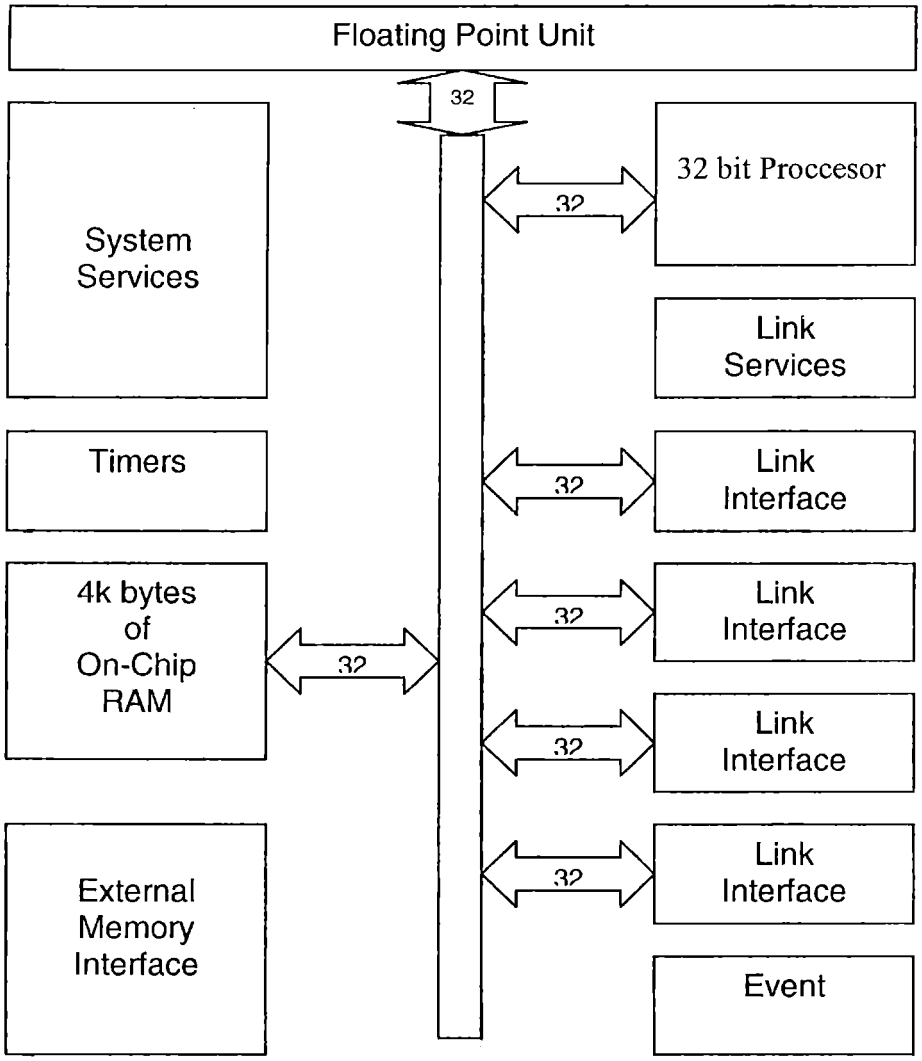


Figura 3.2: Diagrama de bloque de un transputer

Operación basada en pila

El procesador posee seis registros. Tres (los registros A, B y C) son usados como registros de datos e implementan una pila. No es necesario especificar qué registro del procesador recibirá los datos. El procesador toma los datos uno tras otro de la pila y apila el resultado de la operación.

La ventaja de operar con una pila de datos radica en que elimina la necesidad de agregar bits extras en la instrucción para especificar el registro al que debe acceder. Como resultado, las instrucciones pueden ser empaquetadas en palabras más pequeñas, caben mejor en memoria, y se pierde menos tiempo para recuperar las instrucciones de memoria.

Multitasking

El proceso por el cual un procesador divide su tiempo entre varios programas ejecutándose al mismo tiempo se llama *multitasking*. Multitasking es una las formas más importantes de mejorar la performance de un procesador permitiendo comenzar otro programa si uno de los que se está ejecutando actualmente no puede seguir por un tiempo, porque por ejemplo realiza una operación de E/S.

Multitasking es la primera forma de paralelismo disponible por el transputer, y la realiza manteniendo una *lista de tareas* que se deben ser ejecutadas. En cualquier momento una tarea de un transputer puede encontrarse en alguno de estos estados:

Activa
En este estado la tarea se está ejecutando o está en la lista de tareas esperando a ser ejecutada.

Inactiva
La tarea no está en la lista de tareas activas, no puede ejecutarse por una de las siguientes condiciones:

- La tarea está esperando por una entrada en uno de los ports de E/S
- La tarea está esperando por una salida en uno de los ports de E/S
- La tarea pasó a estar inactiva por un periodo de tiempo determinado.

Tareas activas

El transputer mantiene las tareas activas encadenadas en una lista enlazada, y dos de sus registros internos son usados para referenciar el frente y final de la lista. La lista actual es almacenada en memoria, y los registros contienen las direcciones de memoria de las celdas que definen la tarea. Para incrementar la flexibilidad y el poder del ambiente de multitasking, el transputer implementa dos niveles de *prioridad* para las tareas:

Tareas de alta prioridad (nivel 0):

Una vez que obtienen el control del procesador, continúan ejecutándose hasta que terminan, o hasta que necesitan enviar información en un link serial.

Tareas de baja prioridad (nivel 1):

Estas tareas se ejecutan cuando no hay ninguna tarea de alta prioridad activa, y se ejecutan durante un quantum de tiempo, intercambiando en un modelo round robin.

El transputer en realidad necesita mantener dos listas enlazadas, una para las tareas de baja prioridad y otra para las tareas de alta prioridad, y usa un total de cuatro registros para referenciar el frente y final de las listas.

Rápido intercambio de tareas

El intercambio de tareas pertenecientes a la misma lista de prioridad o a diferentes listas de prioridad es manejado directamente por el procesador, y todos los registros que son actualizados son controlados internamente, por medio del microcódigo. Al eliminar esta acción del software perteneciente al kernel da como resultado que esta operación sea extremadamente rápida: menos de 1μs típicamente.

Tareas inactivas

Si una tarea no puede continuar con su ejecución, el procesador la detiene y pasa a estar inactiva, ya sea porque expiró su quantum (evento del timer) o realizó una operación en un link. Al pasar al estado de inactiva, es eliminada de su lista enlazada y puesta en el *workspace*, el cual es un área de memoria.

Timers y tareas inactivas

El T800 posee dos timers de 32 bit. Los timers se encuentran fuera de los procesadores como se muestra en la figura 3.2. Cada timer está asociado con una prioridad. Un timer, disponible para las tareas de alta prioridad, es incrementado cada microsegundo. El otro timer esta asociado con las tareas de baja prioridad y es incrementado cada 64 microsegundos.

3.4. Acceso a memoria

El transputer puede acceder a un espacio lineal de direcciones de 4Gbytes, correspondientes a los 32 bits de los registros de direcciones. De estos 4 Gbytes, 4Kbytes se encuentran dentro del circuito del transputer T805 (2 Kbytes en el T400), y corresponden a la parte baja del espacio de direcciones de la memoria.

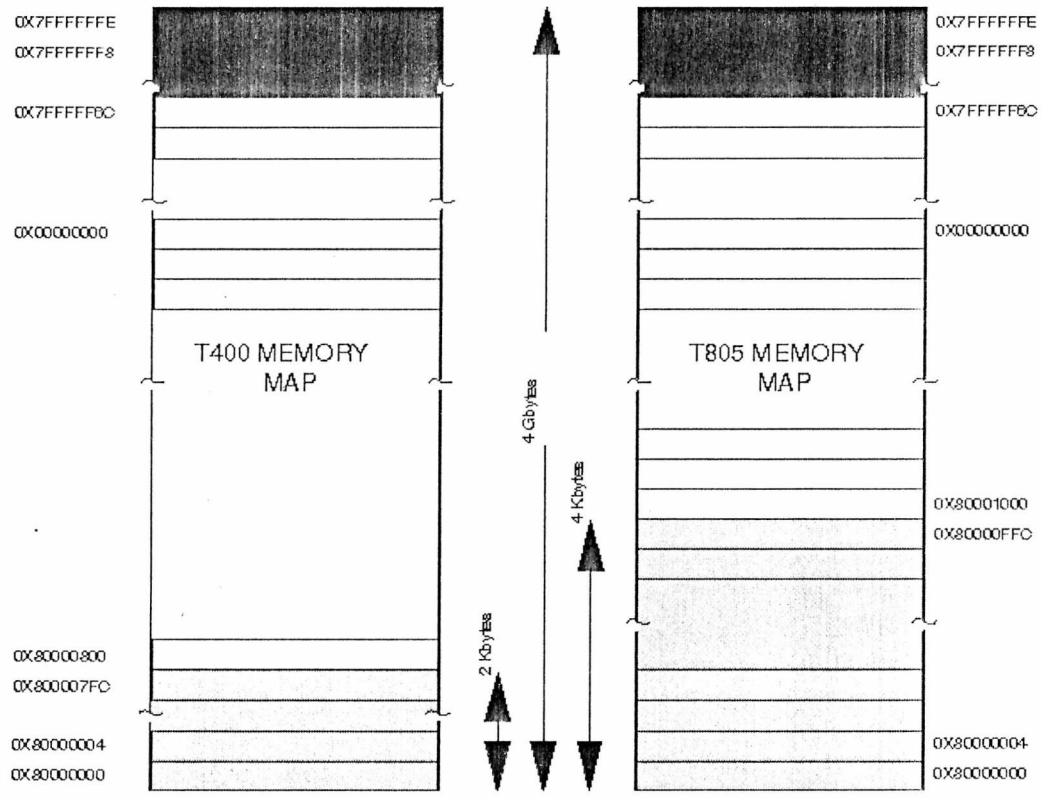


Figura 3.3: Mapas de memoria de los transputers T400 y T800

El Inmos elige el mapeo de forma que las direcciones más bajas de memoria sean negativas, 0x80000000, mientras que la memoria más alta disponible es el entero más grande de 32 bit: 0x7FFFFFF. El rango 0x8000000 hasta 0x80000FFC (0x800007FC para el T400) corresponde a la memoria interna. 0x80001000, la dirección más baja de la memoria externa del T800 se la llama **MemStart** en la terminología de Inmos.

Memoria wrap-around

En la mayoría de los sistemas, no todo el espacio de memoria disponible es utilizado. Por ejemplo, en el CSA Educational Kits, sólo 1 Mbyte de memoria externa es implementado, y sólo los 20 bits más bajos del bus de direcciones son usados. Esto resulta en un "wrap around" del espacio de direcciones de memoria. Como consecuencia el espacio de la pila o heap puede encontrarse fuera de la memoria física, produciendo resultados impredecibles.

En el ejemplo anterior los 12 bits más altos no son usados. Por lo que las direcciones que excedan los 20 bits serán tomadas nuevamente desde la dirección 0 en adelante. O sea, el procesador evita acceder a un espacio de direcciones no existente y para esto toma al espacio de direcciones en forma "circular" .

Rápida memoria interna

Para el transputer, el acceso a su memoria interna estática será aproximadamente 3 veces más rápida que el acceso a su memoria externa. Esta memoria interna deberá ser usada para almacenar los datos y códigos más usados, por ejemplo una pila del usuario.

Otro aspecto de la memoria interna es que afecta directamente el rendimiento del sistema en la tasa transmisión de datos a través de los ports de E/S. La performance mejora si los datos a transmitir se encuentran en memoria interna; pero si no caben en ésta, obviamente la performance se degradará.

3.5. Los ports seriales de E/S

El transputer dispone de cuatro links seriales bidireccionales (dos en el caso del T400). Se utilizará el término link para referirnos a una conexión física entre dos transputers, y el término canal para describir una conexión de software entre dos procesos. La transferencia de datos en un link serial es *sincronizada* y *unbuffered*.

- **Sincronización** se refiere al hecho que si el proceso $P1$ ejecutándose en el transputer $T1$ necesita intercambiar (mandar o recibir) información con el proceso $P2$, entonces debe esperar hasta que $P2$ esté listo para participar en el intercambio. El proceso se dice que es sincronizado, el emisor no puede enviar sus datos hasta que el receptor no esté listo para aceptarlos, y viceversa.

- **Comunicación unbuffered** significa que ningún almacenamiento temporario es necesario para almacenar los mensajes entrantes o salientes. La transferencia se realiza directamente entre la memoria del emisor y la memoria del receptor.

Durante la comunicación, los procesos que inician la transferencia son **bloqueados**. Cada proceso es puesto al final de la lista de tareas inactivas. Debido a que el procesador y los links operan independientemente, el procesador es libre de ejecutar otro proceso cuando uno es bloqueado por comunicación. Si ocho de tales procesos requieren transferir en cada dirección de los cuatro links seriales, todos los links pueden ser activados simultáneamente, obteniendo un "throughput" equivalente al de ocho veces del "throughput" de un link. Como resultado los canales no requieren colas o buffers para los mensajes.

Ports transparentes

Uno de los aspectos arquitecturales de más relevancia de los ports de E/S es que son *mapeados a memoria*. Esto significa que programar un port y pasarle la dirección y longitud del mensaje es realizado escribiendo estos números a posiciones de memoria que son mapeadas a los registros del port.

Como un resultado, las instrucciones que son usadas para programar los ports de E/S son todas instrucciones de lectura y escritura en memoria. Esto hace que estas instrucciones que acceden al link son las que se utilizan las direcciones que mapean a los registros del link. Si dos tareas que se están ejecutando en el mismo transputer,

ambas usando las mismas direcciones, pero una iniciando una salida y la otra una entrada, ambas pueden realizar la comunicación.

Distinguiremos entre *canales de hard* (comunicación entre procesos remotos) versus *canales de soft* (comunicación entre procesos locales al mismo transputer). Sólo el microcódigo sabrá la diferencia entre un canal de soft y un canal de hard. Las instrucciones del lenguaje Assembler no realizan la distinción, y por esta razón, tampoco el código de alto nivel.

Esto significa que un programa paralelo que posee varios procesos concurrentes puede ser adaptado con unos pocos cambios para que se ejecute en un sistema multitransputer.

3.6. Aspectos técnicos del CSA Educational Kit

Se describirá la versión IBM Personal Computer (PC) del CSA Transputer Educational Kit, que es el software de desarrollo utilizado para implementar la aplicación.

Procesador adjunto

El sistema transputer opera como un *procesador adjunto* para la PC. Como tal, no reemplaza el procesador 80x86 en la PC, trabaja en conjunción con él. La PC se llama la *máquina host*, o simplemente *host*. Ningún software ejecutándose en la PC es afectado por la presencia de la red de transputers. En la figura 3.4 se muestra la arquitectura general de una PC con una red de transputers adjunta.

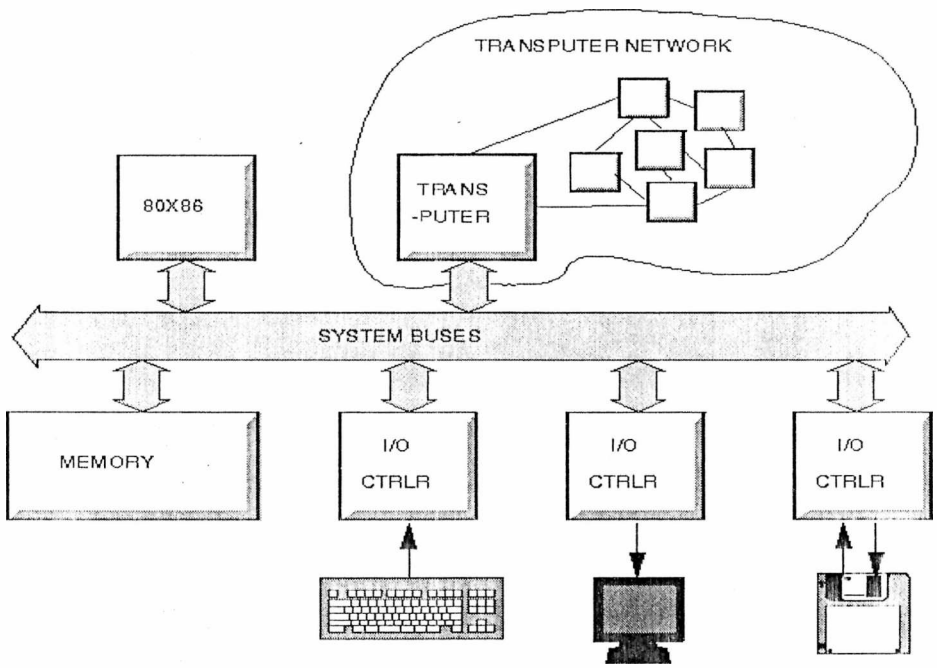


Figura 3.4: El transputer como un procesador adjunto

Uno de los transputers en la red es considerado un "conductor". En la mayoría de las veces se encuentra dentro de la PC, en uno de los slots ISA de E/S de 8 bit, por lo que puede tener acceso a los buses del sistema. Allí, se comporta como un despachador (dispacher), transferiendo información desde la PC hacia la red de transputers (programas y datos de entrada), o desde la red de transputers hacia el host (datos resultantes). A este transputer especial se lo llama transputer *PC-link*, o transputer *root*. El transputer root dedica uno de sus links para la comunicación con la PC.

En algunos sistemas, el transputer root es localizado dentro de la máquina paralela, y la PC es equipada con una plaqueta de interface la cual implementa los ports de E/S como se muestra en la figura 3.5. Con este simple concepto de interface, la red de transputers puede ser conectada a varias PC's, o a una misma PC por medio de varias tarjetas de interface. Tales configuraciones, son para aplicaciones específicas como multiusuarios.

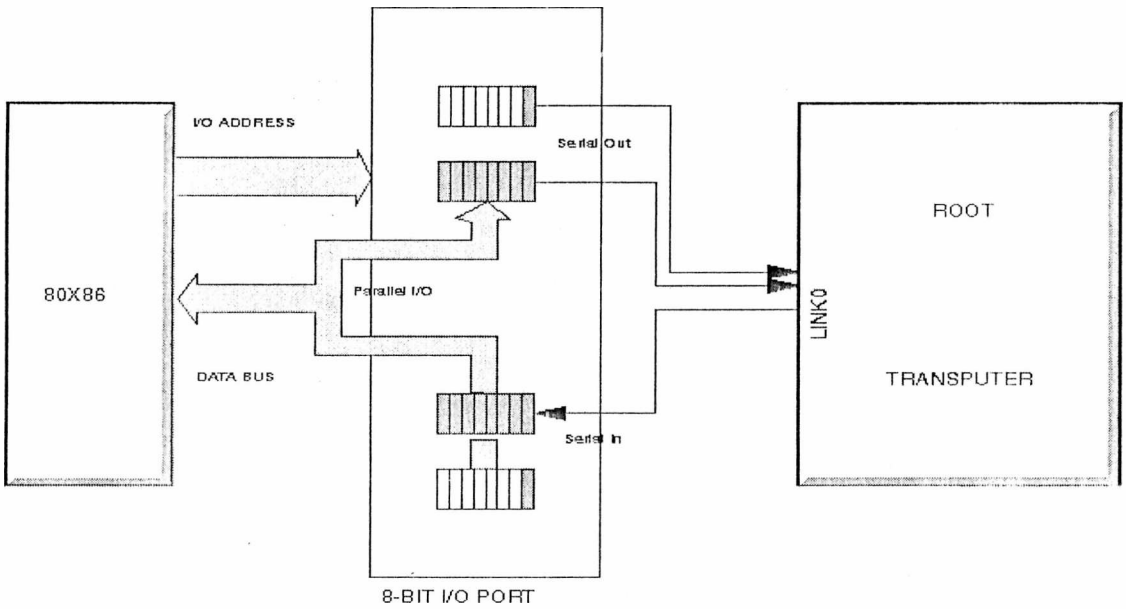


Figura 3.5: Interface de un port de 8 bits entre el host y el transputer root.

Comportamiento de un port serial

La transferencia de información entre el host y el transputer root es administrada por el host como una operación de E/S estándar de 8 bit, muy similar a la comunicación con un port paralelo o RS232.

Cada acceso a los recursos de la PC debe ser a través de port de E/S de 8 bit, el cual es mantenido, por medio de software, por el 80x86. A veces el host será un simple subordinado del transputer root, atendiendo sus requerimientos y poniendo todos sus recursos a disposición del transputer. Otras veces, será más avanzado dividiendo el cómputo entre la red de transputers y el 80x86, y dejando las operaciones de E/S al último.

4. ALGORITMOS DE MULTIPLICACION DE MATRICES

Dadas dos matrices **A** y **B** se desea obtener la matriz producto **C=AxB**. Esta multiplicación se implementó utilizando el algoritmo DIMMA (Distribution-Independent Matrix Multiplication Algorithm) propuesto por Jaeyoung Choi [Cho].

Los algoritmos clásicos de multiplicación de matrices [Kum94] se basan en una grilla de PxP procesadores, donde los bloques de las matrices son mapeados directamente a los procesadores. A diferencia de estos algoritmos, DIMMA y SUMMA (Scalable Universal Matrix Multiplication Algorithm) utilizan una distribución de bloques cíclica (ver más adelante) sobre una grilla de PxQ procesadores, con P y Q arbitrarios.

Primero se describirá la distribución de bloques cíclica y el modelo de cálculo. Luego se detallarán los algoritmos de SUMMA y DIMMA y una comparación de ambos.

4.1. Block Cyclic Data Distribution

La forma en la cual los datos son distribuidos en los procesadores en una computadora concurrente tiene un gran impacto en la rendimiento y escalabilidad de un algoritmo concurrente, ya que influye en las características de comunicación y en el balanceo de carga. La distribución de bloques cíclica (*block cyclic distribution* o también conocida como *block scattered*) [Cho92] [Cho94][Cho] provee una forma simple y de propósito general de distribuir una matriz particionada en bloques sobre una computadora concurrente con memoria distribuida.

Dada una malla de PxQ procesadores, una matriz es particionada en bloques de tamaño *rxs*; y los bloques separados por P filas de distancia, como así también a Q columnas de distancia son asignados al mismo procesador. Así, al procesador en la posición (p, q) ($0 \le p < P$, $0 \le q < Q$) de la malla se le asignan los bloques cuyo índice es:

$$(p + iP , q + jQ)$$

donde $i = 0, \ldots, (M_{bk}-p-1)/P$ y $j = 0, \ldots, (N_{bk}-q-1)/Q$, donde $M_{bk}xN_{bk}$ es el tamaño de la matriz en bloques.

En la figura 4.1 se muestra un ejemplo de la distribución de bloques cíclica, donde una matriz de 7x7 bloques es distribuida sobre una grilla de 2x3 procesadores. Cada cuadrado dentro de la matriz (figura 4.1-a) representa un bloque de elementos, y el valor dentro de este bloque indica su posición en la grilla de procesadores. Todos los bloques etiquetados con el mismo número son asignados al mismo procesador. Los números a la izquierda y arriba de la matriz representan los índices de las filas y columnas de bloques, respectivamente.

	0	1	2	3	4	5	6
0	0	1	2	0	1	2	0
1	3	4	5	3	4	5	3
2	0	1	2	0	1	2	0
3	3	4	5	3	4	5	3
4	0	1	2	0	1	2	0
5	3	4	5	3	4	5	3
6	0	1	2	0	1	2	0

a) distribución del punto de vista de la matriz

	0	3	6	1	4	2	5
0							
2		P_0		P_1		P_2	
3							
6							
1							
3		P_3		P_4		P_5	
5							

b) distribución del punto de vista de los procesadores

Figura 4.1: distribución de bloques cíclicas de una matriz de 7x7 bloques en una grilla de 3x4 procesdores

Como puede observarse en la figura 4.1(b), en este ejemplo no todos los procesadores reciben la misma cantidad de bloques. Esto se debe a que la cantidad de filas de bloques de la matriz no es múltiplo de la cantidad de filas de procesadores en la grilla y además la cantidad de columnas de bloques de la matriz no es múltiplo de la cantidad de columnas de procesadores en la grilla. La cantidad de bloques que reciben los procesadores difieren a lo sumo en una fila y/o una columna, pero todos los procesadores en una misma fila de la grilla tendrán la misma cantidad de filas de bloques, y los procesadores en una misma columna de la grilla tendrán la misma cantidad de columnas de bloques. Esto explica como la distribución de bloques cíclica balancea la carga de los procesadores. Este balanceo es a nivel de bloque, por lo cual si el tamaño de bloque es muy grande este balanceo no será tan real. Por ejemplo si los bloques fueran de tamaño 100 entonces las matrices de P_0 serán de 100 filas más que las matrices de los procesadores que se encuentren en otra fila de procesadores, como sucede con P_5; lo mismo ocurre con las columnas. Con una distribución directa de bloques de la matriz a los procesadores, los procesadores de la última fila y última columna de la grilla tendrán porciones de la matriz mucho más pequeñas (especialmente el procesador de la última fila y última columna, como por ejemplo el procesador P_5 de la figura 4.1 que tendrá sólo el bloque de la fila 6 y columna 6 de la matriz).

Esta forma de distribución también contribuye a la escalabilidad de los algoritmos que la utilizan. No impone ninguna restricción en cuanto a la cantidad procesadores en la grilla, el tamaño de las matrices y de los bloques dentro de cada matriz.

4.2. Modelo de cálculo

Para obtener la matriz producto **C**=**A***__B__, se asuma que **A**, **B** y **C** son de tamaño MxK, KxN, y MxN respectivamente. A su vez, el tamaño de los bloques de **A** es m_bxk_b, y los bloques de **B** y **C** son de k_bxn_b y m_bxn_b elementos, respectivamente. Por lo cual el número de bloques de las matrices **A**, **B**, y **C** son M_{bk}xK_{bk}, K_{bk}xN_{bk}, respectivamente, donde $M_{bk}=M/m_b$, $N_{bk}=N/n_b$, y $K_{bk}=K/k_b$.

4.3. SUMMA

En SUMMA [Gei95], las matrices **A** y **B** son divididas en bloques de tamaño K_b, o sea que los bloques tienen igual cantidad de filas y columnas. Los procesadores multiplican la primera columna de bloques de **A** con la primera fila de bloques de **B**. Luego multiplican la próxima columna de bloques de **A** con la próxima fila de bloques de **B**, y así sucesivamente se repite el proceso para todas las columnas de **A** y filas de **B**, obteniendo así la matriz resultado **C**.

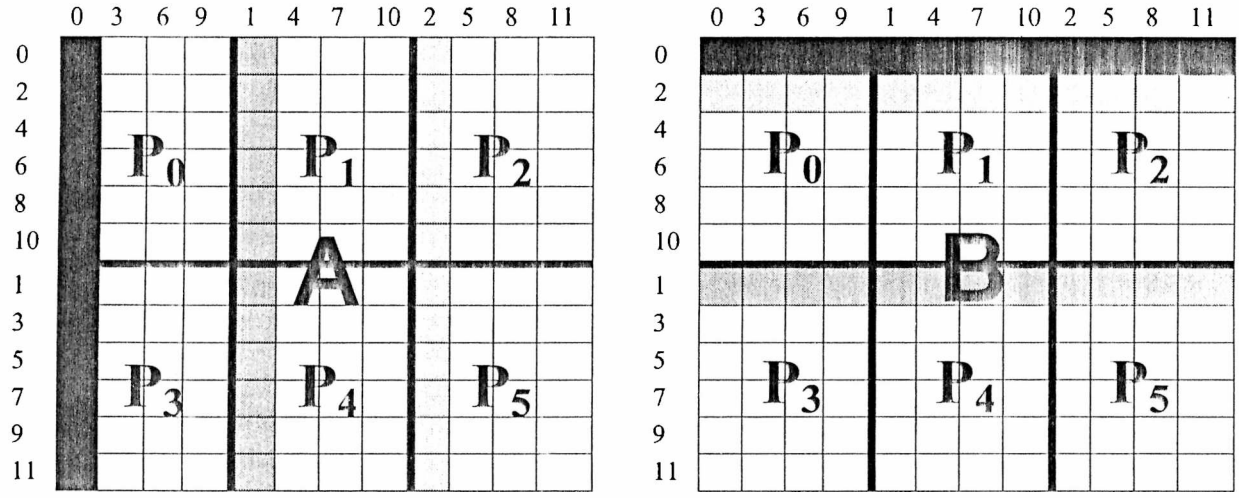


Figura 4.2: Algoritmo SUMMA. Los bloques más obscuros son enviados primero y los más claros después.

En la figura 4.2 se muestra un ejemplo de SUMMA, con una grilla de 2x3 procesadores, y la matrices **A**, **B** y **C** poseen 12x12 bloques. En el primer paso, la primera columna de procesadores, P_0 y P_3, comienzan realizando un broadcast de la primera columna de bloques de **A** hacia cada fila de procesadores. Al mismo tiempo la primera fila de procesadores, P_0, P_1, y P_2, envían un broadcast con la primera fila de bloques de **B** hacia cada columna de procesadores. Luego cada procesador realiza la multiplicación local obteniendo un resultado parcial de C correspondiente al primer paso. En el próximo paso, la segunda columna de procesadores, P_1 y P_4, envían un broadcast de la segunda columna de **A** hacia cada fila de procesadores, y la segunda fila de procesadores, P_3, P_4, y P_5, realiza un broadcast de la segunda fila de **B** hacia cada columna de procesadores. Este proceso continúa hasta la última columna de bloques de **A** y la última fila de bloques de **B**. En la tabla 2.1 se muestran cada uno de los pasos de SUMMA, detallándose para cada paso, qué envía y qué recibe cada proceso para realizar luego su cálculo local.

Paso 0			**Paso 1**		
Proceso	Envía	Destino	Proceso	Envía	Destino
P_0	col 0 de **A**	P_1, P_2	P_1	col 1 de **A**	P_2, P_0
P_3	col 0 de **A**	P_4, P_5	P_4	col 1 de **A**	P_5, P_3
P_0	fil 0 de **B**	P_3	P_3	fil 1 de **B**	P_0
P_1	fil 0 de **B**	P_4	P_4	fil 1 de **B**	P_1
P_2	fil 0 de **B**	P_5	P_5	fil 1 de **B**	P_2
Paso 2			**Paso 3**		
Proceso	Envía	Destino	Proceso	Envía	Destino
P_2	col 2 de **A**	P_0, P_1	P_0	col 3 de **A**	P_1, P_2
P_5	col 2 de **A**	P_3, P_4	P_3	col 3 de **A**	P_4, P_5
P_0	fil 2 de **B**	P_3	P_3	fil 3 de **B**	P_0
P_1	fil 2 de **B**	P_4	P_4	fil 3 de **B**	P_1
P_2	fil 2 de **B**	P_5	P_5	fil 3 de **B**	P_2
Paso 4			**Paso 5**		
Proceso	Envía	Destino	Proceso	Envía	Destino
P_1	col 4 de **A**	P_2, P_0	P_2	col 5 de **A**	P_0, P_1
P_4	col 4 de **A**	P_5, P_3	P_5	col 5 de **A**	P_3, P_4
P_0	fil 4 de **B**	P_3	P_3	fil 5 de **B**	P_0
P_1	fil 4 de **B**	P_4	P_4	fil 5 de **B**	P_1
P_2	fil 4 de **B**	P_5	P_5	fil 5 de **B**	P_2
Paso 6			**Paso 7**		
Proceso	Envía	Destino	Proceso	Envía	Destino
P_0	col 6 de **A**	P_1, P_2	P_1	col 7 de **A**	P_2, P_0
P_3	col 6 de **A**	P_4, P_5	P_4	col 7 de **A**	P_5, P_3
P_0	fil 6 de **B**	P_3	P_3	fil 7 de **B**	P_0
P_1	fil 6 de **B**	P_4	P_4	fil 7 de **B**	P_1
P_2	fil 6 de **B**	P_5	P_5	fil 7 de **B**	P_2
Paso 8			**Paso 9**		
Proceso	Envía	Destino	Proceso	Envía	Destino
P_2	col 8 de **A**	P_0, P_1	P_0	col 9 de **A**	P_1, P_2
P_5	col 8 de **A**	P_3, P_4	P_3	col 9 de **A**	P_4, P_5
P_0	fil 8 de **B**	P_3	P_3	fil 9 de **B**	P_0
P_1	fil 8 de **B**	P_4	P_4	fil 9 de **B**	P_1
P_2	fil 8 de **B**	P_5	P_5	fil 9 de **B**	P_2
Paso 10			**Paso 11**		
Proceso	Envía	Destino	Proceso	Envía	Destino
P_1	col 10 de **A**	P_2, P_0	P_2	col 11 de **A**	P_0, P_1
P_4	col 10 de **A**	P_5, P_3	P_5	col 11 de **A**	P_3, P_4
P_0	fil 10 de **B**	P_3	P_3	fil 11 de **B**	P_0
P_1	fil 10 de **B**	P_4	P_4	fil 11 de **B**	P_1
P_2	fil 10 de **B**	P_5	P_5	fil 11 de **B**	P_2

Tabla 2.1: Pasos en la ejecución del algoritmo SUMMA.

4.3.1. Esquema de comunicación de SUMMA

Este algoritmo [Gei95] explota el esquema de comunicación *pipelined*, en donde el broadcast es implementado pasando una columna (o fila) de bloques por todo el anillo lógico que forma la fila (o columna) de procesadores. Por ejemplo, cuando P_0 realiza un broadcast de la primera columna de **A** hacia toda la fila de procesadores, en realidad esta columna es enviada a P_1, el cual luego la reenviará a P_2 en el próximo paso. En la figura 4.3 se muestra este esquema de comunicación, que posibilita cómputo y comunicación en forma simultánea y eficiente en toda la máquina paralela.

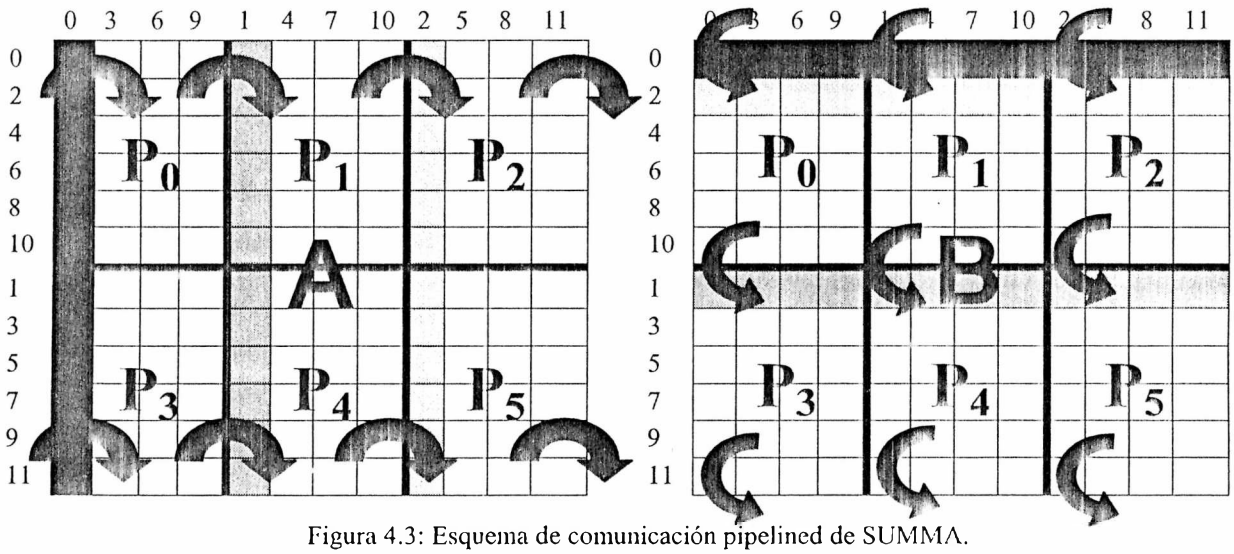
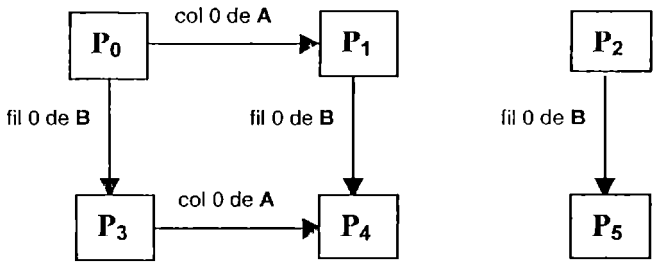


Figura 4.3: Esquema de comunicación pipelined de SUMMA.

4.3.2. Tiempo de espera innecesario de SUMMA

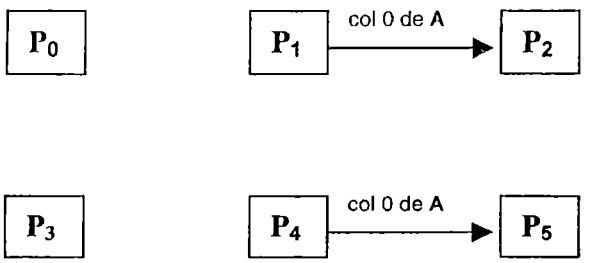
En SUMMA existe un tiempo de espera innecesario que puede mejorarse. Este tiempo de espera se puede observar en el ejemplo anterior: luego de que P_0 multiplica y envía la columna 0 de **A** y la fila 0 de **B**, debe esperar por la columna 1 de **A** (que la tiene P_1) y la fila 1 de **B** que la tiene P_3. Cuando P_1 realiza un broadcast de la columna 1 de **A**, por el esquema de comunicación pipelined, se la envía a P_2, el cual luego se la envía a P_0. Se puede observar que P_0 tuvo que esperar dos (en realidad Q-1) pasos. Más adelante, cuando P_1 tenga que enviar su próxima columna (4), P_0 (que ya envió la columna 3) tendrá que volver a esperar otros dos pasos, y así para todas sus columnas. En la figura 4.4 se detallan los primeros 13 pasos de la ejecución de SUMMA para el ejemplo anterior. Las flechas entre procesadores indican el envío de datos, y para cada paso se puede observar el estado de cada procesador, si esta calculando o esperando. Si se sigue cuidadosamente esta figura se puede notar que cada procesador, luego de enviar sus datos, debe esperar al menos dos pasos para poder recibir la siguiente columna de **A** y fila de **B** para realizar la multiplicación.

Paso 1



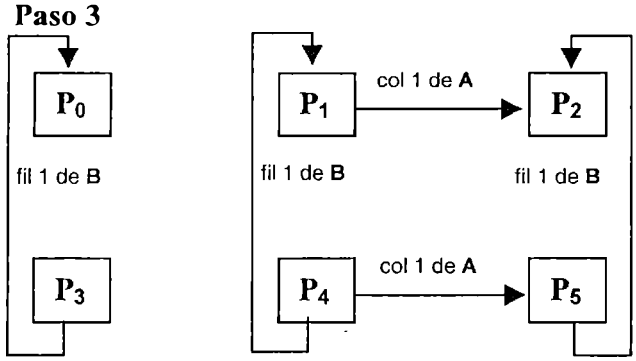
P_0 multiplica col 0 de **A** con fil 0 de **B**
P_1 espera col 0 de **A**
P_2 espera col 0 de **A**
P_3 espera fil 0 de **B**
P_4 espera col 0 de **A** y fila 0 de **B**
P_5 espera col 0 de **A** y fila 0 de **B**

Paso 2



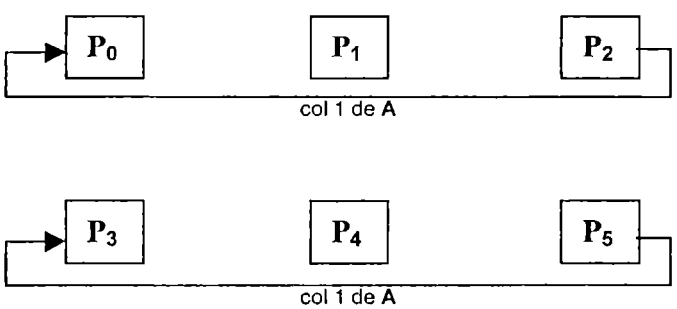
P_0 espera col 1 de **A** y fila 1 de **B**
P_1 multiplica col 0 de **A** con fil 0 de **B**
P_2 espera col 0 de **A**
P_3 multiplica col 0 de **A** con fil 0 de **B**
P_4 multiplica col 0 de **A** con fil 0 de **B**
P_5 espera col 0 de **A**

Paso 3



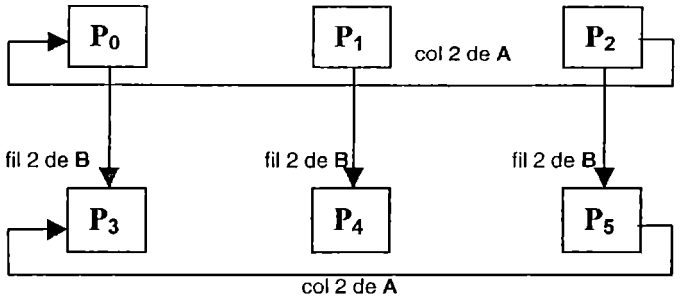
P_0 espera col 1 de **A** y fila 1 de **B**
P_1 espera fil 1 de **B**
P_2 multiplica col 0 de **A** con fil 0 de **B**
P_3 espera col 1 de **A**
P_4 multiplica col 1 de **A** con fil 1 de **B**
P_5 multiplica col 0 de **A** con fil 0 de **B**

Paso 4



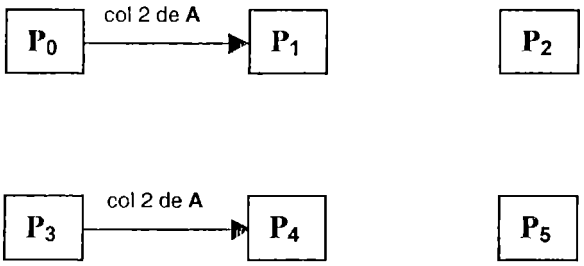
P_0 espera col 1 de **A**
P_1 multiplica col 1 de **A** con fil 1 de **B**
P_2 multiplica col 1 de **A** con fil 1 de **B**
P_3 espera col 1 de **A**
P_4 espera col 2 de **A** y fila 2 de **B**
P_5 multiplica col 1 de **A** con fil 1 de **B**

Paso 5



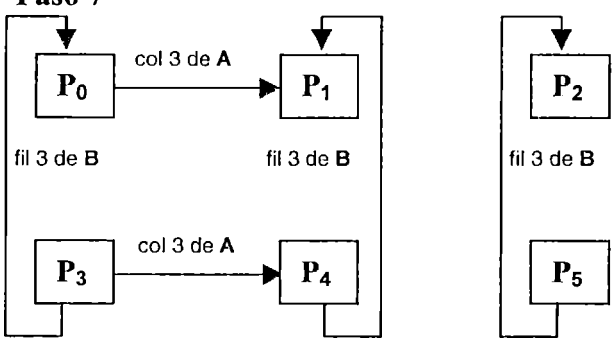
P_0 multiplica col 1 de **A** con fil 1 de **B**
P_1 espera col 2 de **A**
P_2 multiplica col 2 de **A** con fil 2 de **B**
P_3 multiplica col 1 de **A** con fil 1 de **B**
P_4 espera col 2 de **A** y fila 2 de **B**
P_5 espera fila 2 de **B**

Paso 6



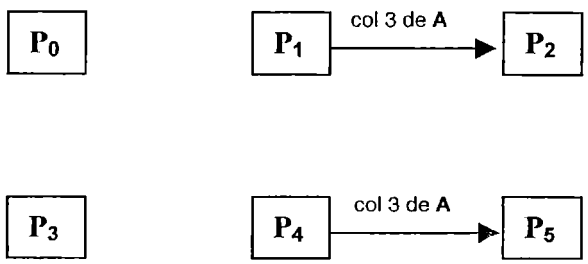
P_0 multiplica col 2 de **A** con fil 2 de **B**
P_1 espera col 2 de **A**
P_2 espera col 3 de **A** y fila 3 de **B**
P_3 multiplica col 2 de **A** con fil 2 de **B**
P_4 espera col 2 de **A**
P_5 multiplica col 2 de **A** con fil 2 de **B**

Paso 7



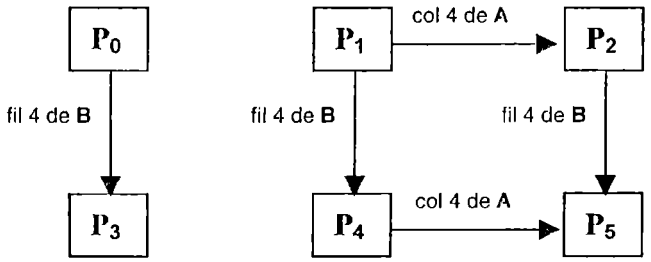
P_0 espera fil 3 de **B**
P_1 multiplica col 2 de **A** con fil 2 de **B**
P_2 espera col 3 de **A** y fila 3 de **B**
P_3 multiplica col 3 de **A** con fil 3 de **B**
P_4 multiplica col 2 de **A** con fil 2 de **B**
P_5 espera col 3 de **A**

Paso 8



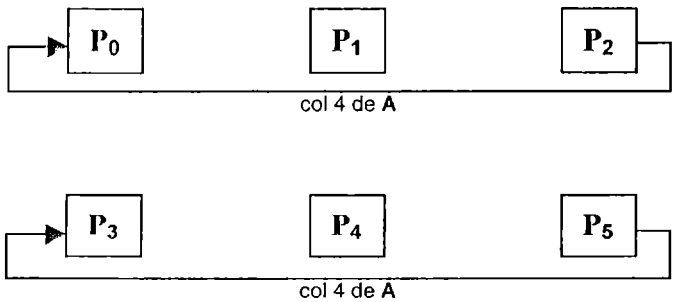
P_0 multiplica col 3 de **A** con fil 3 de **B**
P_1 multiplica col 3 de **A** con fil 3 de **B**
P_2 espera col 3 de **A**
P_3 espera col 4 de **A** y fila 4 de **B**
P_4 multiplica col 3 de **A** con fil 3 de **B**
P_5 espera col 3 de **A**

Paso 9



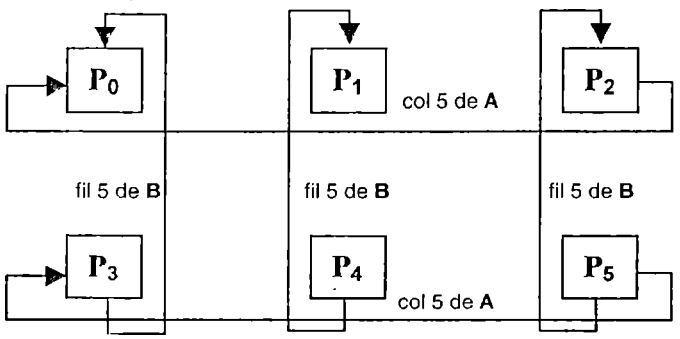
P_0 espera col 4 de **A**
P_1 multiplica col 4 de **A** con fil 4 de **B**
P_2 multiplica col 3 de **A** con fil 3 de **B**
P_3 espera col 4 de **A** y fila 4 de **B**
P_4 espera fila 4 de **B**
P_5 multiplica col 3 de **A** con fil 3 de **B**

Paso 10



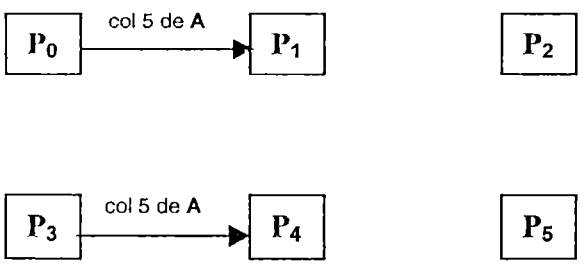
P_0 espera col 4 de **A**
P_1 espera col 5 de **A** y fila 5 de **B**
P_2 multiplica col 4 de **A** con fil 4 de **B**
P_3 espera col 4 de **A**
P_4 multiplica col 4 de **A** con fil 4 de **B**
P_5 multiplica col 4 de **A** con fil 4 de **B**

Paso 11



P_0 multiplica col 4 de **A** con fil 4 de **B**
P_1 espera col 5 de **A** y fila 5 de **B**
P_2 espera fil 5 de **B**
P_3 multiplica col 4 de **A** con fil 4 de **B**
P_4 espera col 5 de **A**
P_5 multiplica col 5 de **A** con fil 5 de **B**

Paso 12



P_0 multiplica col 5 de **A** con fil 5 de **B**
P_1 espera col 5 de **A**
P_2 multiplica col 5 de **A** con fil 5 de **B**
P_3 multiplica col 5 de **A** con fil 5 de **B**
P_4 espera col 5 de **A**
P_5 espera col 6 de **A** con fil 6 de **B**

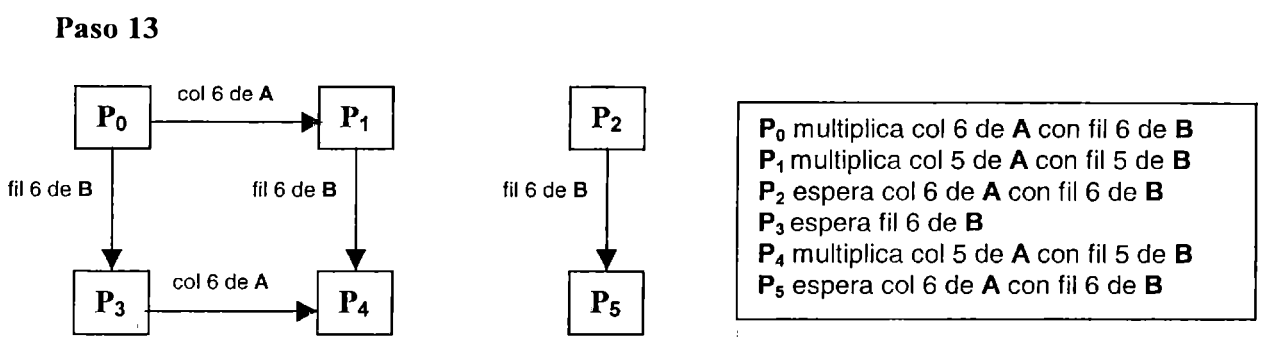


Figura 4.4: Pasos intervientes en la comunicación pipelined de SUMMA.

4.4. DIMMA

DIMMA implementa la multiplicación de matrices en forma rápida y escalable con distribución de datos cíclica sobre computadoras concurrentes con memoria distribuida. Este nuevo algoritmo se basa en el algoritmo SUMMA [Gei95], al que le incorpora dos nuevas ideas. Por un lado utiliza un "esquema de comunicación pipelined modificado", que hace que el algoritmo sea eficiente al superponer comunicación y cómputo en forma efectiva. Además utiliza el concepto del mínimo común múltiplo LCM detallado más adelante.

4.4.1. Esquema de comunicación de DIMMA

DIMMA modifica el esquema de comunicación de SUMMA de la siguiente manera: cada procesador realiza un broadcast de todas las columnas de **A** y filas de **B** que tiene a los demás procesadores antes de que el próximo procesador comience a realizar un broadcast de sus datos.

Con este nuevo esquema, DIMMA se implementa de la siguiente manera. Sea L_m el número de columna de bloques de **A** y fila de bloques de **B** correspondiente al paso actual. En el primer paso, L_m=0, se envía un broadcast y multiplica la columna L_m de **A** y la fila L_m de **B**. En el próximo paso, L_m tendrá valor 6, por lo cual la primera columna de procesadores, P_0 y P_3, realiza un broadcast de la columna número 6 de **A** hacia cada fila de procesadores; y la primera fila de procesadores, P_0, P_1, y P_2, envían la fila 6 de **B** hacia cada columna de procesadores, como se muestra en la figura 4.5. El valor 6 se debe a que es el mínimo común múltiplo (LCM) de P=2 y Q=3, y se utiliza porque los bloques a distancia LCM, en filas y columnas, siempre se encontrarán en el mismo procesador por la forma de distribución cíclica. En cambio, en el algoritmo SUMMA, en el segundo paso, L_m sería igual a 1, por lo que los procesadores P_1 y P_4 enviarían la columna 1 de **A** y los procesadores P_3, P_4, P_5 enviarían la columna de **B**, produciéndose las esperas antedichas.

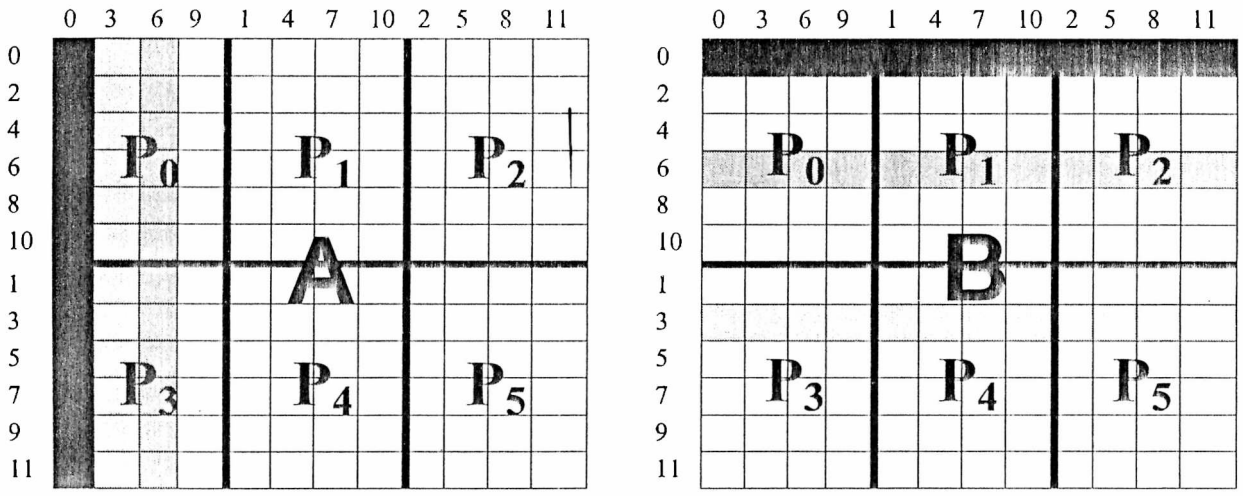
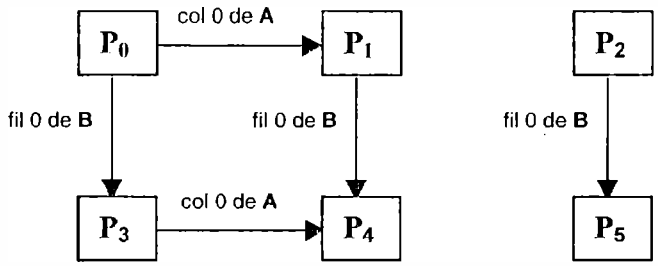


Figura 4.5: Algoritmo DIMMA. Los bloques más obscuros son enviados primero y los más claros después.

En el tercer y cuarto paso, L_m tendrá los valores 3 y 9 respectivamente, por lo cual la primera columna de procesadores, P_0 y P_3, envían hacia todas la filas, de las columna 3 y 9 de **A**, y la segunda fila de procesadores, P_3, P_4, y P_5, envían un broadcast hacia todas la columnas, de las filas 3 y 9 de **B**, respectivamente.

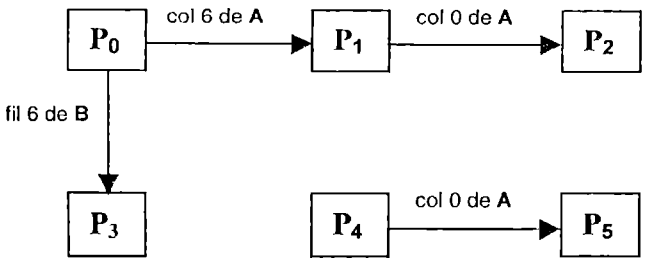
Este nuevo esquema de comunicación introducido por DIMMA elimina el tiempo de espera innecesario que existe en SUMMA. Esto lo realiza al enviar cada procesador todas sus columnas y filas en forma consecutiva, de esta manera el tiempo de espera de SUMMA aparecerá sólo cuando cada procesador termine de enviar todos sus datos y debe esperar por los del siguiente procesador. En el caso de SUMMA este tiempo aparece por cada L_m, ya que éste L_m se recorre en forma consecutiva y los L_m consecutivos pertenecen a distintos procesadores, tanto en columnas y filas de **A** y **B** (ver distribución cíclica). En la Figura 4.6, se muestran los primeros 13 pasos de la comunicación pipelined de DIMMA.

Paso 1



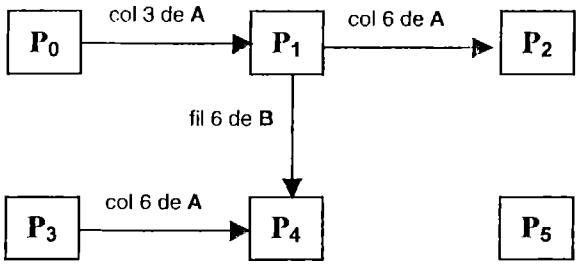
P_0 multiplica col 0 de **A** con fil 0 de **B**
P_1 espera col 0 de **A**
P_2 espera col 0 de **A**
P_3 espera fil 0 de **B**
P_4 espera col 0 de **A** y fila 0 de **B**
P_5 espera col 0 de **A** y fila 0 de **B**

Paso 2



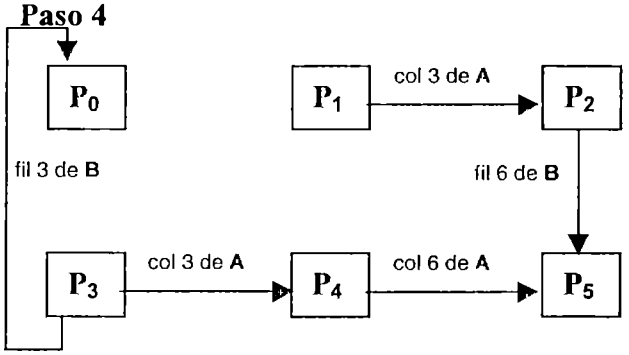
P_0 multiplica col 6 de **A** con fil 6 de **B**
P_1 multiplica col 0 de **A** con fil 0 de **B**
P_2 espera col 0 de **A**
P_3 multiplica col 0 de **A** con fil 0 de **B**
P_4 multiplica col 0 de **A** con fil 0 de **B**
P_5 espera col 0 de **A**

Paso 3



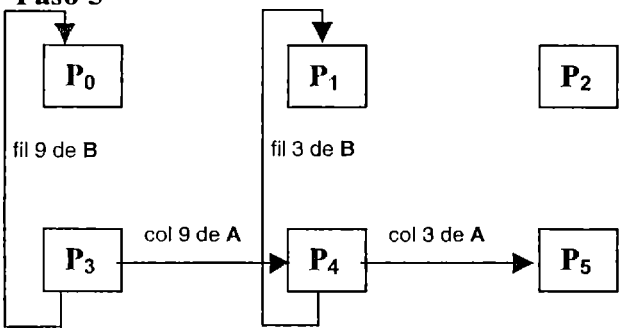
P_0 espera fil 3 de **B**
P_1 multiplica col 6 de **A** con fil 6 de **B**
P_2 multiplica col 0 de **A** con fil 0 de **B**
P_3 multiplica col 6 de **A** con fil 6 de **B**
P_4 espera col 6 de **A** y fil 6 de **B**
P_5 multiplica col 0 de **A** con fil 0 de **B**

Paso 4



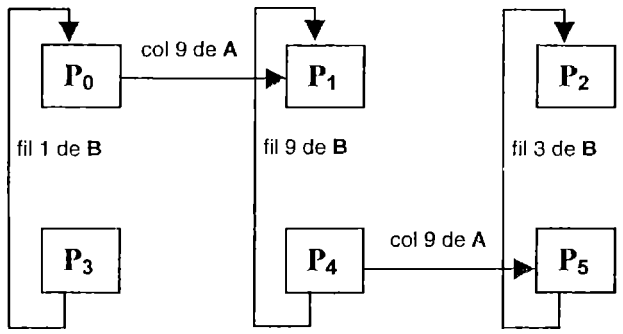
P_0 espera fil 3 de **B**
P_1 espera fil 3 de **B**
P_2 multiplica col 6 de **A** con fil 6 de **B**
P_3 multiplica col 3 de **A** con fil 3 de **B**
P_4 multiplica col 6 de **A** con fil 6 de **B**
P_5 espera col 6 de **A** y fil 6 de **B**

Paso 5



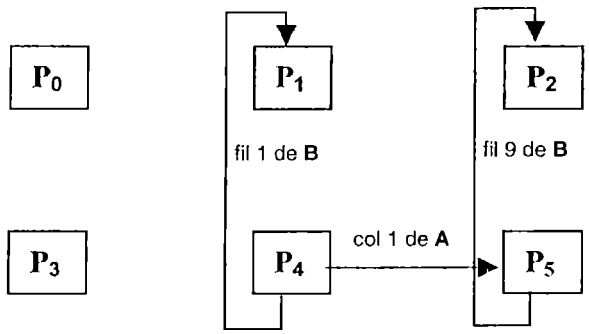
P_0 multiplica col 3 de **A** con fil 3 de **B**
P_1 espera fil 3 de **B**
P_2 espera fil 3 de **B**
P_3 multiplica col 9 de **A** con fil 9 de **B**
P_4 multiplica col 3 de **A** con fil 3 de **B**
P_5 multiplica col 6 de **A** con fil 6 de **B**

Paso 6



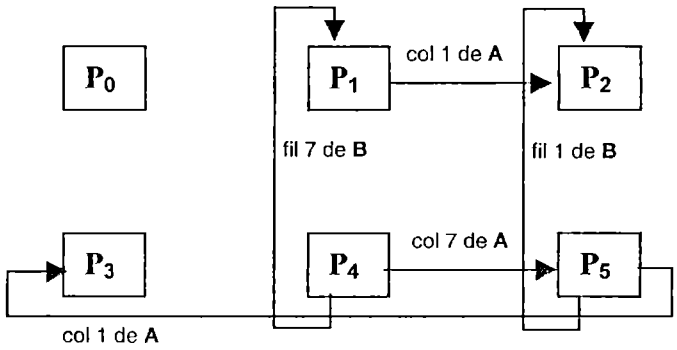
P0 multiplica col 9 de **A** con fil 9 de **B**
P1 multiplica col 3 de **A** con fil 3 de **B**
P2 espera fila 3 de **B**
P3 espera col 1 de **A**
P4 multiplica col 9 de **A** con fil 9 de **B**
P5 multiplica col 3 de **A** con fil 3 de **B**

Paso 7



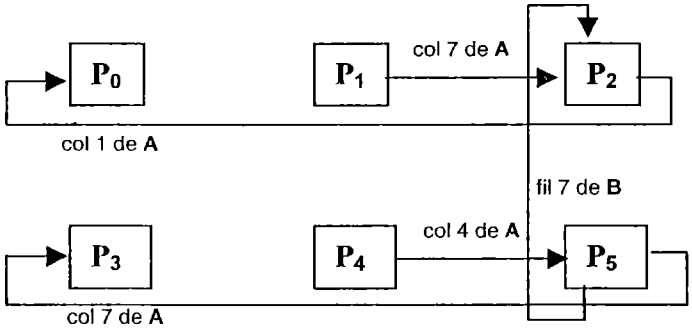
P0 espera col 1 de **A**
P1 multiplica col 9 de **A** con fil 9 de **B**
P2 multiplica col 3 de **A** con fil 3 de **B**
P3 espera col 1 de **A**
P4 multiplica col 1 de **A** con fil 1 de **B**
P5 multiplica col 9 de **A** con fil 9 de **B**

Paso 8

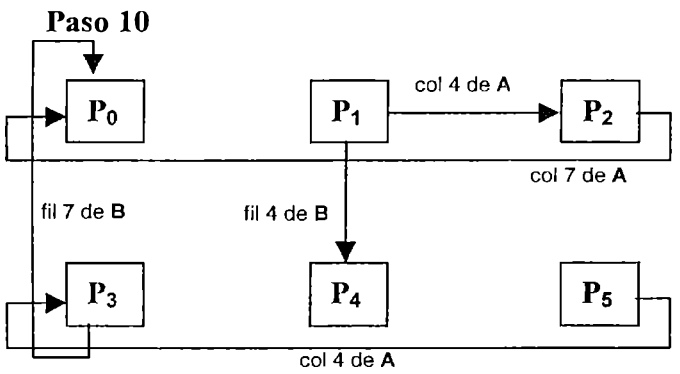


P0 espera col 1 de **A**
P1 multiplica col 1 de **A** con fil 1 de **B**
P2 multiplica col 9 de **A** con fil 9 de **B**
P3 espera col 1 de **A**
P4 multiplica col 7 de **A** con fil 7 de **B**
P5 multiplica col 1 de **A** con fil 1 de **B**

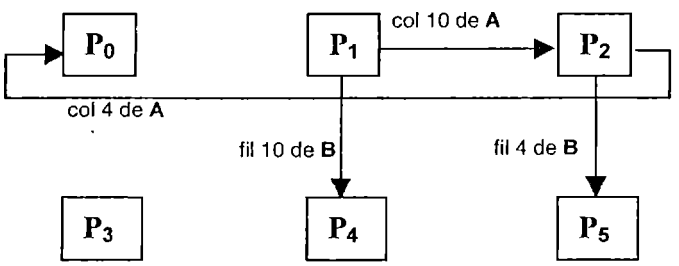
Paso 9



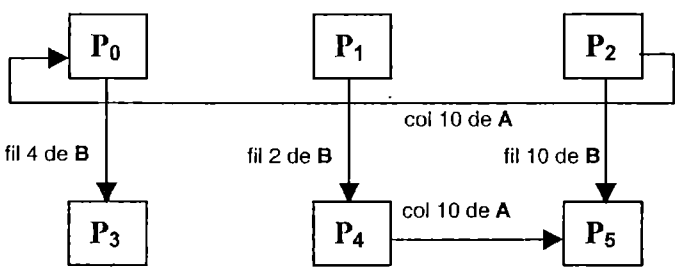
P0 espera col 1 de **A**
P1 multiplica col 7 de **A** con fil 7 de **B**
P2 multiplica col 1 de **A** con fil 1 de **B**
P3 multiplica col 1 de **A** con fil 1 de **B**
P4 espera fila 4 de **B**
P5 multiplica col 7 de **A** con fil 7 de **B**



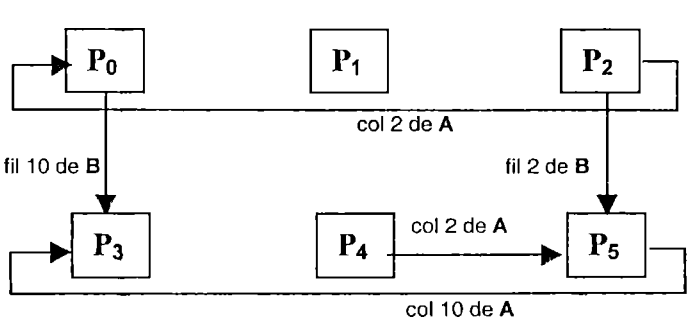
P_0 multiplica col 1 de **A** con fil 1 de **B**
P_1 multiplica col 4 de **A** con fil 4 de **B**
P_2 multiplica col 7 de **A** con fil 7 de **B**
P_3 multiplica col 7 de **A** con fil 7 de **B**
P_4 espera fila 4 de **B**
P_5 espera fila 4 de **B**



P_0 multiplica col 7 de **A** con fil 7 de **B**
P_1 espera col 10 de **A** y fila 10 de **B**
P_2 multiplica col 4 de **A** con fil 4 de **B**
P_3 espera fil 4 de **B**
P_4 multiplica col 4 de **A** con fil 4 de **B**
P_5 espera fila 4 de **B**



P_0 multiplica col 4 de **A** con fil 4 de **B**
P_1 espera col 2 de **A**
P_2 multiplica col 10 de **A** con fil 10 de **B**
P_3 espera fil 4 de **B**
P_4 multiplica col 10 de **A** con fil 10 de **B**
P_5 multiplica col 4 de **A** con fil 4 de **B**



P_0 multiplica col 10 de **A** con fil 10 de **B**
P_1 espera col 2 de **A**
P_2 multiplica col 2 de **A** con fil 2 de **B**
P_3 multiplica col 4 de **A** con fil 4 de **B**
P_4 multiplica col 2 de **A** con fil 2 de **B**
P_5 multiplica col 10 de **A** con fil 10 de **B**

Figura 4.6: Pasos intervientes en la comunicación pipelined de DIMMA.

4.4.2. Concepto de LCM

DIMMA incorpora el concepto de LCM. Nos referiremos a un cuadrado de LCMxLCM bloques como un bloque LCM. La matriz de la figura 4.3 puede ser vista como una matriz de 2x2 de LCM bloques. Un algoritmo paralelo, el cual el orden de ejecución puede ser entremezclarse la multiplicación de matrices y transposición de matrices, puede ser desarrollado para el primer bloque LCM. Entonces puede ser directamente aplicado a los demás bloques LCM, los cuales tienen la misma estructura y la misma distribución que el primer bloque LCM, esto es, cuando una operación es ejecutada en el primer bloque LCM, la misma operación puede ser realizada simultáneamente en los otros bloques LCM [Choi].

La idea del concepto de LCM es manejar simultáneamente diversas "columnas delgadas" de bloques de **A**, y el mismo número de "filas delgadas" de bloques de **B** de manera que cada procesador multiplica distintas "matrices delgadas" de **A** y **B** simultáneamente para obtener la mayor rendimiento de la máquina. En lugar de realizar un broadcast de una sola columna de **A** y una sola fila de **B**, una columna de procesadores puede realizar un broadcast de distintas (M_x) columnas de bloques de **A** hacia cada fila de procesadores. La distancia entre estas columnas de **A** es de LCM bloques. Al mismo tiempo, una fila de procesadores realiza un broadcast del mismo número de bloques de **B** hacia cada columna de procesadores, donde la distancia entre estas filas de **B** es de LCM bloques. Luego cada procesador ejecuta su propia multiplicación.

El valor de M_x puede determinarse teniendo en cuenta parámetros como el tamaño de bloque, espacio de memoria disponible, y características del procesador, como el rendimiento y velocidad de comunicación. Generalmente M_x se toma como K_{opt}/K_b, donde K_{opt} es el tamaño de bloque óptimo para el cálculo.

Por ejemplo, si P=2, Q=3, K_b=10, K_{opt}=20, y M_x= K_{opt}/K_b=2 (es decir que los procesos van a manipular dos columnas de **A** y dos filas de **B** al mismo tiempo). La primera columna de procesadores, P_0 y P_3, copia las columnas 0 y 6 de **A** a T_A, y realiza un broadcast de T_A hacia cada fila de procesadores. La primera fila de procesadores, P_0, P_1, y P_2, copia las filas 0 y 6 de **B** a T_B, y envía un broadcast de T_B hacia cada columna de procesadores. Luego cada procesador multiplica T_A con T_B para obtener **C**. A continuación, la segunda columna de procesadores, P_2 y P_4, copia las columnas 1 y 7 de **A** a T_A, y realiza un broadcast de T_A hacia cada fila de procesadores y la segunda fila de procesadores, P_3, P_4, y P_5, copia las filas 1 y 7 de **B** a T_B, y envía un broadcast de T_B hacia cada columna de procesadores. El producto de T_A y T_B se le suma a **C** en cada procesador. Notar que este procedimiento de comunicación y cálculo de T_A y T_B es el mismo que el mencionado anteriormente pero en este caso utiliza M_x columnas y filas de **A** y **B** respectivamente.

4.4.3. Pseudo-codigo de DIMMA

En la figura 4.7 se muestra un pseudo-código del algoritmo DIMMA. En el primer loop L1 representa el número de columnas de procesadores que enviarán todos

sus datos. En el segundo loop L2 representa una fila de procesadores al cual pertenece las filas de **B** que se enviarán en forma de broadcast. El loop L3 más interno es utilizado si k_b es menor que el k_{opt}, para enviar las filas o columnas restantes, a distancia LCM*M_X, que no entraron cuando se enviaron las M_X filas o columnas de bloques. La variable Lm representa la columna de **A** y la fila de **B** correspondiente al paso. En cada paso cada procesador si contiene la columna Lm de **A**, la copia a T_A junto con las siguientes M_X columnas que se encuentran a distancia LCM y las envía a derecha. Si no contiene esta columna Lm, recibe T_A de la izquierda y la envía a la derecha. En forma similar se trata a las columnas de **B** para obtener y comunicar T_B.

```
DIMMA (A,B,C)

C(i,j) = 0 ∀ i,j
Mx = K_opt/k_b
FOR L1 = 0 TO Q-1
      FOR L2 = 0 TO (LCM/Q) - 1
            Lx = LCM * MX
            FOR L3 = 0 TO (KG/LX)-1
                  Lm = L1 + L2 * Q + L3 * Lx
                  ObtenerYComunicarT_AT_B(A,B,T_A,T_B,Lm,LCM,(L3+1)*Lx-1)
                  C = C + T_A * T_B
            END
      END
END
ObtenerYComunicarT_AT_B (A,B,T_A,T_B,Lm,stride,fin)

IF la fila Lm (de bloques) de A esta en este procesador y
la columna Lm(de bloques) de B esta en este procesador THEN
            copiar en T_A las columnas de bloques de A desde Lm
 hasta fin desplazándose de a stride pasos
            copiar en T_B las filas de bloques de B desde Lm hasta
 fin desplazándose de a stride pasos
            enviar a derecha a T_A
            enviar a abajo a T_B
     END
ELSE IF la fila Lm (de bloques) de A esta en este procesador
     THEN
            copiar en T_A las columnas de bloques de A desde
 Lm hasta fin desplazándose de a stride pasos
            enviar a derecha a T_A
                  recibir T_B de arriba
                  enviar T_B abajo
     END
ELSE IF la columna Lm (de bloques) de B esta en este
 Procesador
 THEN
                  copiar en T_B las filas de bloques de B desde Lm
 hasta fin desplazándose de a stride pasos
            enviar a abajo a T_B
                  recibir T_A de la izquierda
                  enviar T_A a la derecha
            END
      ELSE
   recibir T_A de la izquierda
                  enviar T_A a la derecha
            recibir T_B de arriba
                  enviar T_B a abajo
      END
```

Figura 4.7: Pseudo-código del algoritmo DIMMA.

5. Diseño de Multiplicación de Matrices en Transputers

La multiplicación de matrices en la red de transputers se desarrollará utilizando el algoritmo DIMMA descripto anteriormente en el capítulo 4. La implementación de este algoritmo se corresponderá con una arquitectura SPMD (Single-Program Multiple-Data) [Kum94][Gei94][Thi95], en la cual, el mismo programa (DIMMA) se ejecutará simultáneamente en cada uno de los transputers. Una aplicación SPMD permite que el esquema de comunicación entre los procesos sea fijo y conocido a priori. Esto favorece el diseño del ruteo de los mensajes entre los procesadores para obtener resultados óptimos.

5.1. Ruteo de Mensajes

El envío de mensajes de tipo broadcast propios del algorimo DIMMA, se implementara adoptando el esquema de comunicación pipelined (ver capítulo 4). Con este esquema de comunicación, cuando un proceso envía un mensaje broadcast hacia toda su fila (o columna), este mensaje no se envía simultáneamente a todos los procesadores de esa fila (o columna). Por el contrario, el mensaje es enviado al próximo procesador de la fila (o columna), el cual tomará este mensaje y lo reenviará a su procesador contiguo en la fila (o columna), y así sucesivamente hasta que toda la fila (o columna) de procesadores reciba el mensaje. En la figura 5.1 se detalla el camino recorrido por un mensaje de tipo broadcast enviado por el procesador P_1 hacia toda una fila de cuatro procesadores. Puede observarse que un mensaje tardará n-1 pasos para llegar a todos los procesadores de la fila; donde n es la cantidad de columnas de procesadores.

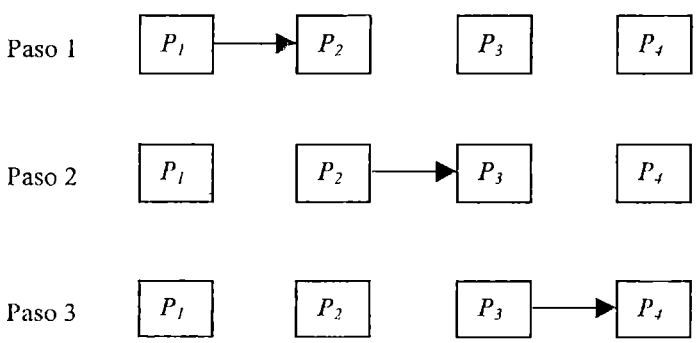


Figura 5.1: Envío de un mensaje broadcast con un esquema de comunicación pipelined.

Este tipo de comunicación será manejado por los procesos *Router*, que se ejecutarán en cada uno de los procesadores intervinientes en la multiplicación de matrices.

Existirán dos procesos *Router* por cada transputer, donde uno será el encargado de la comunicación referente a la fila de transputers (*Router* horizontal) y el otro será el encargado de la comunicación correpondiente a la columna de transputers (*Router* vertical), como lo indica la figura 5.2.

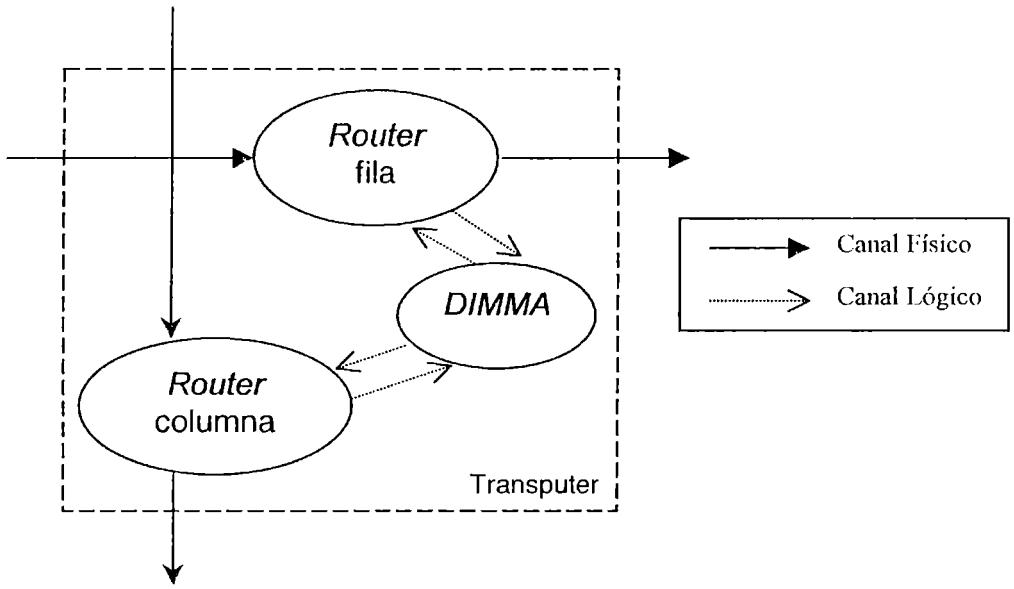


Figura 5.2: Routers dentro de un transputer.

Ambos *Routers*, pueden recibir mensajes tanto del proceso DIMMA que se ejecuta en el mismo transputer, como también del transputer anterior en la fila (o columna). Si el mensaje proviene del transputer anterior, este mensaje es enviado al proceso DIMMA y reenviado al próximo transputer en la fila (o columna). Si por el contrario el mensaje proviene del proceso DIMMA, este mensaje es solamente enviado al próximo transputer (en la fila o columna).

Puede notarse que la comunicación en ambos *Routers* se produce en un solo sentido (en una fila hacia la izquierda, y en una columna hacia abajo). Esto se debe a que, como se mencionó anteriormente, al ser una aplicación SPMD, el esquema de comunicación puede conocerse de antemano y no varía. En el caso del algoritmo DIMMA, por el *Router* de fila (horizonal) circularán sólo mensajes correspondientes a la matriz A; y por el *Router* de columna (vertical) sólo viajarán mensajes de la matriz B.

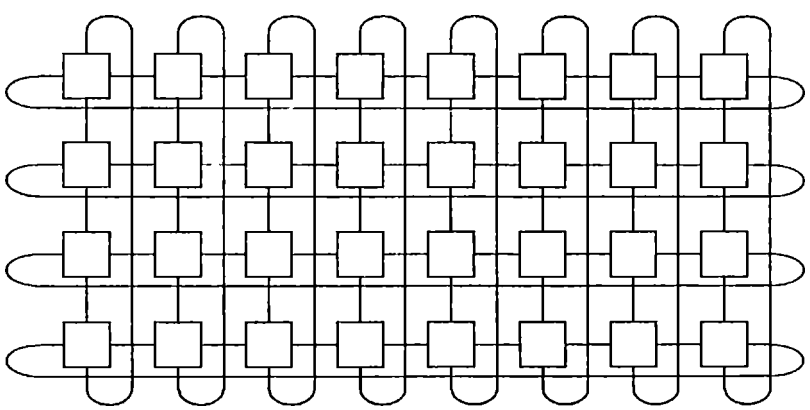


Figura 5.3: Grilla de 32 transputers ideal.

Teniendo en cuenta que se dispone de 32 transputers, idealmente la grilla de transputers debería estar conformada como lo indica la figura 5.3 en donde se puede observar que se forma un anillo en cada fila y columna de la grilla. El anillo formado en cada columna se encuentra físicamente conformado, por lo cual el número de filas siempre debe ser cuatro (porque en cada placa se encuentran cuatro transputers), y el número de columnas está dado por la cantidad de placas que conforman el hipercubo. Pero la topología de la figura 5.3 no se pudo lograr ya que a uno de los links del primer transputer llega una conexión correspondiente a la E/S, por lo que no se pudo cerrar el anillo correspondiente a esta fila (ver Apendice C: Configuración del hipercubo en forma de grilla). Al ser una aplicación SPMD, no se tuvo en cuenta esta fila como un caso especial, por lo que no se formó el anillo en ninguna de las filas. De esta manera las comunicaciones serán iguales para todos los procesadores independientemente de la fila en que se encuentre. En la figura 5.4 se muestra la grilla utilizada.

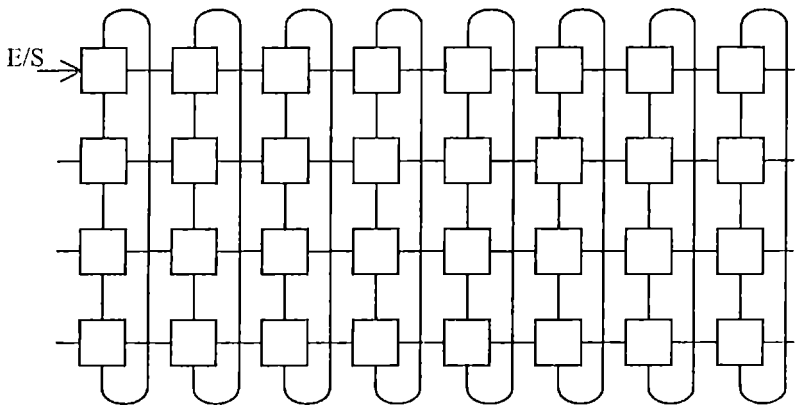


Figura 5.4: Grilla de 32 transputers utilizada.

Si se dispusiese de un transputer adicional, se prodría solucionar este problema, como se indica en la figura 5.5. El primer transputer de la grilla se utiliza en conjunción con el transputer que recibe la señal de E/S. Cuando este transputer recibe un mensaje por el link ubicado en la parte superior, lo redirecciona hacia el primer transputer. De esta manera, entre los dos transputers disponemos de los cuatro links necesarios para obtener tanto el anillo de la fila como el de la columna.

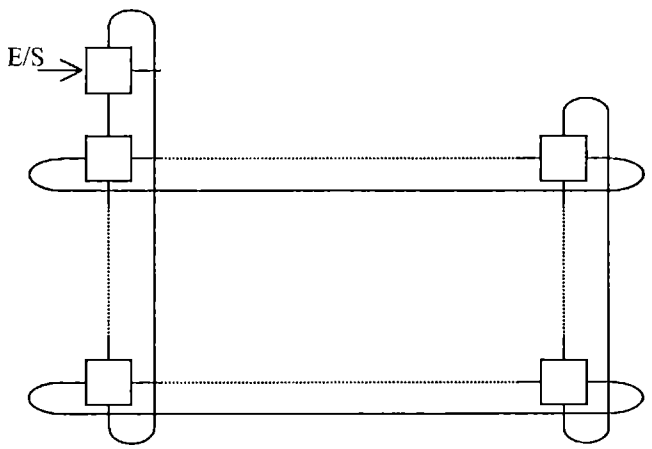


Figura 5.4: Utilización de 33 transputers para conformar la grilla.

La nueva conformación de la grilla mencionada anteriormente modifica el ruteo efectuado por los *Routers* horizontales. Ahora, cuando un procesador de la última columna debe enviar o reenviar un mensaje broadcast, no existe la conexión necesaria para enviárselo al transputer de la primera columna. Por esto el esquema de comunicación para los *Routers* horizontales (los *Routers* verticales no se modifican) se cambió de la siguiente manera:

- Cuando un transputer desea enviar un mensaje broadcast, lo hace en ambas direcciones. Es decir que lo manda al transputer de la columna anterior (si el transputer no se encontraba en la primera columna) y al transputer de la columna siguiente (si no estaba en la última columna).
- Cuando llega un mensaje broadcast al *Router* horizonal, además de enviárselo al proceso local como anteriormente, se reenvía al siguiente transputer en la dirección en que circulaba el mensaje.

En la figura 5.6 se muestra este nuevo esquema de comunicación para una fila, en donde P_3 envía un mensaje broadcast. Con este nuevo esquema circularán datos en ambas direcciones en una misma fila, lo cual influye en la aparición de nuevos casos de deadlocks que serán tratados más adelante. Además influirá en el rendimiento del algoritmo DIMMA, ya que está diseñado teniendo en cuenta una grilla como la que se presenta en la figura 5.3 y un mensaje puede llegar a un transputer en menos pasos que los esperados. Por ejemplo, en la figura 5.6, P_1 debería recibir el mensaje en el paso 4, pero con este esquema lo recibe antes, en el paso 2.

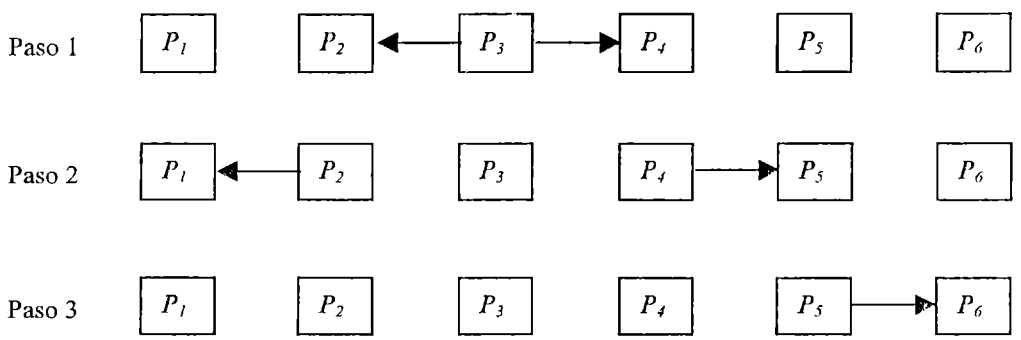


Figura 5.6: Envío de un mensaje broadcast con el nuevo esquema de comunicación.

5.2. Procesos Master y Slave.

Las matrices a multiplicar en la red de transputers pueden ser recibidas del exterior o simplemente recibir parámetros a partir de los cuales cada transputer generará sus matrices parciales. En cualquiera de los casos debe existir un proceso Master encargado de distribuir las matrices o los parámetros recibidos y luego recibir los resultados.

En todos los transputers además se ejecutará un proceso Slave, que inicialmente recibirá las matrices parciales o los parámetros para generarlas, luego ejecutará el algoritmo DIMMA, y por último retornará sus resultados. En el transputer donde se ejecute el proceso

Master también coexistirá un proceso Slave, ya que no se dispone de transputers adicionales.

6. Implementación de la Multiplicación de Matrices en Transputers

La multiplicación de matrices sobre la red de trasnputers se desarrolló mediante la implementación del algoritmo DIMMA. Este algoritmo opera junto con dos procesos *Routers*, encargados de la circulación de mensajes en la grilla de procesadores.

Se utilizó una grilla de 4x8 procesadores con las características mencionadas en la sección 5.2. La numeración asignada a cada transputer se encuentra en la figura 6.1.

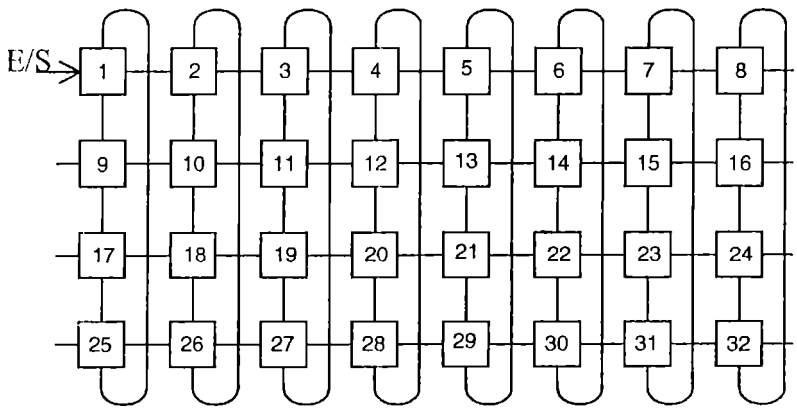


Figura 6.1: Numeración de la Grilla de transputers.

6.1. Implementación del Router

En cada transputer, coexisten dos procesos *Router*: el *Router* horizontal y el *Router* Vertical. El primero es el encargado de manejar los mensajes de tipo broadcast de una fila, en tanto que el segundo maneja los mensajes broadcast de una columna. Estos *Routers* están diseñados para aplicaciones que utilicen un esquema de comunicación de tipo broadcast pipelined.

La comunicación entre un *Router* y los procesos que se ejecutan en el mismo transputer se lleva a cabo mediante la utilización de canales de software (CS) [Thi95][Log94b]. Por cada proceso que se ejecute en un mismo transputer y desea utilizar el *Router* se requerirá de un canal de entrada y un canal de salida. Por el canal de entrada llegarán todos los mensajes que arriban al *Router*, en tanto que por el canal de salida el proceso envía los mensajes de tipo broadcast.

Por otra parte, la comunicación de un *Router* con otro *Router* situado en un transputer contiguo (en la fila o columna) se lleva a cabo por medio de canales físicos (CF) [Thi95][Log94b]. Los canales utilizados por cada *Router* se detallan en la figura 6.2.

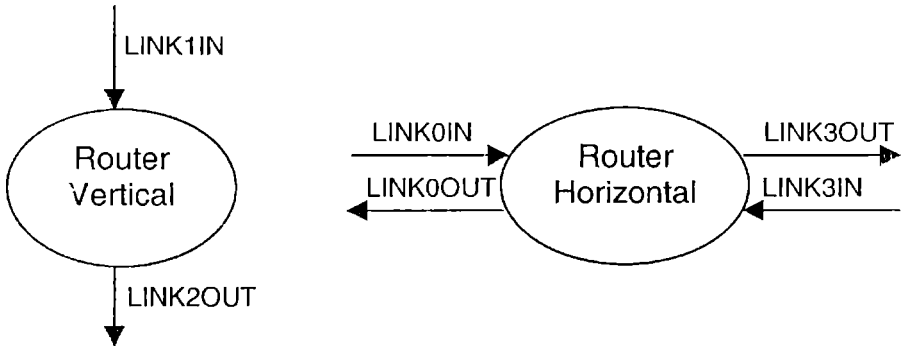


Figura 6.2: Canales Físicos utilizados por los *Routers*.

Puede observarse en la figura 6.2 que el *Router* vertical recibe y envía mensajes por un solo canal, en tanto que el *Router* horizontal lo hace por dos canales. Esto se debe a que en el *Router* horizontal circulan mensajes en ambas direcciones por el esquema de comunicación adoptado, por la falta de un enlace entre el último transputer de cada fila con el primero (como se detalla en la sección 5.2).

6.1.1. Deadlocks

La forma adoptada para prevenir la aparición de deadlocks es asegurando que la condición de espera circular no se cumpla, ya que la espera circular es una condición necesaria para que ocurra deadlock.

En el caso de la implementación de los *Routers* en los transputers, una espera circular puede producirse en dos casos: entre el *Router* y los procesos locales, y entre dos o más *Routers*.

Deadlocks entre el Router y los procesos locales

La espera circular causante de este deadlock puede surgir cuando un *Router* recibe un mensaje para el proceso y se lo envía, y al mismo tiempo, el proceso desea enviar un mensaje a otro transputer por medio del *Router*. Al ser la comunicación bloqueante tanto para el envío como para la recepción, el *Router* esperará a que el proceso reciba el mensaje y el proceso esperará a que el *Router* reciba su mensaje, por lo cual ambos quedan esperando por un evento que nunca ocurrirá.

La estrategia elegida para evitar este tipo de situaciones consiste en asegurar que el *Router* nunca quede bloqueado esperando una comunicación. Esto se logró mediante la implementación de un buffer de mensajes pendientes que funciona de la siguiente manera:

- Cuando un proceso necesita recibir un mensaje, primero envía un requerimiento de recepción (con el tipo de mensaje a recibir) al *Router* que se encuentra en el mismo nodo (mediante ChanOut al CS de salida). Luego ejecuta un ChanIn en el CS de entrada, quedándose bloqueado hasta recibir el mensaje esperado.

- Cuando un proceso necesita enviar un mensaje, lo hace mediante su CS de salida, y este mensaje es inmediatamente recibido por el *Router*. Esto hace que el envío de mensajes ahora no sea bloqueante (no tiene que esperar que el receptor del mensaje lo reciba).
- Cuando un *Router* recibe un mensaje, determina si es para el proceso que se encuentra en el mismo transputer o para otro transputer. Si el destino es otro transputer, el mensaje es enviado al proceso buffer de salida, que se encarga de hacer que este mensaje llegue a su destino. Si el destino del mensaje es el proceso local, el *Router* verifica si existía algún requerimiento (del mismo tipo que el mensaje), y de ser así lo envía inmediatamente. Si no existe ningún requerimiento, se almacena el mensaje en un buffer interno, donde permanece hasta que se produzca un requerimiento.
- El buffer de salida envía los mensajes por los canales físicos hacia otro transputer, ayudando a que el *Router* no quede bloqueado esperando a que el otro transputer reciba la comunicación. Este buffer contendrá a lo sumo un mensaje.
- Los mensajes que llegan a un transputer son leídos por un buffer de entrada, y son reenviados al *Router* local. Al igual que el buffer de salida, este buffer ayuda a que el *Router* no quede bloqueado. El tamaño de este buffer está determinado por una constante definida por el usuario del *Router*, y generalmente es mayor que uno.

Deadlocks entre Routers de distintos transputers

Este tipo de deadlocks se produce como consecuencia de la circulación de mensajes en direcciones opuestas. Por ejemplo, puede suceder que un *Router* horizontal *R1* reciba un mensaje *A* desde la izquierda, y lo tenga que reenviar a derecha hacia el *Router R2*. Si en el mismo momento, el *Router R2* quiere enviarle otro mensaje *B* a *R1*, nunca podrá hacerlo, porque *R1* está bloqueado tratando de enviarle el mensaje *A*, y se produce una espera circular. Esta condición de espera circular debería ser evitada para que no se produzcan estos casos de deadlocks.

Estos deadlocks se solucionaron mediante los buffers de entrada y salida ya mencionados anteriormente. El buffer de salida puede contener un solo mensaje y es el encargado de determinar por cuál link sale un mensaje (cabe destacar que en un *Router* vertical solo hay un posible link de salida, en tanto que en un *Router* horizontal existe un link de salida hacia cada dirección). De esta manera, el *Router*, cuando debe enviar un mensaje hacia otro transputer, en realidad lo envía hacia el buffer de salida para no quedar bloqueado esperando que el *Router* del otro transputer reciba el mensaje.

El buffer de entrada, por el contrario, puede contener más de un mensaje. La función de este buffer de entrada es recibir todos los mensajes provenientes de los transputers vecinos sin necesidad de que se bloquee el *Router*. Por ejemplo, como se muestra en la figura 6.3, puede suceder que un transputer *A* envíe una gran cantidad de mensajes en forma continua hacia el transputer *B* y el transputer *B* debe enviar otros mensajes hacia el transputer *A*. El Router de *B* estará ocupado enviando los mensajes a *A*, por lo cual los mensajes provenientes de *A* deben almacenarse en el buffer de entrada de *B*, el cual debe tener la capacidad suficiente para almacenar todos estos mensajes y no quedar bloqueado. Si este buffer se bloquea por no tener la capacidad suficiente, se bloqueará también el buffer de salida de *A* con el próximo mensaje, y luego con otro mensaje se bloqueará el *Router* de *A* (porque no puede enviar al buffer de salida), y al estar bloqueado este Router de *A* no

recibirá los mensajes de *B*, por lo que se bloqueará también el buffer de entrada de A y luego el *Router* de *B*.

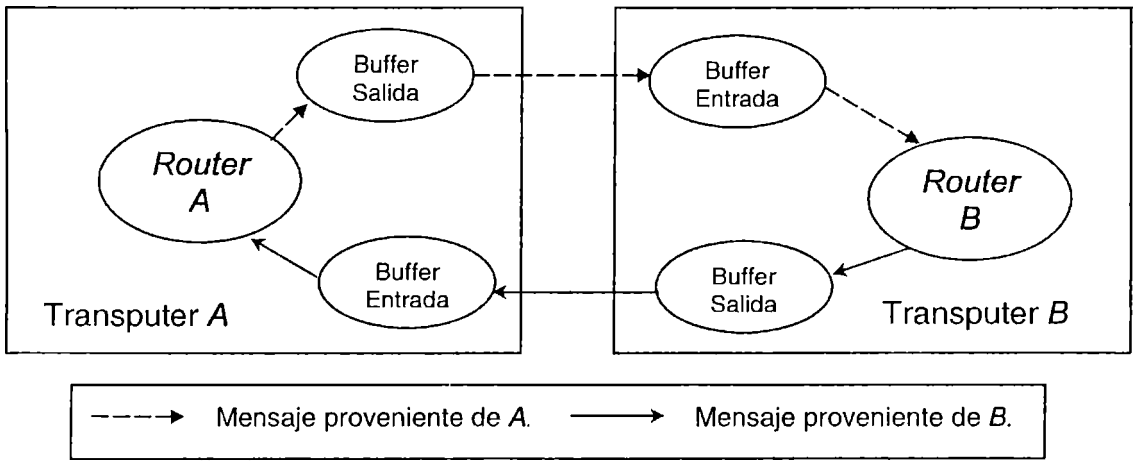


Figura 6.3: Ejemplo de posible espera circular.

Consideraciones acerca de los buffers

El tamaño de los buffers de mensajes pendientes y de entrada depende de la aplicación que utilice el *Router*. Esta dependencia se debe a factores como la cantidad, frecuencia y tamaño de los mensajes a enviar. Cuanto más grandes sean estos buffers, menor es la posibilidad de producirse una espera circular y por ende deadlock. Pero en contrapartida, al ser grandes estos buffers se ocupa mucha memoria (recordar que estos buffers existen para cada *Router*, y se tienen dos *Routers*: uno horizontal y otro vertical), afectando a los procesos propios de la aplicación que utilicen el *Router*.

La comunicación entre el Router y los buffers de entrada y salida se realiza por medio de canales de software.

El buffer de entrada está implementado utilizando el algoritmo de productores/consumidores utilizando semáforos [And91][Sil91]. Este buffer está conformado por dos procesos: uno productor y otro consumidor. El proceso productor lee los mensajes del canal físico de entrada y lo deposita en el buffer, en tanto que el consumidor lee los mensajes del buffer y se los envía por un canal de software al *Router*.

En la figura 6.4 se muestran los procesos que conforman tanto el *Router* horizontal como el vertical en un transputer.

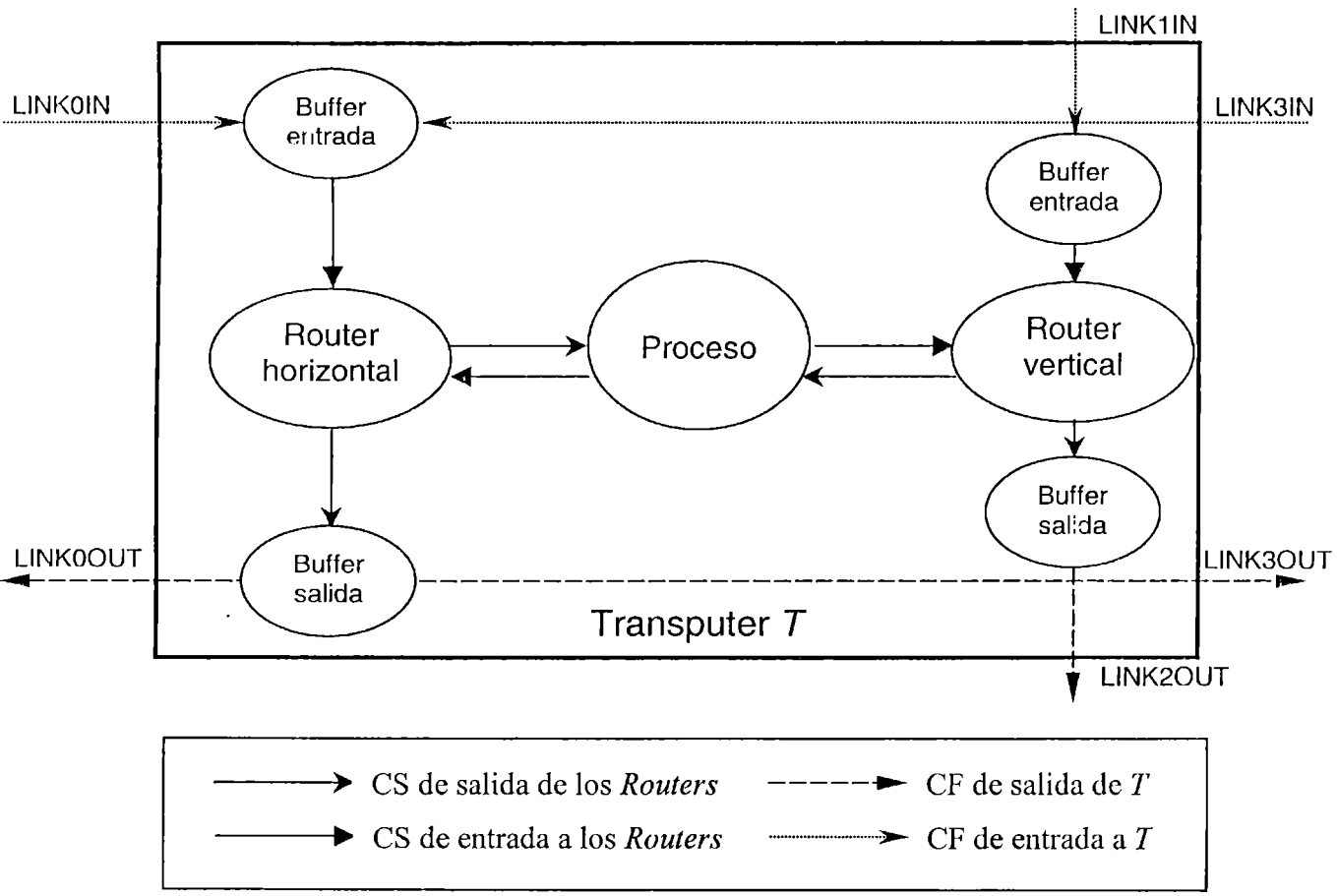


Figura 6.4: Procesos dentro de un transputer.

6.1.2. Canales virtuales

Los canales virtuales permiten comunicar cualquier transputer con cualquier otro [Log94a][Log94b][Thi95]. El problema de estos canales virtuales es que no se conoce cómo realiza el ruteo internamente y tampoco cuánta memoria consume.

Estos canales virtuales producen un overhead innecesario, ya que se tiene en cuenta la posibilidad de cualquier tipo de comunicación (siendo ineficiente en cuanto a tiempo y espacio). Para una aplicación SPMD, como en nuestro caso, la forma de comunicación es conocida y estática por lo cual se puede reducir este overhead. Por estos factores, la comunicación entre los procesos de los distintos transputers no fue implementada con canales virtuales.

6.1.3. Detalles de implementación del Router

Creación de los Routers

La creación de los procesos *Routers* se efectúa mediante la rutina `ProcAlloc()` provista por la librería de C paralelo para transputers, referenciando la siguiente función:

```
void router (Process *p, Channel *canalIn[NROPROCESOS],
Channel *canalOut[NROPROCESOS], int tipoRouter)
```

donde *canalIn* es un arreglo de los canales de entrada al *Router* pertenecientes a los procesos de un mismo transputer, *canalout* es un arreglo con los canales de salida del *router* hacia estos procesos y *tipoRouter* identifica el tipo de *Router* a crear. Los tipos posibles están definidos por las constantes HORIZONTAL (crea *Router* horizontal) y VERTICAL (crea *Router* vertical).

En el siguiente ejemplo se muestra la creación de un *Router* horizontal procRouterHoriz:

```
procRouterHoriz = ProcAlloc(router, WS_SIZE, 3,
          canalesOutHoriz, canalesInHoriz, HORIZONTAL)
```

Una vez creado el proceso *Router*, se procede a su ejecución mediante una llamada a la rutina `ProcRunHigh()`. Los *Routers* son ejecutados en alta prioridad, ya que cuando se produce una E/S por los canales, estos *Routers* deben atenderla inmediatamente, sin tener que esperar que otro proceso le ceda el procesador.

Cabecera de un mensaje

A todo mensaje que circula en la red de transputers por medio de los *Routers* se le agrega una cabecera conformada por la siguiente información:

Información	Descripción
Tipo Mensaje	Tipo de mensaje enviado. Su valor debe ser mayor que cero. El tipo cero esta reservado para los mensajes de tipo requerimiento de recepción.
Broadcast	Indicador de si el mensaje es bradcast
Nodo Origen	Nodo en la red de transputers que generó el mensaje
Proceso Origen	Proceso dentro de Nodo Origen que generó el mensaje
Nodo Destino	Nodo destino del mensaje
Proceso Destino	Proceso destino del mensaje

Constantes utilizadas por el Router

Para la correcta utilización de los *Routers* se deben definir las siguientes constantes, que dependen de la aplicación:

Nombre	Descripción
ELEMTYPE	Tipo de datos de los elementos de un mensaje. Puede ser double o float.
CANTDATOS	Cantidad de datos contenidos en un mensaje (sin tener en cuenta la cabecera)
TBUFFERH	Cantidad de mensajes que conformarán el buffer de mensajes pendientes del Router Horizontal
TBUFFERV	Cantidad de mensajes que conformarán el buffer de mensajes pendientes del Router Vertical
TAMBUF	Cantidad de mensajes que conformarán el buffer de Entrada

Además se definieron las siguientes constantes:

Nombre	Descripción
TAMANIODATOS	Cantidad de datos contenidos en un mensaje más la cabecera
TAMENIOENVIO	Cantidad de bytes de un mensaje (teniendo en cuenta la cabecera)
TAMANIOBLOQUE	Cantidad de bytes de un mensaje (sin tener en cuenta la cabecera)

Envío y Recepción de mensajes por medio del Router

Para el envío y recepción de mensajes por medio del Router se definió la estructura `datosMsg` de la siguiente manera:

Definicion de la estructuta `datosMsg`	
int tipoMsg	Define de qué tipo es el mensaje (su valor debe ser mayor que cero)
int nodoOrigen;	Número de transputer origen del mensaje
int procOrigen;	Identificación del proceso residente en nodo origen que envía los datos
int nodoDestino;	Número de transputer destino del mensaje
int procDestino;	Identificación del proceso residente en nodo destino que recibirá los datos
ELEMTYPE *info;	Datos a enviar. Estos datos deben alocarse externamente y debe ser de CANTDATOS elementos

El envío de mensajes se realiza mediante la siguente rutina:

```
void enviar(Channel *c,struct datosMsg d,
            ELEMTYPE *datos, int broadcast)
```

donde *c* es el canal de entrada al *Router* (puede ser tanto horizontal como vertical), *d* es la información sobre el mensaje a enviar, *datos* es un área de memoria auxiliar y *broadcast* es un indicador de si el mensaje es de tipo boradcast (valor 1) o no (valor 0). El área de

memoria auxiliar *datos*, debe asignarse exteriormente y su tamaño debe ser TAMANIODATOS. Esta área de memoria es utilizada para la ejecución de las primitivas de comunicación y se crea externamente para tener un mayor control de la utilización de la memoria y que no se asigne y libere esta memoria constantemente.

La recepción de mensajes se realiza mediante la rutina:

```
void recibir(Channel *in, Channel *out,
int pReceptor, struct datosMsg *d, ELEMTYPE *datos)
```

donde *in* es el canal de entrada del proceso (salida de *Router*), out es el canal de salida del proceso (entrada del *Router*), *datos* es un área de memoria auxiliar como en la rutina `enviar` y *preceptor* es un indentificador del proceso que espera recibir el mensaje. En la estructura *d* se indican las características de los datos a recibir (por ejemplo tipo de mensaje) y se retornan en esta misma estructura los datos recibidos.

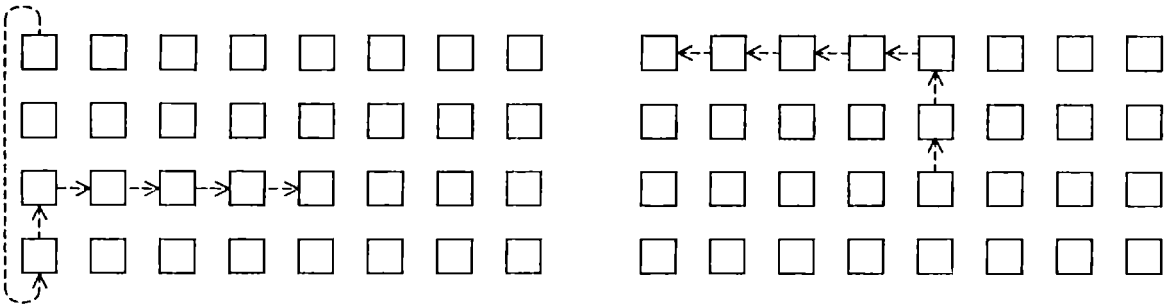
Internamente, en esta rutina, primero se envía un requerimiento de recepción (por canal *out*) al *Router* y luego se espera recibir el mensaje (por canal *in*), de acuerdo a lo explicado en la sección 6.1.1.

Mensajes no broadcast

En el algoritmo DIMMA, todos los mensajes involucrados en el proceso son de tipo broadcast. Pero hay que tener en cuenta los casos especiales en que hay que distribuir las matrices a multiplicar en la red de transputers y luego juntar los resultados. Si bien estos casos no forman parte del algoritmo DIMMA propiamente dicho, hay que tenerlos en cuenta en el ruteo de mensajes.

Hay dos tipos de mensajes no broadcast: los que van del nodo Master hacia cada Slave (distribución de las matrices), y los que van de cada Slave hacia el Master (obtención de la matriz resultado).

Los mensajes no broadcast deben ser enviados por el *Router* vertical. Cuando un *Router* vertical recibe un mensaje que no es broadcast, lo reenvia hacia el nodo de arriba, hasta llegar a la fila del nodo destino. Una vez alcanzada esta fila, el mensaje pasa al *Router* horizontal y mediante éste llega a su destino. En la figura 6.5 a) se muestra el viaje de un mensaje no broadcast desde el nodo Master hacia un Slave ubicado en la fila 3, columna 5. En tanto que en la figura 6.5 b) se muestra el viaje inverso, desde el Slave mencionado anteriormente hacia el Master.



a) desde Master a Slave b) desde Slave a Master

Figura 6.5: envío de mensajes no broadcast

Hay que tener en cuenta que en DIMMA los mensajes de distribución se efectúan al inicio del proceso, y los de obtención al final. Debido a esto nunca circularán en la red mensajes de tipo broadcast y no broadcast al mismo tiempo.

Si bien estos mensajes no broadcast fueron pensados para la aplicación concreta de DIMMA, pueden utilizarse para cualquier otra aplicación en la cual no se mezclan mensajes broadcast con mensajes no broadcast.

6.2. *Implementación DIMMA*

El algoritmo DIMMA fue implementado siguiendo el pseudocódigo de la figura 4.7 para una malla de PxQ procesadores. Cuando un proceso envía bloques de la matriz A, lo hace por medio del *Router* horizonal. En tanto que para enviar bloques de B lo hace por el *Router* vertical. Del mismo modo, cuando recibe bloques de A o B lo hace por el *Router* horizontal y vertical, respectivamente.

6.2.2. Dimensiones de las Matrices

Dadas las matrices A y B de dimensiones MxK y KxN se obtiene la matriz resultado C de dimensión MxN. Las matrices A, B y C son subdivididas en bloques de M_bxK_b, K_bxN_b y M_bxN_b, respectivamente.

Puede suceder que las dimensiones de las matrices no sean divisibles por el tamaño de bloque. En este caso las dimensiones son modificadas (se agrandan) para que sean múltiplos del tamaño del bloque, y los últimos bloques son rellenados con ceros. Al ser los procesadores homogéneos, el hecho de agrandar las matrices para que sean múltiplos del tamaño de bloque no influye en la performance final del algoritmo. Esto se debe a que al ser todos los procesadores iguales, si uno tiene menos elementos terminará sus cálculos primero que los demás, pero igual tiene que esperar a que los otros (que tienen más elementos) terminen. Por lo cual el tiempo será el mismo que en el caso en que todos tengan la misma cantidad de elementos y terminen todos en el mismo momento. Si los procesadores fuesen heterogéneos, este rellenado sí tendría influencia en el rendimiento, porque todos los procesadores no terminan al mismo tiempo en realizar cálculos con la misma cantidad de elementos.

6.2.3. Tamaños de mensajes

El tamaño de mensaje para la utilización del *Router* es fijo. Por esto el tamaño adoptado es el del mayor mensaje que circule. Los mensajes que circulan en DIMMA se corresponden con *TA* y *TB* (ver capítulo 4). Pero el tamaño de cada *TA* depende de la fila de procesadores (por la distribución cíclica de las matrices), y de la misma manera, el tamaño de *TB* depende de la columna de procesadores. Por esto, el tamaño de mensaje utilizado en DIMMA es el máximo entre el mayor *TB* y el mayor *TA*.

6.2.4. Implementación Master/Slave

El proceso Master es el encargado de distribuir las matrices a multiplicar (o los parámetros a partir de los cuales se generan las matrices) y de recibir los resultados de cada Slave. Los procesos Slave se encuentran en todos los transputers de la red y ejecutan el algoritmo DIMMA.

El proceso Master se sitúa en el primer transputer (porque es el único con capacidad de E/S). Inicialmente distribuye las matrices hacia todos los Slaves, luego ejecuta DIMMA (pasa a ser un Slave más) y por último recibe los resultados de cada Slave. Los procesos Master y Slave que se ejecutan en el mismo transputer podrían ser procesos separados, pero la ejecución de uno seguido del otro garantiza que cuando se inicie DIMMA, ya todos los Slave tengan sus matrices parciales. Esto se debe a que el proceso DIMMA del primer transputer es el único que tiene la primera porción de *A* y *B* para enviar, por lo que es el encargado de iniciar el algoritmo.

Inicialización de los procesos Slave

Inicialmente, se reciben las dimensiones de las matrices a multiplicar en el proceso Master. Luego, estas dimensiones deben ser distribuidas a cada slave interviniente en la multiplicación. Esto se efectúa en fase inicial que se ejecuta al inicio de cada slave y al inicio del master. En esta fase, el proceso master comienza enviando la información al próximo transputer situado en su columna. Cada slave recibe esta información, y si está en la misma columna que el proceso master, manda estos datos hacia el próximo transputer en su columna y hacia el próximo transputer en su fila. Si el slave no se encuentra en la misma columna que el proceso master, simplemente reenvía la información hacia el próximo transputer en su fila. De esta manera todos los transputers reciben las dimensiones de las matrices a multiplicar en esta fase.

Pasaje de Parámetros

El algoritmo de multiplicación de matrices que se ejecuta en los transputers debe recibir en algún momento las matrices a multiplicar o los parámetros para generar estas matrices. Esta información es leída por el proceso Master por medio de archivos, junto con las dimensiones de las matrices. Del mismo modo, la matriz resultado obtenida es enviada a un archivo de salida (para más información ver el capítulo 7).

Distribución de las matrices

Tanto las matrices A como B son obtenidas leyendo de la entrada estándar de a filas de bloques. Una vez obtenida una fila de bloques, cada bloque es enviado al transputer correspondiente según la distribución cíclica.

Obtención *del resultado*

Una vez terminado el algoritmo DIMMA, se envía un mensaje a cada transputer solicitándole sus resultados y luego se reciben todos los bloques resultados correspondientes a ese transputer Estos bloques resultados son enviados a la salida estándar del proceso DIMMA.

Por cada fila de bloques del resultado correspondiente a un transputer que se recibe, se envía un mensaje solicitando la próxima fila. Es decir, que se envía un mensaje por cada fila de bloques a recibir por cada transputer. Esto se debe a que si llegan todos los bloques juntos de un transputer (o de todos los tranputers), los buffers de mensajes pendientes (que son pequeños) producen un cuello de botella y se saturan muy pronto, provocando un desbordamiento de los buffers, por lo cual implementó de esta manera, asegurando que esta situación no se produzca.

7. Integración de la Red de Transputers con PVM.

Dada una red de computadoras heterogéneas interconectadas por medio de PVM, se desea que el hipercubo de 32 transputers se integre a esta red global. En particular, esta integración se analizará para la aplicación concreta de la multiplicación de matrices.

En la red PVM se ejecutará un algoritmo (global) de multiplicación de matrices. Uno de los nodos intervinentes en esta multiplicación estará conformado por la red de transputers correspondiente al hipercubo.

Esta integración de los transputers a la red global se realizará en forma transparente. Esto significa que el hipercubo se comportará como una caja negra a la cual le llegan dos matrices y obtiene la matriz resultado, independientemente de que forme parte de otro algoritmo global. Por esto, el algoritmo ejecutado en la red global no necesariamente debe ser el mismo que el algoritmo que se ejecuta en la red de transputers.

7.1. Forma de Integración

Para poder integrar la red de transputers a la red heterogénea formada con PVM, es necesario agregar a la red de PVM la computadora a la cual esta conectado el hipercubo.

Uno de los procesos Slave que forma parte del algoritmo de multiplicación de matrices global, en lugar de efectuar sus cálculos parciales (que son multiplicaciones de matrices más pequeñas) por sí mismo, ejecutara estás multiplicaciones en la máquina paralela conformada por la red de transputers.

Llamaremos *host del hipercubo* a la computadora conectada al hipercubo de 32 transputers. En la figura 7.1 se muestra la forma en que se integra la red de transputers a la red de PVM. Cuando el Slave que se encuentra en el host del hipercubo necesite realizar sus multiplicaciones parciales, enviará las matrices a multiplicar al hipercubo. El hipercubo, una vez recibidas las matrices, las multiplica (utilizando los 32 transputers) y acumula el resultado. Este proceso se repite hasta que el Slave realice todas sus multiplicaciones parciales, momento en el cual le pide al hipercubo los resultados obtenidos.

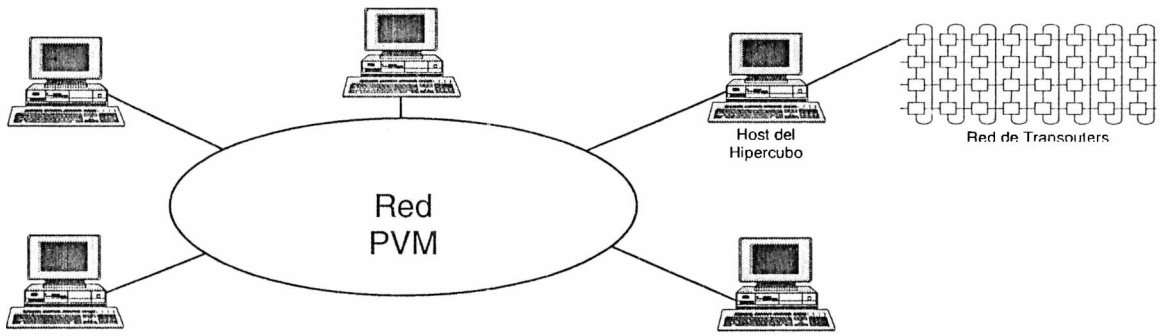


Figura 7.1: Integración del hipercubo en PVM

7.2. Procesos en el host

El proceso slave de la multiplicación global que se ejecuta en PVM, para la utilización de la red de transputers, requiere la creación de otros procesos, que se pueden visualizar en la figura 7.2. El primer proceso generado es *Transp*, el cual es el encargado de invocar la multiplicación de matrices en los transputers, junto con otro proceso *out*, que recibirá los resultados de la multiplicación. Esto se efectúa mediante una llamada al sistema de la forma:

LD-NET dimma | out

donde `LD-NET dimma` ejecuta la multiplicación de matrices en los transputers (ver apéndice A: Lenguaje C paralelo).

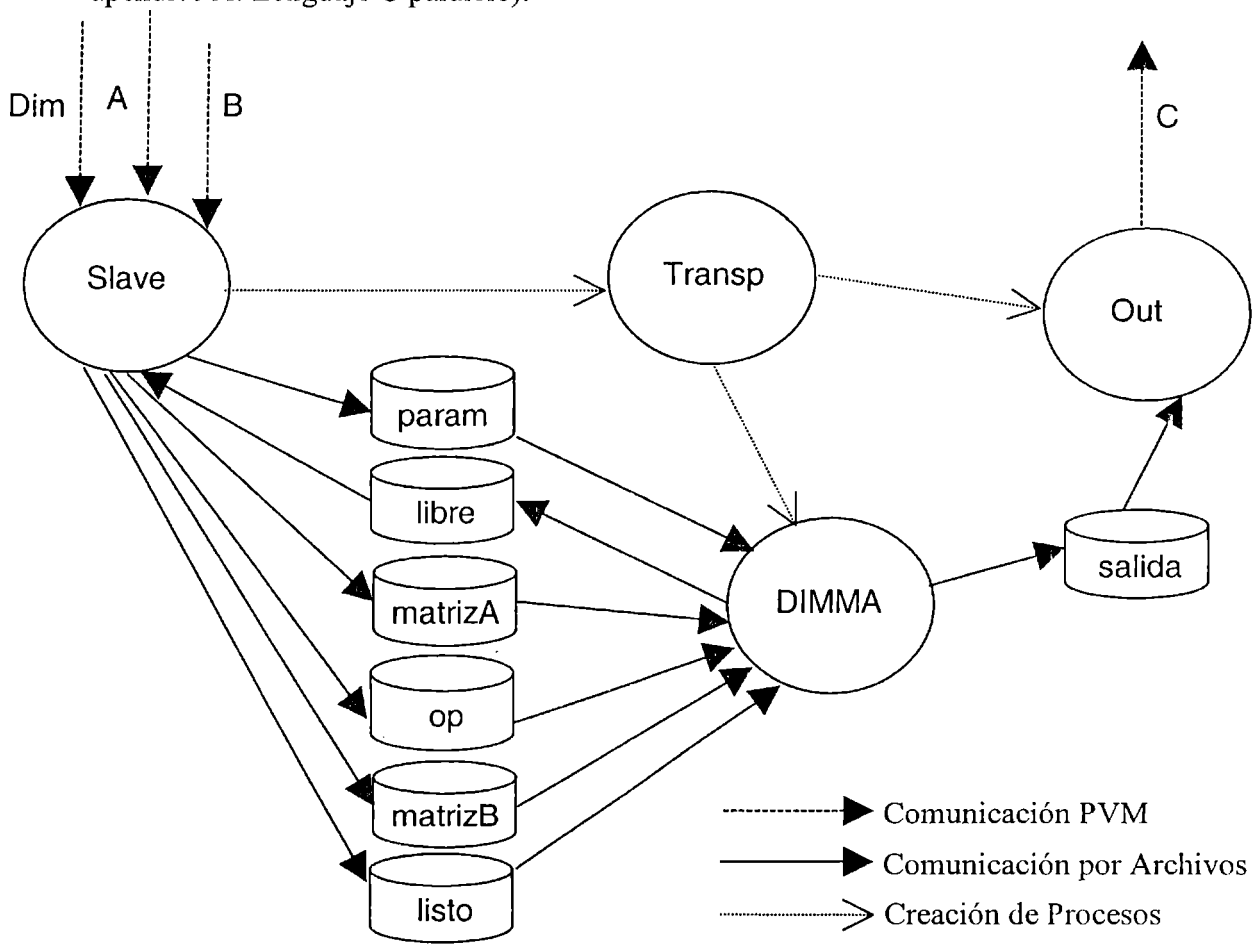


Figura 7.2: Procesos intervinientes en la Integración del hipercubo en PVM y la comunicación entre éstos.

7.2.1. Comunicaciónes entre procesos en el host y los transputers

Cuando se inicia el proceso slave de la multiplicación de matrices global en el host del hipercubo, debe recibir (por medio de comunicación por PVM) las dimensiones de las matrices a multiplicar. Estas dimensiones son enviadas al transputer con capacidad de E/S por medio de un archivo de parámetros.

Para cada multiplicación de matrices parcial, el slave recibe por medio de PVM las matrices a multiplicar, pero para iniciar la multiplicación debe sincronizarse con los transputers. Por esto se utiliza el archivo `libre`, el cual indica si los transputers se encuentran listos para multiplicar. Una vez que los transputers están listos, el slave escribe la matriz *A* en el archivo `matrizA`, y en el archivo `op` (de operación) indica a los transputers que tienen que multiplicar y que ya puede leer la matriz *A*. A continuación escribe la matriz *B* en el archivo `matrizB`, y en el archivo `listo` indica a los transputers que ya pueden leer la matriz B y comenzar a multiplicar. Nótese que mientras se distribuye la matriz *A* en los transputers, el slave de la multiplicación global puede ir escribiendo la matriz *B* simultáneamente. Este proceso se repite para cada multiplicación parcial que efectúa el slave, y el resultado de cada una de estas multiplicaciones se acumula en cada uno de los transputers.

Una vez que se efectuaron todas las multiplicaciones, el slave indica por medio del archivo `op` (el mismo que se utilizaba para indicarle a los transputers que debían multiplicar) que debe retornar el resultado. El resultado de los transputers es escrito al archivo *salida* y es leído por el proceso *out*. El proceso *out*, recibe los bloques de cada transputer, arma la matriz *C* resultado y la envía por medio de PVM al proceso que corresponda. El armado de la matriz *C* se efectúa en este proceso porque los transputers no contienen memoria suficiente para contener la matriz completa.

Inicialmente el resultado de los transputers era enviado a la salida estándar y *out* los recibía por su entrada estándar. El problema surgía debido a que tanto la salida estándar como la entrada estándar (*stdin* y *stdout*) son abiertos como archivo de texto. Al escribir los resultados (no se escriben como texto), en ciertas ocasiones ocurría el caso en el cual en estos datos se formaba el carácter EOF, por lo cual el proceso *out* no recibía todos los datos. Por esto se decidió por utilizar un archivo intermedio.

La sincronización que se produce entre el slave y los transputers por medio de archivos se produce creando el archivo, para el caso que se quiera indicar alguna condición; o esperando que exista el archivo, cuando hay que esperar por una condición. En el caso especial del archivo *op*, el slave además de crearlo escribe en este archivo la operación a efectuar, que puede ser multiplicar o retornar el resultado.

Para esta implementación concreta de la integración se tuvo en cuenta el caso más común de los algoritmos de multiplicación de matrices en donde en cada paso se recibe una porción de la matriz *A* y otra de la matriz *B* y se efectúan los cálculos parciales. Esta implementación puede modificarse fácilmente para el caso de algún algoritmo que requiera

otro método de distribución de las matrices *A* y *B* en cada slave, por ejemplo existen algoritmos que no requieren enviar la matriz *A* en cada paso.

7.2.2. Forma de integración ideal

La mejor forma de implementar la integración entre PVM y los transputers sería mediante la utilización de un triple pipe de la forma:

In | LD-NET dimma | out

donde `LD-NET dimma` y `out` son tal cual se describieron anteriormente y el proceso `In` recibiría las matrices por PVM y las enviaría por la salida estándar a los transputers. De esta manera quedarían sincronizados, porque cada vez que el slave tiene que hacer una multiplicación, manda por medio de `In` las matrices a los transputers y estos transputers las multiplican y esperan por las otras matrices por medio de la entrada estándar.

La causa por la cual no se implementó de esta manera es que los pipes en el sistema operativo DOS no están implementados como tal. Cada proceso interviniente en el pipe se ejecuta (en orden de aparición) hasta terminar y recién en este momento comienza el próximo proceso. Es decir que se ejecutan secuencialmente y no en forma concurrente como se esperaría que sea.

7.2.3. Mejora en la integración

La integración entre PVM y los transputers implementada es bastante primitiva por la necesidad de comunicarse por medio de archivos, dada la falta de otros recursos. Esta integración puede mejorarse modificando y recompilando el código de las herramientas LD-NET y CIO (herramienta encargada de la E/S en los transputers) agregándoles la capacidad de utilizar PVM eficazmente.

7.3. Algoritmo Global

La integración mencionada anteriormente fue instanciada utilizando un algoritmo global de multiplicación de matrices. El objetivo de este algoritmo global es servir de ejemplo de cómo integrar máquinas heterogéneas resolviendo un problema mediante un algoritmo distribuido.

Se utilizó el algoritmo de Fox [Kum94] [Gei94] que calcula $C = AB$, donde *C*, *A*, y *B* son matrices cuadradas. Por simplicidad se asume que *m* x *m* tareas serán usadas para calcular la solución. Cada tarea calculará un subbloque de la matriz resultante *C*. El tamaño de bloque y el valor de *m* son dados como parámetros al programa. Las matrices *A* y *B* son también almacenados como bloques distribuidos en m^2 tareas.

Se asume que se tiene una grilla de *m*x*m* tareas. Cada tarea (t_{ij} donde $0<=i,j<=m$) inicialmente contiene los bloques A_{ij}, B_{ij} y C_{ij}. En el primer paso del algoritmo las tareas en la diagonal (T_{ij} en donde $i=j$) envían su bloque A_{ii} a todas las tareas de la fila *i*. Después de

la transmisión de A_{ii}, todas las tareas calculan A_{ii} **x** B_{ij} y se lo incrementa a C_{ij}. En el próximo paso, las columnas de bloques de *B* son rotadas. Esto es, t_{ij} envía sus bloques de *B* a $t_{(i-1)j}$ (la tarea t_{0j} envía su bloque *B* a $t_{(m-1)j}$). Las tareas retornan al primer paso; $A_{i(i+1)}$ se envía en forma de multicast a todas las tareas en la fila *i*, y el algoritmo continúa. Después de *m* iteraciones la matriz *C* contiene *A* x *B*, y la matriz *B* rotada regresa a su lugar.

8. Resultados Obtenidos y Conclusión

A continuación presentaremos los tiempos de ejecución obtenidos en distintas pruebas. Estas pruebas fueron realizadas en función de parámetros que establecen los tamaños de matrices máximos para la multiplicación de matrices en el hipercubo, el tamaño de bloque óptimo para nuestra implementación y el tamaño de los buffers requeridos para una correcta ejecución del proceso de multiplicación.

Finalmente, se enuncian las conclusiones obtenidas a partir de las decisiones de diseño e implementación y de los tiempos de ejecución calculados.

8.1. Parámetros de experimentación

Los parámetros considerados son los siguientes : dimensión de las matrices; tamaño de los bloques en que fueron subdivididas las matrices; tamaño de los buffers para contener los mensajes en espera.

8.1.1. Dimensiones de las matrices

Las dimensiones máximas de la matrices A, B, y C a multiplicar en la red de transputers son M=2500, K=2500 y M=2500 elementos de tipo float. Estas matrices ocupan aproximadamente el 60% de la capacidad total (32x4MB) de memoria del hipercubo formado por los 32 transputers. El resto de la memoria es ocupado por los buffers de los routers en cada transputer y por datos auxiliares necesarios para la ejecución del algoritmo.

8.1.2. Tamaño de bloques

Las matrices A,B y C fueron subdivididas en bloques de dimensiones M_b=25, K_b=25 y N_b=25. Los tamaños de estos bloques fueron escogidos teniendo en cuenta que se puedan multiplicar matrices de la mayor dimensión posible. Al agrandarse el tamaño de bloque se aumenta el tamaño de mensaje y por consiguiente el de los buffers. Por lo cual se buscó un tamaño que permita multiplicar matrices grandes, pero que a su vez no sea tan pequeño para que no exista tanta comunicación y sí haya más cómputo. En la relación cómputo/comunicación en los transputers mediante distintas pruebas pudimos comprobar que el tiempo de comunicación es aproximadamente diez veces menor al tiempo de cálculo. Por lo que el tiempo de comunicación no influirá en el rendimiento del algoritmo, ya que la comunicación y cálculo se producirán simultáneamente.

8.1.3. Tamaño de buffers

Los buffers fueron ajustados para el caso máximo de matrices de dimensión 2500x2500. El buffer vertical de cada transputer puede contener dos mensajes, en tanto que el horizontal tiene capacidad de hasta 6 mensajes. Nótese que los buffers horizontales son

mayores debido a la falta de conexión entre el último transputer de una fila y el primero, por lo cual se cambió el esquema de comunicación como se describe en la sección 5.

8.2. Resultados obtenidos

A partir de los parámetros especificados anteriormente, se realizaron distintas ejecuciones del algoritmo de DIMMA y global. Las dimensiones de las matrices son: 300x300, 500x500, 800x800 y 1000x1000. En el caso del algoritmo de DIMMA también se realizó una prueba con una dimensión de 2500x2500 que es la máxima dimensión posible en la red de transputers.

8.2.1. Tiempos en los transputers

La cantidad de MFLOPS (Millones de operaciones de punto flotante por segundo) para cualquier algoritmo de multiplicación de matrices se puede calcular con la siguiente fórmula (donde N es la dimensión de las matrices):

$$\text{número de operaciones en pto flotante} = 2N^3 - N^2$$

La ejecución de la multiplicación de matrices en forma secuencial, con dimensiones de 500x500, consumió 678,75 segundos. La cantidad de MFLOPS para un transputer calculada con estos valores es 0.3679558.

En tanto que la ejecución de la multiplicación de matrices en forma paralela en los 32 transputers, con dimensiones de 2500x2500, consumió 2859,04 segundos. Y la cantidad de MFLOPS para un transputer calculada con estos valores es 10.9280563. Dividiendo por 32 obtenemos una cantidad de 0.34150176 MFLOPS. La proximidad entre este valor y la cantidad de MFLOPS obtenida en un transputer demuestra que se no se perdieron MFLOPS de la multiplicación paralela, por lo cual durante todo el trabajo de multiplicación se distribuye homogéneamente entre los procesadores.

Si calculamos el speed-up, esto es, dividimos los MFLOPS de la multiplicación en los 32 transputers por la cantidad de MFLOPS en la multiplicación secuencial; obtenemos un valor de 29.6993721. Este valor nos indica que tenemos una utilización completa de un poco más de 29 procesadores, lo cual es 92,5% de la máquina paralela.

El porcentaje de utilización de la máquina paralela obtenido es satisfactorio. Este resultado es como consecuencia de dos factores: por un lado el algoritmo DIMMA brinda un solapamiento de comunicación y cálculo mejorando el rendimiento global; y además la implementación de éste algoritmo en los transputers demostró una correcta utilización de los mismos teniendo en cuenta las restricciones presentadas al no tener una malla cuadrada de procesadores y tampoco contar con las conexiones requeridas para formar un anillo horizontal y vertical.

8.2.2. Tiempos totales

El tiempo total de la caja negra conformada por la integración de PVM con los trasnputers consiste en el tiempo de comunicación (por medio de PVM) de las matrices *A*, *B* y el resultado *C*; más el tiempo empleado por los transputers para la multiplicación; más el tiempo de la integración propiamente dicha. El tiempo consumido en los transputers, a su vez se subdivide en el tiempos de distribución, el tiempo de cálculo y el tiempo de la obtención del resultado. El tiempo de distribución es el tiempo necesario para que los transputers reciban las matrices a multiplicar y la distribuyan hacia cada nodo de la red de transputers. El tiempo de cálculo es el tiempo de ejecución del algoritmo de DIMMA. En tanto que el tiempo de obtención del resultado es el tiempo empleado desde el momento que se terminó de multiplicar las matrices hasta que se hayan recolectado los resultados de cada transputer.

Tiempos de comunicación en PVM

El tiempo de comunicación en una red de interconexión se puede calcular de la siguiente manera:

$$T(N) = \alpha + \beta * N$$

donde N es el tamaño del mensaje en bytes; α es el tiempo de latencia; β es la tasa de transferencia y T(N) es el tiempo de comunicación.

Tanto α como β son desconocidos y serán estimados utilizando **Regresión Lineal .** Este método de cálculo se explica en forma detallada en el Apéndice D: Regresión Lineal.

El tiempo de comunicación en PVM (que utiliza finalmente TCP/IP) se ajusta al modelo de regresión lineal. Existen parámetros de hardware para α y β, pero no se puede confiar en ellos, porque un mensaje al ser enviado o recibido debe pasar por las distintas capas del protocolo TCP/IP, más una capa adicional propia de PVM. Estas capas introducen una sobrecarga que hace que estos parámetros varíen de los originales.

Se tomó una muestra de $n = 11575$, de las cuales 10000 observaciones se efectuaron con tamaño de mensajes cero. Esta cantidad de observaciones con tamaño cero se debe en la necesidad de obtener una buena estimación de α (más adelante esto es justificado con el intervalo de confianza de α, donde se puede observar que su longitud es muy pequeña). Las restantes 1575 observaciones se efectuaron con tamaño de mensaje mayor que cero y menor que 2,5 MB (con tamaños mayores el sistema efectúa swapping de disco, produciendo un considerable aumento en los tiempos de comunicación). Se obtuvieron las siguientes estimaciones para α y β:

$\hat{\alpha}$=0,00352665 segundos.

$\hat{\beta}$=1,7374 x 10^{-6} segundos/bytes.

$\hat{\sigma}^2$=0,00065982.

Se calcularon siguientes intervalos de confianza de nivel 0,975 (α=0,025)

$S(\hat{\alpha})$=[0,0029622 ; 0,0040911]

$S(\hat{\beta})$=[1,736x10^{-6} ; 1,738x10^{-6}]

$S(\hat{\sigma}^2)$=[0,0015047 ; 0,0018391]

La longitud de estos intervalos de confienza es muy pequeña, lo cual nos indica que las estimaciones son buenas. En el caso de α, la longitud es mayor, pero teniendo en cuenta el peor de los casos no influye demasiado en los tiempos estimados.

Tiempos Calculados

Para la obtención de los tiempos de cálculo en la integración se utilizaron matrices con una dimensión máxima de 1000x1000. Esta restricción se debe a la capacidad de memoria de las máquinas con sistema operativo Linux. Los tiempos calculados se muestran en la tabla 9.1 y están expresados en segundos.

	Tiempo de Distribución	Tiempo de Cálculo	Tiempo de Recolección	Tiempo de comuncicación	Tiempo Integración	Tiempo Total
300	2,13	6,77	6,77	1.38	1.36	40
500	11,4	27,13	19,52	5.22	1.86	139
800	31,46	87,82	54,66	13,35	9.62	570
1000	65,19	172,44	93,34	20,35	22.8	974

Tabla 9.1: Tiempos Calculados

El tiempo de comunicación es el tiempo empleado para enviar las matrices A y B desde el proceso Linux hacia los transputers más el tiempo empleado para la comunicación del resultado. Estos tiempos fueron calculados por medio de regresión lineal como se mencionó anteriormente.

El tiempo de integración esta conformado por el tiempo empleado en el pasaje de parámetros mediante archivos, hacia y desde los transputers.

El tiempo total consiste en el tiempo empleado en la ejecución remota del proceso localizado en los transputers, más el tiempo requerido para la comunicación de las matrices, más el tiempo empleado por la caja negra conformada por la integración del hipercubo con PVM. Se puede observar que este tiempo es mayor que lo esperado. Justificándose esto en el hecho de que las estimaciones se realizan para casos ideales, no teniéndose en cuenta factores como la carga de la red, swapping a disco para mensajes de gran tamaño y el tiempo requerido para la integración.

Para obtener los tiempos de ejecución del algoritmo global de multiplicación de matrices se integraron, por medio de PVM, cuatro máquinas: dos con sistema operativo Windows 95 y dos con sistema operativo Linux. Una de las máquinas con sistema operativo Windows 95 es la que está conectada a la red de transputers. Esta máquina la multiplicación de sus bloques correspondientes la realiza por medio del algoritmo DIMMA que se ejecuta en la red de transputers, las otras realizan la multiplicación de los bloques en forma secuencial. En la tabla 9.2 se muestran los tiempos, expresados en segundos, empleado por el algoritmo global. Comparando con los tiempos de cálculo obtenidos con la red de transputers se observa una clara diferencia. Esto se debe a que el algoritmo global empleado no posee solapamiento de calculo y comunicación y además hay que tener en cuenta la heterogeneidad de las máquinas empleadas. En nuestro caso, una de las máquinas integradas (con sistema operativo Linux) presentaba un poder de procesamiento inferior a las demás por lo que retrasaba el procesamiento de las demás máquinas.

	300	500	800	1000
Tiempo Global	108	245	680	1270

Tabla 9.2: Tiempos del algoritmo global

8.3. *Conclusiones*

Como conclusión de este trabajo podemos afirmar que se lograron los objetivos planteados de integrar el hipercubo con una red de estaciones de trabajo e implementar un algoritmo de multiplicación de matrices.

La implementación del algoritmo DIMMA sobre la grilla de 32 transputers arrojó resultados que demuestran que los MFLOPS obtenidos son aproximados a la solución secuencial. Esto indica que aprovechó al máximo la capacidad de los transputers, al no producirse una pérdida significativa en la cantidad de MFLOPS por segundo. El algoritmo DIMMA posibilitó estos resultados gracias a su característica de balanceo de carga (por medio de la distribución cíclica) y el solapamiento de comunicación y cómputo (por el esquema de comunicación pipelined).

Con relación a los tiempos totales, es de notar que la mayor parte del tiempo es empleado por la comunicación e integración. En el tiempo empleado en comunicación, podemos notar grandes diferencias con los tiempos estimados. Como fue aclarado anteriormente esto se debe a factores propios de la red y de las máquinas intervinientes, como por ejemplo la carga y swapping de disco.

Otro punto a tener en cuenta es el límite impuesto por el número de canales de cada transputer. Esto influye en el hecho en que no se puede plasmar directamente las soluciones de distintos problemas sobre el hipercubo. En nuestro caso, con el algoritmo DIMMA, nos vimos imposibilitados de poder definir un anillo sobre cada fila de la grilla por falta de links. A causa de esto se debió modificar el esquema de comunicación original, acarreando como resultado la necesidad de aumentar el tamaño de los buffers debido a la pérdida de sincronización de los procesos.

A pesar de la restricción anterior, los transputers tienen como ventaja un amplio espacio de memoria distribuido. Cada transputer que conforma el hipercubo posee 4MB de memoria, que utilizados en forma conjunta (128MB) permite realizar cálculo distribuido con una carga de datos importante. Aunque actualmente los PC poseen 128MB de memoria RAM, cada transputer pude tener como máximo 4GB por lo que el total de memoria alcanzaría los 128GB en todo el hipercubo.

Uno de los principales problemas que debimos afrontar fue la comunicación de procesos PVM con procesos ejecutándose en los transputers. Como se mencionó anteriormente, la solución óptima era la utilización de un triple pipelining para el pasaje de parámetros desde y hacia los transputers. Desafortunadamente la implementación de pipelining en DOS no cumple con la funcionalidad esperada al no permitir la ejecución concurrente de los procesos intervinientes. Esto nos llevó a realizar una solución ad-hoc mediante archivos, degradando en forma considerable el rendimiento del sistema. Si bien esta solución es bastante primitiva, queda abierta la posibilidad de mejorarla integrando rutinas PVM con las herramientas provistas por los transputers para la carga y ejecución de programas y para la entrada/salida de los procesos.

8.4. Trabajo futuro

Uno de los principales puntos a mejorar es obtener una administración más óptima de la comunicación en la red heterogénea. En nuestro caso, la implementación de PVM para Windows 95 produce gran cantidad de swapping para el envío de mensajes degradando considerablemente el rendimiento global. Esto se observa claramente en la tabla 9.1 en donde el tiempo total es ampliamente superior a la suma de los restantes tiempos.

Otro punto no tenido en cuenta durante el desarrollo del trabajo es el balance de carga en una red heterogénea. Mediante un diseño cuidadoso en la asignación de la carga de trabajo a cada estación de la red heterogénea, se pude nivelar la carga de trabajo considerando la capacidad de procesamiento de cada una de modo tal que una estación no retrase el procesamiento global. En nuestro caso, nos vimos limitados por una computadora con un procesador de bajo rendimiento comparado con las demás.

Como se mencionó anteriormente, la interface de comunicación entre Windows 95 y los transputers se podría modificar (ya que existen los fuentes) de manera que los programas para los transputers soporten sentencias de PVM. De este modo la comunicación entre PVM y los transputers sería más directa y rápida.

APENDICE A: LENGUAJE 'C' PARALELO

El lenguaje es en realidad un compilador 'C' ANSI más las librerías escritas por Logical Systems [Log94a][Log94b] para posibilitar la definición de procesos que se ejecuten en paralelo y que se comunican entre sí. El paquete contiene un preprocesador standard, compilador, assembler y linker, más las utilidades loader y host driver, requeridas por el ambiente paralelo.

El proceso de compilación y ejecución requiere de cinco pasos: 1) preprocesor, 2) compiler, 3) assembler, 4) linker y 5) loader.

A continuación se hará una breve descripción de cada uno de los pasos necesarios para la compilación, indicando para cada paso los parámetros más relevantes y usados, y además la utilización del programa TCC que automatiza los pasos anteriores. Luego se describirán las utilidades loader y host driver necesarias para la ejecución de programas paralelos. Al final del apéndice se describirán las primitivas más utilizadas para la programación de concurrencia y paralelismo.

Proceso de compilación y linking

En la figura A.1 se muestra los comandos necesarios para ejecutar cada paso.

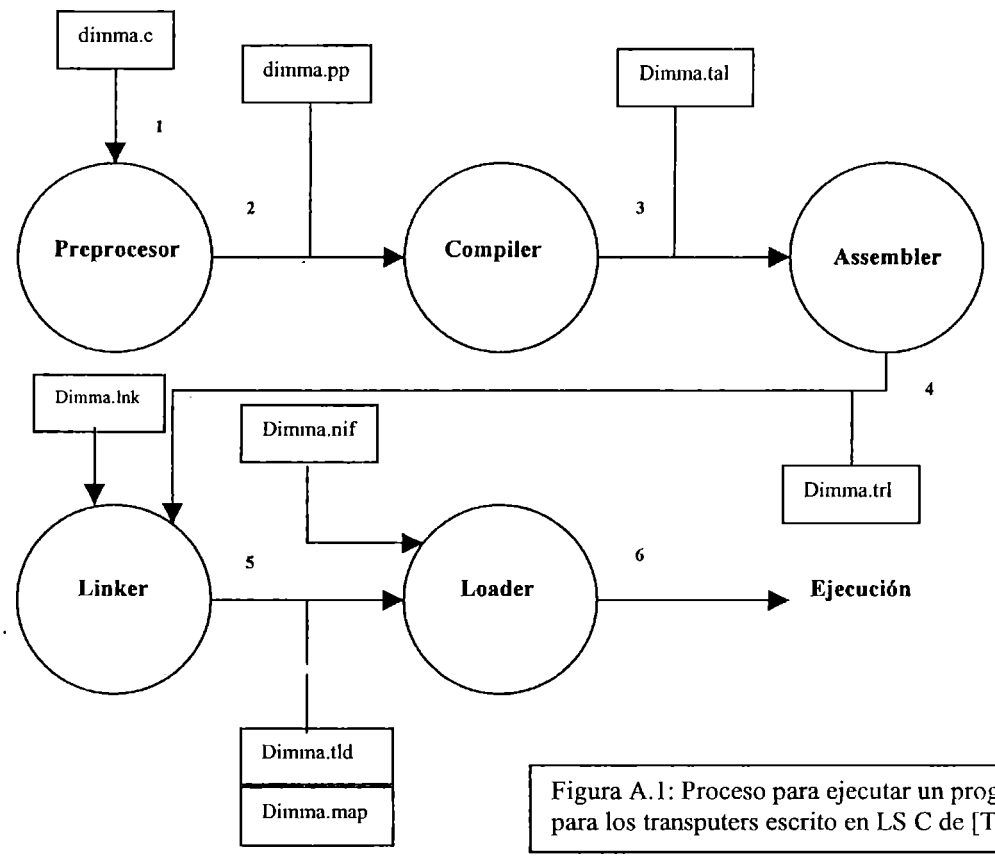


Figura A.1: Proceso para ejecutar un programa para los transputers escrito en LS C de [Thie95]

Preprocessor

El compilador es una combinación de un preprocesador, PP, y un cross-compiler, TCX. El preprocesador PP lee el archivo que contiene el programa C original y genera un nuevo archivo, conteniendo cualquier inclusión de archivo, extensiones de macros, y condicionales de compilación que se encuentren en el código original. La sintaxis para invocar al preprocesador es:

PP <nombre archivo de entrada> [-opciones]

donde [-opciones] representa distintas alternativas del preprocesador. Este paso genera un archivo con el mismo nombre que el original pero con extensión .pp.

Compiling

Luego el archivo con extensión ".pp" es pasado al compilador para su procesamiento.

TCX <nombre archivo de entrada> [-opciones]

De la misma manera [-opciones] representa las distintas alternativas para controlar la compilación. Una de las opciones más utilizadas es –c, la cuál fuerza al compilador a comprimir el archivo de salida eliminando cualquier información de debugging. La opción –f? está asociada con la representación de los números double y float. A menos que se indique otra cosa, el compilador devuelve un archivo con el mismo nombre que el archivo de entrada, pero con la extensión ".tal". Este archivo contiene código assembler del transputer correspondiente al programa original.

Assembling

Tasm es el assembler disponible por Logical Systems. Puede ser utilizado sólo como assembler para procesar programas de usuario, o adicionalmente al compilador tcx para generar código de transputer. La sintaxis del assembler tasm es:

TASM <nombre archivo de entrada> [<dir. temporal>] [-opciones]

donde [-opciones] controla como opera el assembler. El archivo de salida del tasm tiene la extensión ".trl" (**t**ransputer **r**elocatable).

Linking

El proceso de linking es realizado por el programa tlnk. Su función es tomar el código objeto ubicable creado por el assembler y linkearlo con los archivos de librería, o módulos conteniendo el código de la funciones de librería invocadas por el programa del usuario. La salida del tlnk es un archivo listo para ser descargado en el transputer. La extensión del archivo es ".tld" (por **t**ransputer down**l**oa**d**). Entre la información requerida

por el linker se encuentra *load* y *stack*, las cuales deben ser utilizadas en conjunto para evitar tener resultados inesperados.

- **LOAD <número decimal, octal o hexadecimal>** el número deber ser compatible con el formato de C y representa la dirección inicial de carga del programa, generalmente es la primera dirección de la memoria off-chip

- **STACK <número decimal, octal o hexadecimal>** el número deber ser compatible con el formato de C y representa la dirección de la pila. Esta dirección es la dirección de la palabra más alta de la pila, y su valor es asignado de forma tal que la pila se encuentre on-chip (asignando al puntero de la pila la dirección de la primera palabra de la memoria off-chip). La pila crece hacia abajo. Es útil para aplicaciones que necesitan mayor tamaño de pila que la convencional.

El programa TCC

El programa TCC está diseñado para automatizar el proceso de compilación, assembling y linking de los programas para transputer. En vez de ejecutar PP, TCX, TASM y TLNK individualmente, puede utilizar TCC para ejecutar las herramientas en forma apropiada. La sintaxis del programa TCC es la siguiente:

TCC <nombre archivo de entrada> [-opciones]

Al menos un archivo de entrada debe ser dado. La extensión del archivo de entrada determinará qué tipo de procesamiento realizará el TCC. Cero o más opciones pueden ser especificadas al control de operaciones del TCC.

Tiene como salida dos archivos. Uno con extensión ".trl" el cual es un archivo objeto ubicable producido por el TASM, y uno con ".tld" el cual es un ejecutable linkeado producido por el TLNK.

La opción "-a address" especifica la dirección inicial de carga del programa ejecutable y su contenido es enviado al TLNK para el comando LOAD.

La opción "-e ?" especifica dónde el archivo linkeado será comenzado. El contenido de esta opción es pasado al TLNK como el comando "entrypoint". Por defecto es "_main". Para la utilización de canales virtuales se debe especificar "_vcmain".

La opción "-t [f] tipo" se corresponde con la opción "f?" del TCX y define la representación numérica.

La opción "-w address" especifica la dirección inicial de la pila y su contenido es enviado al TLNK para el comando STACK.

Carga y ejecución

La carga del programa para el transputer en la red de transputers y su envío requiere el programa ld-net (loader) para realizar las siguientes tareas:

- Debe cargar el programa en el transputer root.
- Una vez que el programa paralelo está cargado, debe dar señal al transputer root para comenzar la ejecución.
- Debe comenzar la ejecución de un programa en la computadora host para los requerimientos de entrada y salida del transputer root que pueda necesitar.

La forma general de la línea de comando del ld-net es la siguiente:

LD-NET [-v] [-x] <archivo NIF> <argumentos >

La opción "-v" indica al ld-net si ejecuta en modo "verbal" o "quieto". En el primero muestra en pantalla el progreso del proceso de descarga y en el segundo sólo informa errores. La opción "-x" previene al ld-net de comenzar automáticamente un servidor de E/S y se utiliza para facilidades de debugging. El "archivo NIF" especifica el archivo a leer para determinar la topología de la red a ser cargada. Cualquier parámetro restante es pasado al programa del host server.

Alternativamente se dispone del programa ld-one que permite cargar y ejecutar en un único transputer. Carga un único programa con extensión ".tld" en el transputer root y la línea de comando es la siguiente:

LD-ONE [-v] [-x] <archivo NIF> <argumentos >

Es similar al ld-net pero con la carga y ejecución de un sólo programa.

Archivo de información de la red

El loader obtiene la información que indica la manera en que debe realizar los pasos anteriores de un archivo, el *archivo de información de la red* (*network information file*, NIF). Consiste de dos partes. La primera sección contiene cuatro líneas para el control del proceso de carga. La segunda sección contiene información que describe el grafo que representa la red de transputers.

Comandos

El comando **Buffer_size** define el número de bytes de la memoria interna del transputer cuando los programas son descargados. Un buffer es usado para almacenar los datos transferidos.

El comando **Host_server** identifica el programa que se ejecutará en la máquina host. El programa disponible en el CSA es el cio.exe, pero el usuario puede crear el propio para adaptarlo a sus necesidades.

El comando **Level_timeout** y **decode_timeout** son utilizados para inicializar la red de transputers. A partir de ambos comandos se puede detectar cuando un transputer falla en la carga de la red. Ver el apéndice "Configuración del hipercubo como una grilla".

Descripción del nodo

La segunda parte del NIF describe la configuración física de la red de transputers, incluyendo los link de vecino a vecino, y la alocación de programas a transputers. El formato de cada línea es el siguiente:

Node#, Program, Parent, [Link0], [Link1], [Link2], [Link3];

El campo *Node* indica el número de transputer en el que será cargado el programa, el transputer con número 1 es el transputer root. El campo *Program* es el programa que será cargado en el transputer y el campo *Parent* indica el transputer que inicializará el proceso de inicialización para formar el árbol de transputers. En los siguientes campos se indica el link y número de transputer con el que está conectado el transputer actual. SE ENTIENDE?

Las definiciones de canales virtuales tienen el siguiente formato general (todos los campos son numéricos y usan las convenciones de las constantes del lenguaje 'C'):

frm_Node [frm_Chan], to_Node [to_Chan], frm_Load, to_Load;

El campo *frm_Node* define el nodo origen y el campo *frm_Chan* define el canal lógico en el que el nodo realizará los envíos. Los campos *to_Node* y *to_Chan* definen lo mismo pero para la parte receptora. Los dos últimos campos son opcionales y especifican el "canal de carga" para la dirección enviante (*frm_Load*), y la dirección reversa (*to_Load*).

Primitivas para implementar concurrencia

El transputer tiene un importante conjunto de instrucciones para implementar sistemas concurrentes. El conjunto incluye instrucciones para comienzo, finalización y pasaje de mensajes entre procesos. El hardware incluye características para bloquear y reiniciar los procesos en espera de una comunicación y seleccionar un nuevo proceso después de que ha transcurrido un período de tiempo, ubicando a este proceso al final de la cola de procesos activos.

La librería conc.h (transp toolset reference y library) tiene funciones y variables que implementan un modelo de concurrencia similar al de Occam. A continuación se describen brevemente las rutinas más utilizadas de esta librería. Para más información de las rutinas consultar.

Concurrencia

Antes que un nuevo proceso sea ejecutado, se debe reservar espacio para su pila. Una vez reservado este espacio, la pila debe ser inicializada para la correcta ejecución del proceso. Un nuevo proceso es alocado de la siguiente manera:

```
Process *ProcAlloc (func, sp, nparam, p1, p2, ..., pn)
    int (*func)();
    int sp;
    int nparam;
```

La rutina *ProcAlloc()* toma un puntero *func* a una función que contiene el código para el proceso. El parámetro *sp* indica la cantidad de espacio de pila requerida por el proceso, un valor de cero para éste parámetro hace que la rutina reserve un espacio de *DEFAULTWSSIZE* bytes (64K) de espacio para la pila. El parámetro *nparam* especifica el número de palabras de ocupadas por los parámetros de la función.

Retorna un puntero a una estructura *Process* que constituye el proceso. En el momento de la ejecución de la función se le pasan los parámetros (process_ptr,p1, p2,..., pn). En donde process_ptr es el puntero a la estructura retornada por el llamado a ProcAlloc.

Las siguientes son rutinas para la ejecución de procesos:

```
ProcRun (Process *p)
ProcRunHigh (Process *p)
ProcRunLow (Process *p)
ProcPar (Process *p1, p2, p3,...., pn, 0)
ProcParList (Process **plist)
```

Las rutinas *ProcRun()*; *ProcRunHigh()*, y *ProcRunLow()* ejecutan los procesos en forma asíncrona. El proceso comienza la ejecución y está fuera del control del proceso inicializador. El proceso inicializador no tiene medios para determinar o alterar el estado del proceso creado excepto a través de un medio de comunicación que el usuario establezca explícitamente. La rutina *ProcRun()* ejecuta al proceso en la misma prioridad que el proceso actual, *ProcRunHigh()* ejecuta al proceso en alta prioridad y *ProcRunLow()* ejecuta el proceso en baja prioridad.

Las rutinas *ProcPar()*, *ProcParList()* comienzan un grupo de procesos. El control es retornado al proceso inicializador cuando todos los procesos iniciados terminan. *ProcPar()* toma un null explícito como fin de la lista de procesos, todos los procesos son ejecutados en la prioridad actual. *ProcParList()* toma un array terminado en null de punteros a procesos, todos los procesos son ejecutados en la prioridad actual.

Comunicación entre procesos

El transputer proporciona un protocolo de pasaje de mensajes para la comunicación entre procesos. Un canal es un flujo (stream) de mensajes unidireccional entre dos procesos. Cuando un proceso realiza una entrada o salida a un canal, el proceso es bloqueado hasta que el proceso correspondiente realiza su respectiva entrada o salida. De esta forma, los canales pueden ser usados como un mecanismo de sincronización adicionalmente al mecanismo de comunicación. Los procesos intervinientes deben tener la precaución de realizar operaciones sobre los canales con el mismo tamaño de datos para evitar resultados inesperados.

```
ChanOut (Channel *c, char *cp, int cnt)
ChanOutChar (Channel *c, char ch)
ChanOutInt (Channel *c, int n)
ChanIn (Channel *c, char *cp, int cnt)
int ChanInInt (Channel *c)
char ChanInChar (Channel *c)
```

La rutina *ChanOut()* escribe *cnt* bytes de datos en el canal referenciado por *c*, desde el buffer referenciado por *cp*. Las rutinas *ChanOutChar* y *ChanOutInt* pueden ser utilizadas para escribir un valor de un byte o word, respectivamente, al canal referenciado por *c*.

La rutina *ChanIn()* lee *cnt* bytes de datos, desde el canal referenciado por *c*, al buffer referenciado por *cp*. Las rutinas *ChanInChar* y *ChanInInt* pueden ser utilizadas para leer, y retornar, el valor de un byte o palabra, respectivamente, leído del canal referenciado por *c*.

Los canales requieren ser inicializados antes de que sean usados para la comunicación. La rutina *ChanReset()* inicia un canal, retornando información contenida en el canal. La rutina *ChanAlloc()* retorna un puntero a un canal inicializado.

```
int ChanReset (Channel *c)
Channel *ChanAlloc ()
```

El concepto de canal va más allá de los límites de un único transputer. Los cuatro links seriales de un transputer T800 corresponden a ocho punteros a canales (cuatro de entrada y cuatro de salida) con direcciones de hardware específicas. Estas direcciones están contenidas en la librería conc.h:

```
#define LINK0OUT        ((Channel *) 0x80000000)
#define LINK1OUT        ((Channel *) 0x80000004)
#define LINK2OUT        ((Channel *) 0x80000008)
#define LINK3OUT        ((Channel *) 0x8000000c)
#define LINK0IN         ((Channel *) 0x80000010)
#define LINK1IN         ((Channel *) 0x80000014)
#define LINK2IN         ((Channel *) 0x80000018)
#define LINK3IN         ((Channel *) 0x8000001c)
```

Un punto a tener en cuenta en la comunicación es que no existe buffering ni packaging de la información enviada. El mensaje es enviado como es, desde el primer byte hasta el último. Como consecuencia, el proceso receptor debe saber de ante mano la cantidad de bytes a esperar.

Alternación

Las siguientes rutinas permiten determinar el estado de los canales y posiblemente esperar hasta que un canal este listo para leer:

```
int ProcAlt (Channel *c1, *c2, . . . , cn, 0)
int ProcAltList (Channel **clist)
int ProcSkipAlt (Channel *c1, *c2, . . . , cn, 0)
int ProcSkipAltList (Channel **clist)
int ProcTimerAlt (int time, Channel *c1, *c2, . . . ,cn, 0)
int ProcTimerAltList (int time, Channel **clist)
```

Las rutinas *ProcAlt()* y *ProcAltList()* bloquean al proceso hasta que uno de los canales de su lista de argumentos este listo para leer. Luego, retorna un índice de la lista de parámetros que indica el canal listo para la entrada.

Las rutinas *ProcSkipAlt()* y *ProcSkipAltList()* verifican los canales especificados. Si uno de los canales esta listo para leer, retorna un índice en la lista de argumentos, sino retorna –1. Estas rutinas no bloquean esperando por uno de los canales, retornan inmediatamente. El resultado es no determinístico.

Las rutinas *ProcTimerAlt()* y *ProcTimerAltList()* bloquea el proceso actual hasta que uno de los canales esta listo para leer o expira el tiempo especificado. Si la rutina expira, retorna –1, sino retorna un índice en la lista de argumentos.

Las rutinas *ProcAlt()*, *ProcSkipAlt()* y *ProcTimerAlt()* toman un null explícito como fin de la lista de canales. Las rutinas *ProcAltList()*, *ProcSkipAltList()*, y *ProcTimerAltList()* toman un NULL explícito como fin del array de canales.

Semáforos

Los transputers no proveen en forma explícita semáforos pero pueden ser implementados eficientemente usando las instrucciones para concurrencia del transputer. Un semáforo puede ser creado de dos formas:

```
Semáforo      sem = SEMAPHOREINIT;

Semaphore     *sem;
sem = SemAlloc ();
```

Un semáforo es adquirido con la siguiente rutina:

SemP (Semaphore sem);

Un semáforo es liberado con la siguiente rutina:

SemV (Semaphore sem);

La rutina *SemP()* bloquea el proceso actual y lo pone en una cola si el semáforo está en uso, sino setea al semáforo como adquirido y continúa la ejecución. La rutina no retornará hasta que el proceso adquiera el semáforo. La rutina *SemV()* libera el semáforo y ejecuta el primer proceso en la cola si existe alguno en espera.

Misceláneas

El valor del reloj puede ser obtenido con la función *Time()*. Esta función es una función atómica. El reloj es diferente para los procesos de baja y alta prioridad. El reloj de baja prioridad es incrementado cada 64 uS, el reloj de alta prioridad es incrementado cada 1 uS. La ejecución de un proceso puede ser bloqueada hasta un tiempo especificado con la rutina *ProcAfter()*. La ejecución puede ser suspendida por un número especificado de períodos de reloj usando *ProcWait()*. Si es necesario que un proceso realice una "espera ocupada" en un recurso, el proceso puede ser puesto al final de la cola de procesos activos con la rutina *ProcReschedule()*. Un proceso puede determinar su prioridad con la rutina *ProcGetPriority()*. Esta rutina retorna 1 para un proceso de baja prioridad y 0 para un proceso de alta prioridad. Un proceso puede ser detenido con la rutina *ProcStop()*. Bajo circunstancias normales, el proceso no podrá volver a ejecutar luego de invocar a *ProcStop()*. Estas funciones misceláneas son las siguientes:

```
int Time()
ProcWait (int time)
ProcAfter (int time)
ProcReschedule ()
int ProcGetPriority ()
ProcStop ()
```

Administración de la pila y heap

La ejecución de procesos fuera de la pila crea un programa con un comportamiento impredecible que a veces puede ser muy difícil de diagnosticar. En un ambiente paralelo, el overflow de una pila es sólo una de las numerosas posibles razones que un programa pueda tener un comportamiento impredecible, y una administración cuidadosa de la pila es de vital importancia. Desafortunadamente, el compilador LSC no dispone de una opción para verificar el overflow de la pila, y el transputer no tiene ningún soporte de hardware para verificar la pila. El transputer mantiene sólo un registro, el *registro de workspace*, para definir el área de la pila. La responsabilidad de reservar suficiente espacio para la pila es dejada al programador.

El área de la pila para una función puede ser normalmente alocada en una de las dos áreas, dependiendo si la función es el resultado de un llamado anidado desde la función

principal, o si el resultado de un llamado anidado de un proceso paralelo. Nos referiremos a la primera área de *pila del sistema*, y el a la segunda como área de *pila del usuario*.

La pila del sistema

La pila del sistema es definida en el proceso de linking. La forma normal de proveer esta información al linker es por medio de la entrada STACK y LOAD. Se debe tener en cuenta que el tamaño del área de la pila por defecto debe ser lo suficientemente grande para almacenar las variables locales declaradas en las funciones llamadas por el main, y también lo suficientemente grande para ejecutar las funciones de las librerías.

La pila del usuario

La pila del usuario es ubicada en el almacenamiento heap. Es creada por el llamado a la rutina ProcAlloc. La pila del sistema trabaja de la misma forma que la pila del usuario lo hace, excepto que la pila del usuario es privada a cada proceso paralelo. El tamaño de cada uno debe ser lo suficiente grande para contener los frames de todas las funciones llamadas directa o indirectamente por el proceso. La desventaja de tener varias tareas paralelas es que el espacio de pila requerido no puede ser compartido, pudiendo consumir grandes cantidades de memoria a pesar de que sean tareas idénticas las que se estén ejecutando.

Haciendo espacio para las pilas

Así como el número de procesos paralelos en ejecución de un transputer crece, también la memoria del usuario para sus workspaces. Debemos garantizar que la cantidad de memoria dinámica disponible es lo suficientemente grande como para almacenarlos. En principio esta cantidad es de 128 Kbytes. Para incrementar el límite más allá de este valor, la variable externa _heapend definida en la librería estándar debe ser modificada. Esta variable contiene la dirección más alta disponible.

Canales Virtuales

Los canales virtuales pueden ser usados de la misma manera que los canales de soft o de hard. Los canales virtuales pueden conectar procesos residiendo en el mismo transputer, o en diferentes transputers. Pero la principal diferencia, y es donde se destacan, es que pueden conectar transputers que tal vez no sean vecinos. Los canales virtuales son canales full duplex, que permiten la transmisión de información en ambas direcciones, al mismo tiempo.

La principal característica de los canales virtuales es que son independientes de la topología descrita por los canales de hard y pueden conectar transputers que no están directamente conectados. Más aun, un transputer puede tener un número ilimitado (para todos los propósitos prácticos) de canales virtuales. El espacio de memoria disponible es la mayor limitación.

Se requieren de tres acciones para usar un canal virtual: su *declaración*, *inicialización*, y *uso*. La declaración de un canal virtual es realizada de la misma manera que la declaración de un canal de soft. Luego en la inicialización se le asigna un nombre que debe ser único en el mismo transputer e independiente de los demás transputers. La conección de los canales virtuales se define en el archivo de información de la red (NIF).

En general las primitivas de comunicación para los canales virtuales son similares que las utilizadas para los canales de soft o de hard. La principal diferencia se establece en que los nombres de las primitivas, asociadas con los canales virtuales, son antecedidas por la letra "V".

Al utilizar canales virtuales no es posible utilizar canales de hard directamente. Esto se debe a que los canales virtuales usan los links de hard en forma de multiplexor, con routers que controlan directamente el tráfico.

El router virtual agrega un nivel extra de software que es involucrado con cada bit de información que es intercambiado entre dos procesos (remoto o no) en un canal virtual. Por lo que es de esperar pérdida de rendimiento, ya que el router virtual debe utilizar ciclos del procesador para su propio uso. Más aun, cuando un mensaje es enviado en un canal virtual, es empaquetado en unidades que pueden ser mucho mas pequeñas que el mensaje original, teniendo overhead adicional.

APENDICE B: LIBRERIA DE PVM

A continuación se describirán brevemente las rutinas que se encuentran en la librería PVM3 [Gei94]. Las rutinas están agrupadas según su función. Sólo se consideraron las funciones para el lenguaje C.

El modelo de comunicación de PVM asume que cualquier tarea puede enviar un mensaje a cualquier otra tarea PVM y no hay limitaciones en el tamaño o número de mensajes a enviar. Se asume también que existe suficiente memoria para enviar un mensaje, es decir que el espacio de buffer está limitado solo por la memoria física (los buffers son creados dinámicamente). El modelo de comunicación de PVM provee envío de mensajes asincrónico no bloqueante, recepción asincrónica bloqueante y funciones de recepción de mensajes no bloqueantes. Que el envío sea no bloqueante significa que retorna tan pronto como el buffer es libre para ser reusado, y el envío es asincrónico ya que no requiere que el receptor lea el mensaje para poder retornar.

La recepción no bloqueante retorna inmediatamente con los datos o con un flag indicando que no llegaron los datos, mientras que la recepción bloqueante retorna solo cuando los datos se encuentran en el buffer. Además de estas funciones de comunicación, el modelo soporta envío de mensajes de tipo multicast hacia un conjunto de tareas y broadcast hacia un grupo de tareas definido por el usuario.

El modelo de comunicación de PVM garantiza que se preserva el orden en que se envían los mensajes. Los buffers de los mensajes son creados dinámicamente, por lo que, el tamaño máximo de mensaje que puede ser enviado o recibido está sólo limitado por la cantidad de memoria disponible.

Control de Procesos

```
int tid = pvm_mytid( void )
```

La rutina pvm_mytid() retorna el identificador `TID` del proceso que llamó y puede ser llamada varias veces. Esta rutina registra el proceso, que la invoca, en PVM si es la primera llamada a PVM. Cualquier llamada al sistema PVM registrará la tarea en PVM si esta tarea no había sido registrada anteriormente, pero es una práctica común llamar a pvm_mytid() al principio del programa para registrarse.

```
int info = pvm_exit( void )
```

La rutina pvm_exit() le indica al pvmd local que el proceso está saliendo de PVM. Esto no significa que se termina el proceso. Típicamente, los usuarios llaman pvm_exit antes de terminar su programa C.

```
int numt = pvm_spawn(char *task, char **argv, int flag,
                     char *where, int ntask, int *tids )
```

La rutina pvm_spawn() inicia `ntask` copias de un archivo ejecutable `task` en la máquina virtual. `argv` es un puntero a un arreglo de argumentos (terminado en NULL) que serán pasados a cada tarea `task`. El argumento `flag` se utiliza para especificar opciones, y es una suma de las siguientes constantes:

Valor	Opción	Significado
0	PvmTaskDefault	PVM escoge donde iniciar (spawn) los procesos.
1	PvmTaskHost	se inicia en el host especificado por el argumento where.
2	PvmTaskArch	se inicia en una arqutectura (PVM_ARCH) especificada por el argumento where.
4	PvmTaskDebug	comienza tareas bajo un debugger.
8	PvmTaskTrace	se generan datos de trace.
16	PvmMppFront	comienza tareas en un front-end MPP.
32	PvmHostCompl	Complementa el conjunto host en argumento where.

Estas constantes están predefinidas en **pvm3/include/pvm3.h**..

El valor de retorno de pvm_spawn() es `numt`, que es un entero que indica el número de tareas satisfactoriamente iniciadas (spawned) o un código de error indicando si no se pudieron iniciar las tareas. Si las tareas fueron iniciadas, pvm_spawn() retorna un vector `tids` con los TIDs de las tareas creadas; y si una tarea no pudo iniciarse, los códigos de error correspondientes son situados el las posiciones *ntask - numt* del vector.

```
int info = pvm_kill( int tid )
```

La rutina pvm_kill() finaliza una tarea de PVM identificada por `tid`. Esta rutina no está diseñada para terminar la tarea que realiza el llamado. Esto puede hacerse mediante un llamado a pvm_exit() seguido de un exit().

```
int info = pvm_catchout( FILE *ff )
```

Por default, PVM escribe las salidas *stderr* y *stdout* de las tareas disparadas (spawned) al archivo `/tmp/pvml.<uid>`. La rutina `pvm_catchout` hace que la tarea que produce el llamado capture la salida de las tareas que serán disparadas. Los caracteres impresos en *stdout* o *stderr* en las tareas hijas son recogidos por los pvmds y son enviados, por medio de mensajes de control, a la tarea padre, la cual rotula cada línea y la agrega al archivo especificado (en C) o en la salida estándar (en Fortran).

Información

```
int tid = pvm_parent( void )
```

La rutina pvm_parent() retorna el `TID` del proceso que creó esta tarea, o el valor PvmNoParent si no fue creada con pvm_spawn().

```
int dtid  = pvm_tidtohost( int tid )
```

La rutina pvm_tidtohost() retorna en `dtid` el TID del daemon que se está ejecutando el mismo host que `TID`. Esta rutina sirve para determinar en qué host se ejecuta una tarea dada. Otras informaciones más generales sobre la máquina virtual se pueden obtener con las siguientes funciones:

```
int info  = pvm_config(    int *nhost, int *narch,
```

```
                                    struct pvmhostinfo **hostp )
```

La rutina pvm_config() retorna información sobre la máquina virtual, incluyendo el número de hosts, `nhost`, y la cantidad de formatos de datos diferentes, `narch`. También retorna `hostp` que es un puntero a un arreglo cuyos elementos son estructuras `pvmhostinfo`. Este arreglo debe ser al menos de tamaño `nhost`. Cuando retorna, cada estructura `pvmhostinfo` contiene el TID del pvmd, el nombre del host, el nombre de la arquitectura, y la velocidad relativa de CPU para ese host.

```
int info  = pvm_tasks(    int which, int *ntask,
                          struct pvmtaskinfo **taskp )
```

La rutina pvm_tasks() retorna información sobre las tareas de PVM que están ejecutándose en la máquina virtual.. El entero `which` especifica de qué tareas se quiere obtener información. Si su valor es cero, significa todas las tareas; un TID de un pvmd significa todas las tareas que se ejecutan en ese host, sino el valor de `which` debe ser el TID de una tarea.

El número de tareas es retornado en `ntask`. El parámetro `taskp` es un puntero a un arreglo (de tamaño `ntask`) de estructuras `pvmtaskinfo`. Cada estructura `pvmtaskinfo` contiene la siguiente información de la tarea: el TID, pvmd TID, el TID de la tarea que la creó, un flag de estado, y el nombre de archivo con el cual fue creada la tarea.

Configuración Dinámica

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos)
int info = pvm_delhosts( char **hosts, int nhost, int *infos)
```

Las rutinas de C agregan o eliminan un conjunto de `hosts` en la máquina virtual. En la versión de C, `info` retornará el número de hosts satisfactoriamente agregados. El argumento `infos` es un arreglo de longitud `nhost` que contiene un código de estado para cada host agregado o eliminado. Esto le permite al usuario chequear cuando uno de los hosts causó un problema y evitar intentar agregar o eliminar todo el conjunto de hosts de nuevo.

Estas rutinas son usadas para configurar la máquina virtual, pero a menudo son utilizadas para aumentar la flexibilidad y tolerancia a fallas de una aplicación de gran dimensión. También permiten a una aplicación aumentar la capacidad de procesamiento (agregando nuevos hosts) si el problema que se está resolviendo requiere mayor cómputo. Otro uso podría ser para aumentar la tolerancia a fallas de la aplicación teniendo que detectar la falla de un host y realizar un reemplazo del mismo.

Configuración y Consulta de Opciones

```
int oldval = pvm_setopt( int what, int val )
int val = pvm_getopt( int what )
```

La rutina `pvm_setopt` es una función de propósito general que permite al usuario configurar y consultar las opciones del sistema PVM. En PVM 3, esta rutina puede ser utilizada para configurar varias opciones, incluyendo impresión automática de mensajes de error, nivel de debugging, y método de ruteo de comunicación para todos los subsecuentes llamados a PVM. Retorna el valor previo de la opción de configuración `oldval`. En la versión PVM 3.3 el parámetro `what` puede tener los siguientes valores:

Opción	valor	significado
PvmRoute	1	política de ruteo
PvmDebugMask	2	máscara para debugging
PvmAutoErr	3	reporte de auto error
PvmOutputTid	4	salida estándar para los hijos
PvmOutputCode	5	salida de msgtag
PvmTraceTid	6	destino de los hijos
PvmTraceCode	7	destino de msgtag
PvmFragSize	8	tamaño de fragmentación de los mensajes
PvmResvTids	9	permiso de mensajes para tags y tids reservadas
PvmSelfOutputTid	10	salida estándar de la tarea que ejecuta la función
PvmSelfOutputCode	11	salida de msgtag
PvmSelfTraceTid	12	destino para mi
PvmSelfTraceCode	13	destino de msgtag

El uso más popular de pvm_setopt es permitir un ruteo directo para la comunicación entre tareas de PVM .

Pasaje de mensajes

El envío de un mensaje se lleva a cabo en 3 pasos. Primero, un buffer de envío debe ser inicializado por medio de una invocación a pvm_initsend() o pvm_mkbuf(). Segundo, el mensaje debe ser empaquetado en este buffer usando cualquier número y combinación de rutinas pvm_pk*(). Tercero, el mensaje completo es enviado a otro proceso con una llamada a la rutina pvm_send() o broadcast con la rutina pvm_mcast().

Un mensaje es recibido invocando a una rutina de recepción bloqueante o no bloqueante y luego debe ser desempaquetado cada ítem empaquetado del buffer receptor. La rutina de recepción puede ser configurada para aceptar *cualquier* mensaje, o cualquier mensaje de un origen específico, o cualquier mensaje con una etiqueta de mensaje específica, o sólo aquellos mensajes con una etiqueta dada de un origen dado.

Buffers de Mensajes

```
int bufid = pvm_initsend( int encoding )
```

Si el usuario está utilizando sólo un buffer de envío (el caso más típico) entonces pvm_initsend() es la única rutina de buffer requerida. Debe ser invocada antes de empaquetar un nuevo mensaje en el buffer. La rutina pvm_initsend limpia el buffer de envío y crea uno nuevo para empaquetar un nuevo mensaje. El esquema de codificación

usado para esta forma de empaquetado es configurado por `encoding`. El identificador del nuevo buffer es retornado en `bufid`.

Las opciones de `encoding` son las siguientes:

PvmDataDefault

- la codificación XDR es usada por defecto porque PVM debe cubrir el caso en que el mensaje sea enviado a una máquina con distinto formato. Si el usuario sabe que el próximo mensaje será enviado sólo a una máquina que entiende el formato nativo, entonces el puede usar la codificación *PvmDataRaw* y ahorrar costos de codificación.

PvmDataRaw

- no se realiza codificación. Los mensajes son enviados en su formato original. Si el proceso receptor no puede leer este formato, retornará un error cuando realice el unpacking.

PvmDataInPlace

- Los datos son dejados en el lugar para ahorrar costos en el empaquetado. El buffer sólo contiene los tamaños y punteros a los ítems que serán enviados. Cuando pvm_send() es invocado, los ítems son copiados directamente fuera de la memoria del usuario. Esta opción decrementa el número de veces que el mensaje es copiado a expensas de requerir que el usuario no modifique los ítems entre que son empaquetados y enviados.

Las siguientes rutinas de buffer para los mensajes son requeridas sólo si el usuario desea manejar múltiples buffers de mensajes dentro de la aplicación. Múltiples buffers de mensajes no son requeridos para la mayoría de pasaje de mensajes entre procesos. En PVM 3 hay un solo buffer de envío *activo* y un solo buffer de recepción *activo* por cada proceso en cualquier momento. El programador puede crear cualquier número de buffers de mensajes y alternar entre ellos para empaquetar y enviar los datos. Las rutinas para empaquetar, envío, recepción, y desempaquetar afectan sólo los buffers *activos.*

```
int bufid = pvm_mkbuf( int encoding )
```

La rutina pvm_mkbuf crea un nuevo buffer de envío vacío y especifica el método de codificación para empaquetar los mensajes. Retorna el identificador del buffer en `bufid`.

```
int info = pvm_freebuf( int bufid )
```

La rutina pvm_freebuf() libera el buffer con identificador `bufid`. Esto debe realizarse antes de que el mensaje sea enviado y no volverá a usarse el buffer.

```
int bufid = pvm_getsbuf( void )
int bufid = pvm_getrbuf( void )
```

La rutina pvm_getsbuf() retorna el identificador del buffer de envío activo. La rutina pvm_getrbuf() retorna el identificador del buffer de recepción activo.

```
int oldbuf  = pvm_setsbuf( int bufid )
int oldbuf  = pvm_setrbuf( int bufid )
```

Estas rutinas activan el buffer de envío (o recepción) `bufid`, guardan el estado del buffer anterior, y retornan el identificador del buffer activo anterior en `oldbuf`. Si el parámetro `bufid` tiene valor 0 en pvm_setsbuf() o pvm_setrbuf(), entonces el buffer actual es guardado y no hay buffer activo. Esta característica puede ser utilizada para guardar el estado actual del buffer y evitar cualquier interferencia con otra tarea.

Es posible reenviar un mensaje sin reempaquetarlo usando las rutinas de buffering de mensajes. Esto se ilustra en el siguiente fragmento:

```
bufid = pvm_recv( src, tag );
oldid = pvm_setsbuf( bufid );
info  = pvm_send( dst, tag );
info  = pvm_freebuf( oldid );
```

Empaquetamiento de Datos

Cada una de las siguientes rutinas de C empaqueta un array de un tipo de datos dado en el buffer de envío activo. Pueden ser invocadas varias veces para empaquetar datos en un mismo mensaje. Un mensaje puede contener varios arrays cada uno con tipos diferentes de datos. Para los datos de tipo estructurados (Struct) deben ser empaquetados por campo. No hay límites en la complejidad de los mensajes empaquetados, pero una aplicación debería desempaquetar los mensajes exactamente como fueron empaquetados. Aunque esto no es estrictamente requerido, es una buena práctica de programación.

```
int info = pvm_pkbyte(   char   *cp, int nitem, int stride )
int info = pvm_pkcplx(   float  *xp, int nitem, int stride )
int info = pvm_pkdcplx (  double *zp, int nitem, int stride )
int info = pvm_pkdouble(  double *dp, int nitem, int stride )
int info = pvm_pkfloat (  float  *fp, int nitem, int stride )
int info = pvm_pkint   (  int    *np, int nitem, int stride )
int info = pvm_pklong  (  long   *np, int nitem, int stride )
int info = pvm_pkshort (  short  *np, int nitem, int stride )
int info = pvm_pkstr   (  char   *cp )

int info = pvm_packf( const char *fmt, ... )
```

Los argumentos para cada una de las rutinas son un puntero al primer ítem a empaquetar, `nitem` es el número de ítems a empaquetar de éste array, y `stride` es el "paso" usado cuando se empaqueta. Un paso de 1 significa que un vector continuo es empaquetado, un paso de 2 significa cada 2 ítems es empaquetado, y así siguiendo. Una excepción es pvm_pkstr() que empaqueta un string de caracteres terminado en NULL por lo que no necesita los argumentos `nitem` or `stride`.

PVM también provee una rutina de empaquetamiento que usa el mismo formato de una expresión de printf para especificar que datos empaquetar y cómo empaquetarlos en el buffer de envío. Todas las variables son enviadas como direcciones si la cantidad y el paso son especificados; sino, las variables son tomadas como valores.

Envío y Recepción de datos

```
int info = pvm_send( int tid, int msgtag )

int info = pvm_mcast( int *tids, int ntask, int msgtag )
```

La rutina pvm_send() etiqueta el mensaje con un identificador entero `msgtag` y lo envía inmediatamente al proceso `TID`.

La rutina pvm_mcast() etiqueta el mensaje con un identificador entero `msgtag` y realiza un envío multicast del mensaje a todas las tareas especificadas en el arrays de enteros `tids` (excepto así mismo). El array `tids` tiene longitud `ntask`.

```
int info = pvm_psend(   int tid, int msgtag,
                        void *vp, int cnt, int type )
```

La rutina pvm_psend() empaqueta y envía un array del tipo de datos especificado a la tarea identificada por `TID`. El argumento `type` puede ser uno de los siguientes:

```
PVM_STR         PVM_FLOAT
PVM_BYTE        PVM_CPLX
PVM_SHORT       PVM_DOUBLE
PVM_INT         PVM_DCPLX
PVM_LONG        PVM_UINT
PVM_USHORT      PVM_ULONG
```

PVM posee varios métodos para la recepción de mensajes en una tarea. No hay funciones que deban coincidir en PVM, por ejemplo, que pvm_psend debe coincidir con una pvm_precv. Cualquiera de las siguientes rutinas puede ser invocada para cualquier mensaje entrante sin importar cómo fue enviado (o haber realizado un multicast).

```
int bufid = pvm_recv( int tid, int msgtag )
```

Esta rutina de recepción bloqueante esperará hasta que un mensaje con la etiqueta `msgtag` arribe de la tarea `TID`. Un valor de –1 en `msgtag` o `TID` coincide con cualquiera (comodín). Coloca el mensaje en un nuevo buffer de recepción activo que es creado. El anterior buffer de recepción activo es limpiado a menos que haya sido guardado con un llamado a pvm_setrbuf().

```
int bufid = pvm_nrecv( int tid, int msgtag )
```

Si el mensaje requerido no ha arribado, entonces la rutina de recepción no bloqueante pvm_nrecv() retorna `bufid` = 0. Esta rutina puede ser invocada varias veces por el mismo proceso para verificar su arribo, mientras se realiza otro trabajo entre los llamados. Cuando no se requiere realizar más trabajo, la rutina de recepción bloqueante pvm_recv() puede ser invocada para el mismo mensaje. Si el mensaje con etiqueta `msgtag` arriba desde la tarea `TID`, la rutina pvm_nrecv() coloca este mensaje en un nuevo buffer de recepción activo (el cual crea) y retorna el ID de este buffer. El anterior buffer de recepción activo es limpiado a menos que haya sido guardado con un llamado a pvm_setrbuf(). Un valor de –1 en `msgtag` o `TID` coincide con cualquiera (comodín).

```
int bufid = pvm_probe( int tid, int msgtag )
```

Si el mensaje requerido no ha arribado, entonces pvm_probe() retorna `bufid` = 0. Si no, retorna un bufid para el mensaje, pero no lo "recibe". Esta rutina puede ser invocada varias veces para el mismo mensaje para verificar si ha llegado, mientras se realiza otro trabajo entre los llamados. Adicionalmente, pvm_bufinfo() puede ser invocado con el el bufid retornado para obtener información acerca del mensaje antes de recibirlo.

```
int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )
```

PVM también provee una versión timeout de la recepción. Considerar el caso donde un mensaje nunca arribará (por un error o falla); la rutina pvm_recv se bloquearía por siempre. Para evitar tal situación, el usuario puede especificar un tiempo de espera fijo. La rutina pvm_trecv() permite al usuario especificar un período de timeout. Si el período de tiempo es muy largo, entonces pvm_trecv actúa como pvm_recv. Si período timeout es cero, entonces pvm_trecv actúa como pvm_nrecv. Por lo que, pvm_trecv llena la brecha entre las funciones de recepción bloqueantes y no bloqueantes.

```
int info = pvm_bufinfo( int bufid, int *bytes, int *msgtag, int *tid )
```

La rutina pvm_bufinfo() retorna el msgtag, TID origen, y la cantidad en bytes del mensaje identificado por `bufid`. Puede ser usado para determinar la etiqueta u origen de un mensaje que se recibió especificando comodines.

```
int info = pvm_precv(  int tid, int msgtag, void *vp, int cnt,
                       int type, int *rtid, int *rtag, int *rcnt )
```

Una rutina pvm_precv() combina las funciones de una recepción bloqueante y desempaqueta el buffer de recepción. No retorna un `bufid`. En su lugar, retorna los valores actuales de `TID`, `msgtag`, y `cnt`.

```
int (*old)() = pvm_recvf(int (*new)(int buf, int tid, int tag))
```

La rutina pvm_recvf() modifica el contexto de recepción usado por las funciones de recepción y puede ser usada para extender PVM. El contexto de recepción por defecto es que coincida el origen y etiqueta del mensaje. Esto puede ser modificado por cualquier función de comparación definida por el usuario.

Desempaquetamiento de datos

Las siguientes rutinas de C desempaquetan tipos de datos (múltiples) desde un buffer de recepción activo. En una aplicación deberían coincidir con su correspondiente rutina de empaquetamiento en tipo, número de items, y paso. `Nitem` es el número de items de un tipo dado a desempaquetar, y `stride` es el paso.

```
int info = pvm_upkbyte(   char   *cp, int nitem, int stride )
int info = pvm_upkcplx(   float  *xp, int nitem, int stride )
int info = pvm_upkdcplx(  double *zp, int nitem, int stride )
int info = pvm_upkdouble( double *dp, int nitem, int stride )
int info = pvm_upkfloat(  float  *fp, int nitem, int stride )
```

```
int info = pvm_upkint(    int    *np, int nitem, int stride )
int info = pvm_upklong(   long   *np, int nitem, int stride )
int info = pvm_upkshort(  short  *np, int nitem, int stride )
int info = pvm_upkstr(    char   *cp )

int info = pvm_unpackf( const char *fmt, ... )
```

La rutina pvm_unpackf() usa una expresión con formato similar al de printf para especificar que datos desempaquetar y como desempaquetarlos desde el buffer de recepción.

APENDICE C: Configuración del hipercubo como una Grilla

El hipercubo de 32 transputers que se encuentra en la facultad consta de ocho placas ***VME-XP***^{TM} de Alta Technology [ALT94]. Cada una de estas placas pueden contener hasta nueve módulos de transputers (**TRA**nsputer **M**odules: TRAMs) aunque cada una de ellas contiene actualmente sólo cuatro transputers. En el diagrama de la figura C.1 se puede observar la conformación de una placa.

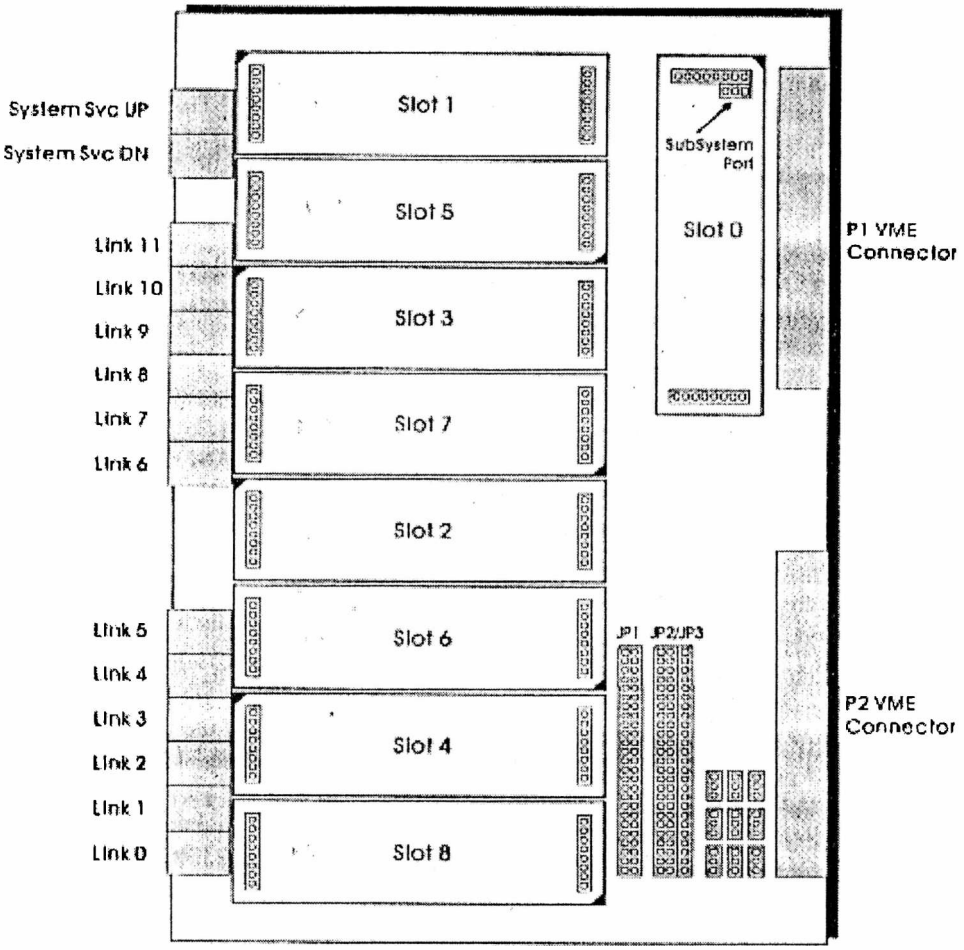


Figura C.1: Placa VME-XP.

Los transputers dentro de la placa se conectan en los slots numerados del cero al nueve. El panel frontal (front panel) consta de catorce conectores externos. Doce de estos conectores se utilizan para acceder a diferentes links correspondientes a los transputers dentro de la placa (Figura C.2). Los dos restantes conectores (UP and DN System Services), junto con los jumpers de System Services, permiten rutear las señales de RESET, ANALYSE y ERROR.

Los conectores del front panel de las distintas placas se pueden conectar mediante cables externos. Para poder operar correctamente, se requieren al menos dos

cables externos conectados con el host. Un cable provee las señales de control (Error, Reset y Analize) y otro que provee una comunicación serial con el host.

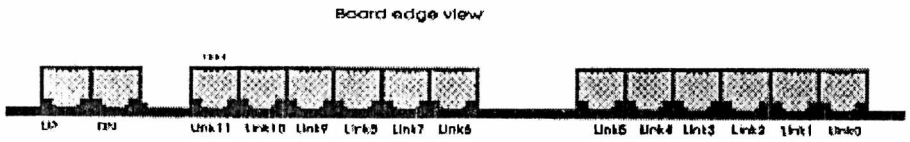


Figura C.2: Conectores externos de una placa.

Los slots numerados del 1 al 4, como también los slots del 5 al 8, se encuentran conectados físicamente en la placa (no se puede cambiar) formando un pipe a través de los links 1 y 2 (Figura C.3). Como se detallará más adelante, mediante jumpers (JP1) es posible conectar estos dos pipes, o realizar un anillo con cada uno de ellos.

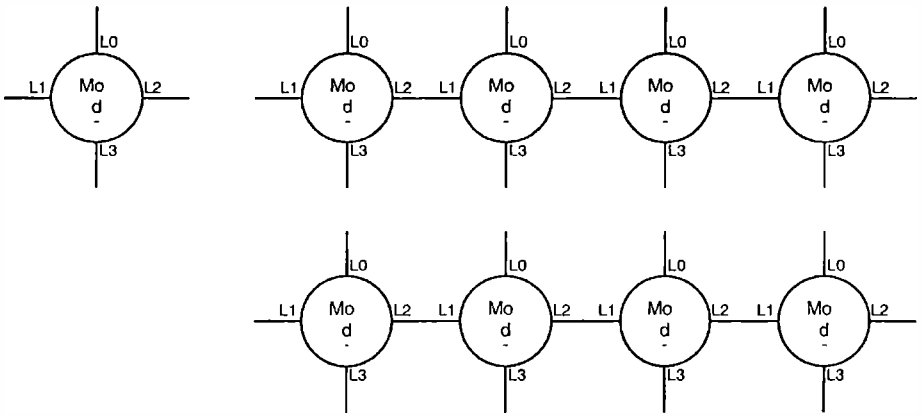


Figura C.3: Conecciones físicas dentro de una placa.

Jumpers de la placa VME-XP

La placa *VME-XP* posee jumpers para la configuración de links y servicios del sistema. Cada link consiste de un par de pines (wires), correspondientes a LINKIN y LINKOUT, orientados de manera que dos conectores verticales conectarán el LINKIN y LINKOUT de un link con el LINKOUT y LINKIN del link adyacente. En la figura 4 se muestra la conexión de dos links adyacentes.

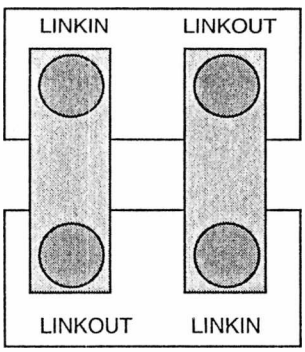


Figura C.4: Conecciones de dos links adyacentes.

Link Configuration Array

El arreglo de configuración de links (hardwired Link Configuration Array) se encuentra etiquetado en la placa como los bloques de jumpers JP1, JP2 y JP3. Este arreglo contiene todos los links de los transputers que se pueden conectar mediante jumpers (figura C.5).

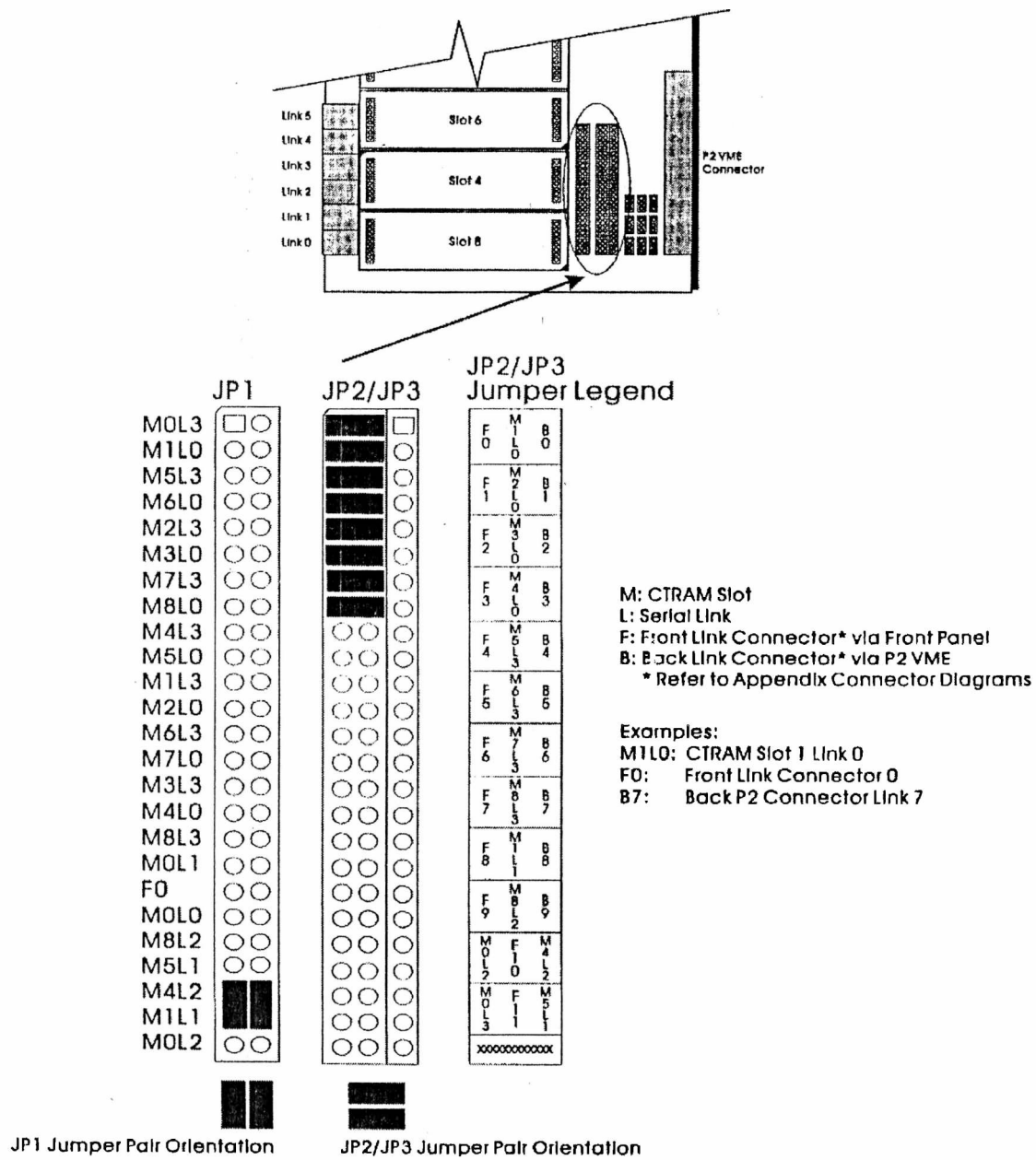


Figura C.5: Configuración original de los arreglos de jumpers en cada placa.

El bloque de jumpers JP1 permiten conectar dos links de diferentes módulos en la placa. Pero estos links deben estar adyacentes en JP1. Por ejemplo, es posible realizar un anillo con los transputers correspondientes a los módulos del 1 al 4, cerrando el pipe físico ya existente mediante la conexión (utilizando dos conectores en forma vertical) de

los pines etiquetados con M4L2 (modulo 4, link 2) y M1L1 (modulo 1, link 1) que se encuentran consecutivos en.

Los bloques de jumpers JP2 y JP3 permiten conectar los links de los módulos con los links del front panel (etiquetados con F0-F11) y con los conectores P2 VM2 (etiquetados como B0-B9), respectivamente. En la figura C.5, al lado de los bloques de jumpers JP2/JP3 se observan las etiquetas (JP2/JP3 Jumper Legend) para las posibles conexiones. Por ejemplo, en esta figura está conectado el link 0 del front panel (F0) con el link 0 del modulo 1 (M1L0). Para hacer esta conexión se utilizaron dos jumpers colocados en forma horizontal.

Configuración Anterior

Como se mencionó anteriormente cada placa contiene sólo cuatro transputers. Estos transputers se encuentran situados en los slots 1 al 4. También como se describió anteriormente, estos módulos están conectados en forma de pipe a través de los links 1 y 2. En la figura 4 se puede observar como se encontraban situados los jumpers en cada placa.

En JP1, sólo se conecta el link 2 del módulo 4 (M4L2) con el link 1 del módulo 1 (M1L1), para formar un anillo con los cuatro transputers. En JP2 los jumpers fueron colocados de manera que el link 0 del módulo 1 (M1L0) se conecte con el link 0 del front pannel (F0), el link 0 del módulo 2 (M2L0) se conecte con el link 1 del front pannel (F1), el link 0 del módulo 3 (M3L0) se conecte con el link 2 del front pannel (F2), y por último que el link 0 del módulo 4 (M4L0) se conecte con el link 3 del front pannel (F3). De esta manera se puede conectar los links 0 de los cuatro módulos de las ocho placas por médio de cables externos colocados sobre los conectores del front panel. En la figura C.6 se muestra un esquema del exterior chasis donde se hallan las ocho placas configuradas como se mencionó anteriormente.

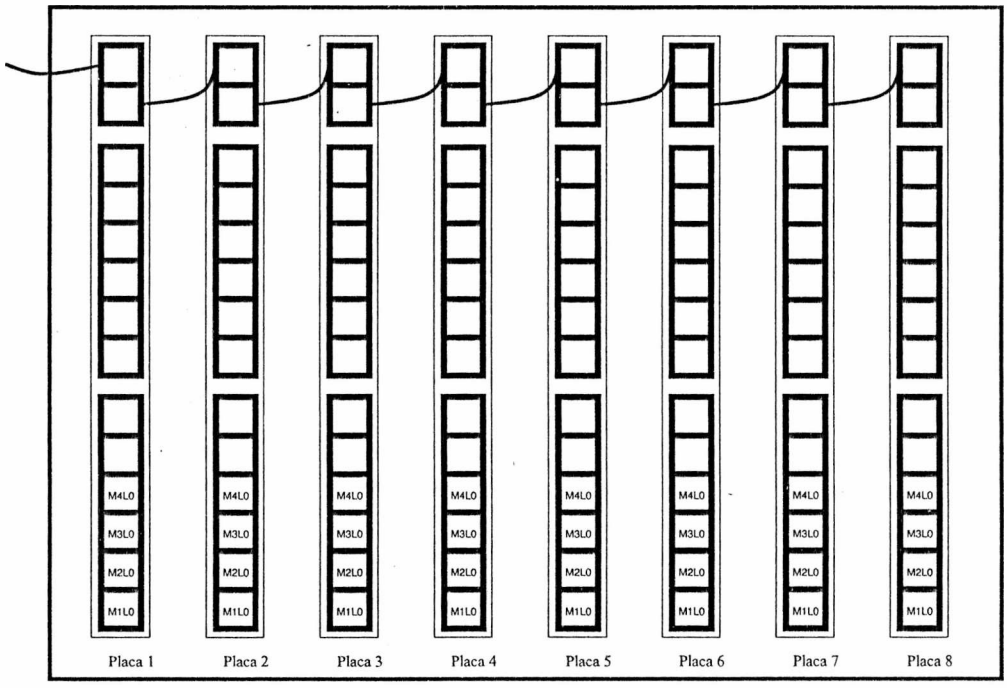


Figura C.6: Placas ubicadas en el chassis del hipercubo

El problema con esta configuración es que no se tenía acceso a los links 3 de cada uno de los módulos. Si se observan las leyendas para los jumpers de JP2/JP3, no hay manera de conectar estos links 3 de cada uno de los módulos con los links del front panel. Sólo es posible conectar los links 0 de los módulos 1 al 4 y los links 3 de los módulos 5 al 8 (los cuales no disponemos).

Nueva configuración para DIMMA

La configuración del hipercubo se modificó para poder formar una grilla de 4x8 transputers como se indica en la figura C.7. Los anillos de cada columna ya se encontraban configurados en cada placa como se mencionó anteriormente. Se buscaba conectar el link 3 de cada módulo con el link 0 del módulo de igual posición en la placa adyacente.

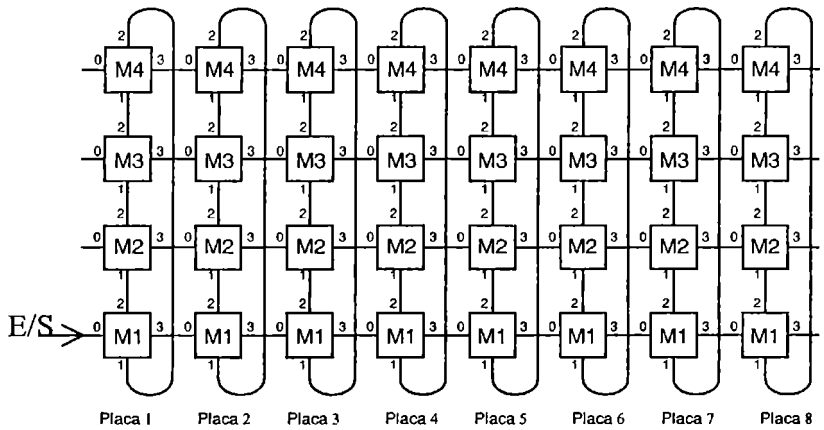
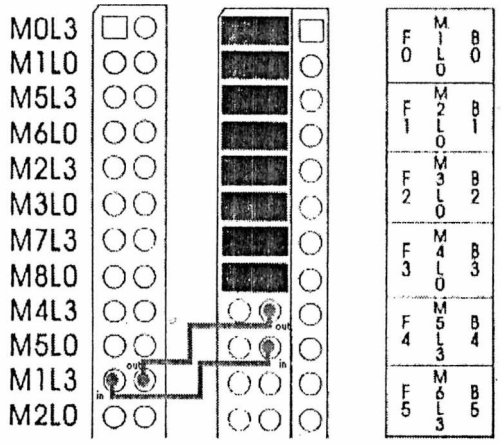
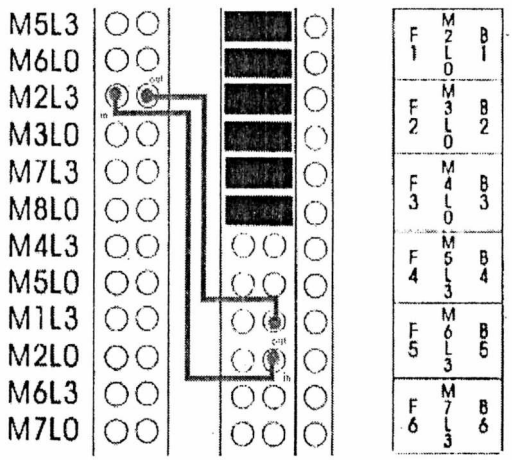


Figura C.7: grilla formada con el hipercubo

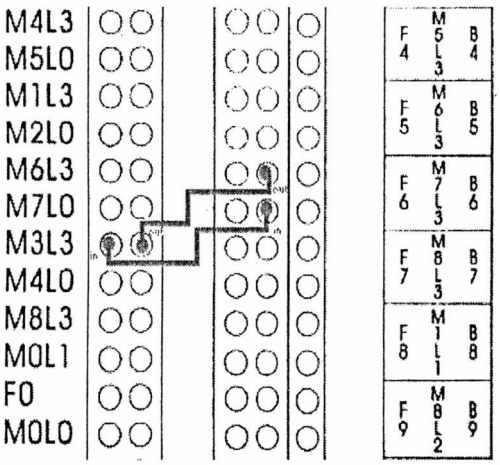
Para poder acceder a los links 3 de cada modulo a traves del front panel fue necesario realizar la siguiente serie de conecciones: Se conectaron mediante cables los pines del LINKIN y LINKOUT del módulo 1 link3 (M1L3) en JP1, con los pines LINKIN y LINKOUT respectivamente del link 5 del front panel (F5) en JP2, como se muestra en la figura C.8-a. De esta manera se puede acceder al link 3 del módulo 1 por medio del link 5 del front panel. Lo mismo se realizó para conectar el link 3 del módulo 2 (M2L3) con el link 6 del fornt panel (F6), el link 3 del módulo 3 (M3L3) con el link 7 del front panel (F7), y el link 3 del módulo 4 (M4L3) con el link 8 del front panel (F8), como se puede observar en las figuras C.8-b, C.8-c, y C.8-d, respectivamente.



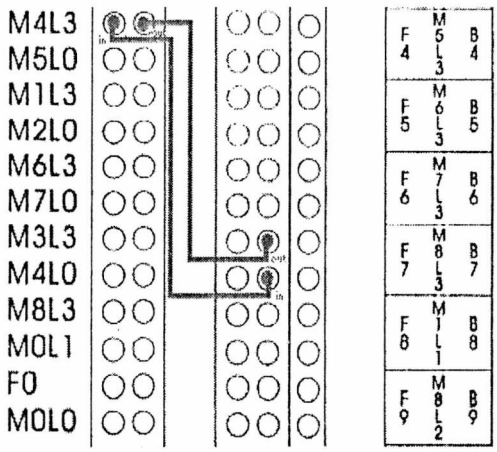
a) Conección del link 3 del módulo 1 al link 4 del Front Panel.



b) Conección del link 3 del módulo 2 al link 5 del Front Panel.



c) Conección del link 3 del módulo 3 al link 6 del Front Panel.



d) Conección del link 3 del módulo 4 al link 7 del Front Panel.

Figura 8: conecciones en los jumpers para tener acceso a los link 3 externamente.

Con esta nueva configuración ahora se puede acceder a los link 0 y 3 de cada transputer, teniendo ahora la posibilidad de formar la grilla de la figura 8.

Las conexiones necesarias para formar la grilla se efectúan mediante cables externos, y ahora el chassis presentará el aspecto indicado en la figura 9. Puede observarse que en el link 0 del módulo 1 de la primer placa llega el cable correspondiente a la señal del host. Esto impide poder conectar este link con el link 3 del módulo 1 de la ultima placa y formar un anillo en esta fila de la grilla. Por esto no se formo un anillo en cada fila de la grilla, a pesar que en las demás filas era posible realizarlo.

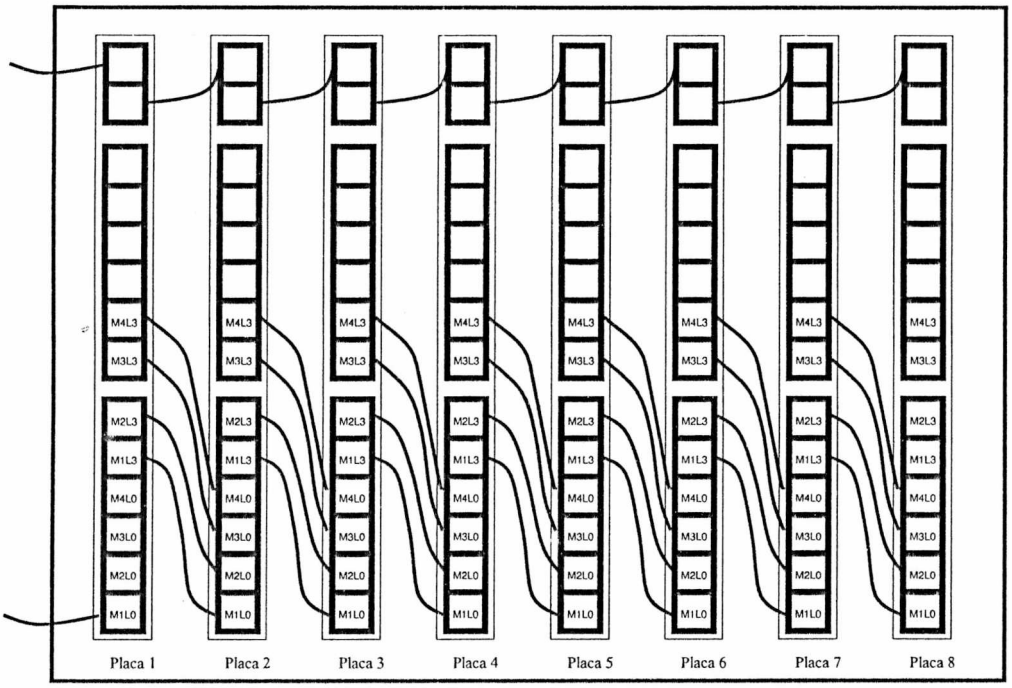


Figura C.9: vista externa del chassis del hipercubo configurado como grilla

Numeración de los transputers

La numeración asignada a cada transputer es bastante particular y puede observarse a partir de la herramienta CHECK [Tra94]. Esta numeración se obtiene formando un árbol de transputers y recorriendo este árbol por niveles. Debido a ésto, la númeración obtenida no será la númeración clásica para una grilla que se espera obtener.

El árbol es formado de la siguiente manera:

1. Primero se toma el transputer al cual le llega la señal del host (en nuestro caso el módulo 1 de la primera placa) como la raiz del árbol.
2. A continuación se agregan los transputers conectados a la raiz como sus hijos (se encontrarán en profundidad 1), ordenados por el número de link.
3. Luego se recorren los nodos de profundidad 1, y para cada uno se agregan los transputers conectados a éste como hijos (si no pertenecían anteriormente al árbol), ordenados por número de link.
4. Se repite el paso 3, aumentando la profundidad, hasta que todos los transputers se encuentren en el árbol.

La numeracíon del árbol también se realiza por profundidad: el transputer ubicado en la raiz (nivel 0) se numera con el número 0. Luego se numeran los nodos que se encuentran a profundidad 1 (nivel 1), y así sucesivamente para todos los niveles.

En la figura C.10 se muestra el árbol generado con su respectiva numeración para la grilla de la figura C.7. En tanto que en la figura C.11 se muestra cómo queda numerada la grilla. En estas figuras se utiliza la siguiente notación: para nombrar al transputer 3 (módulo 3) de la placa 5, se denota como *T3P5*.

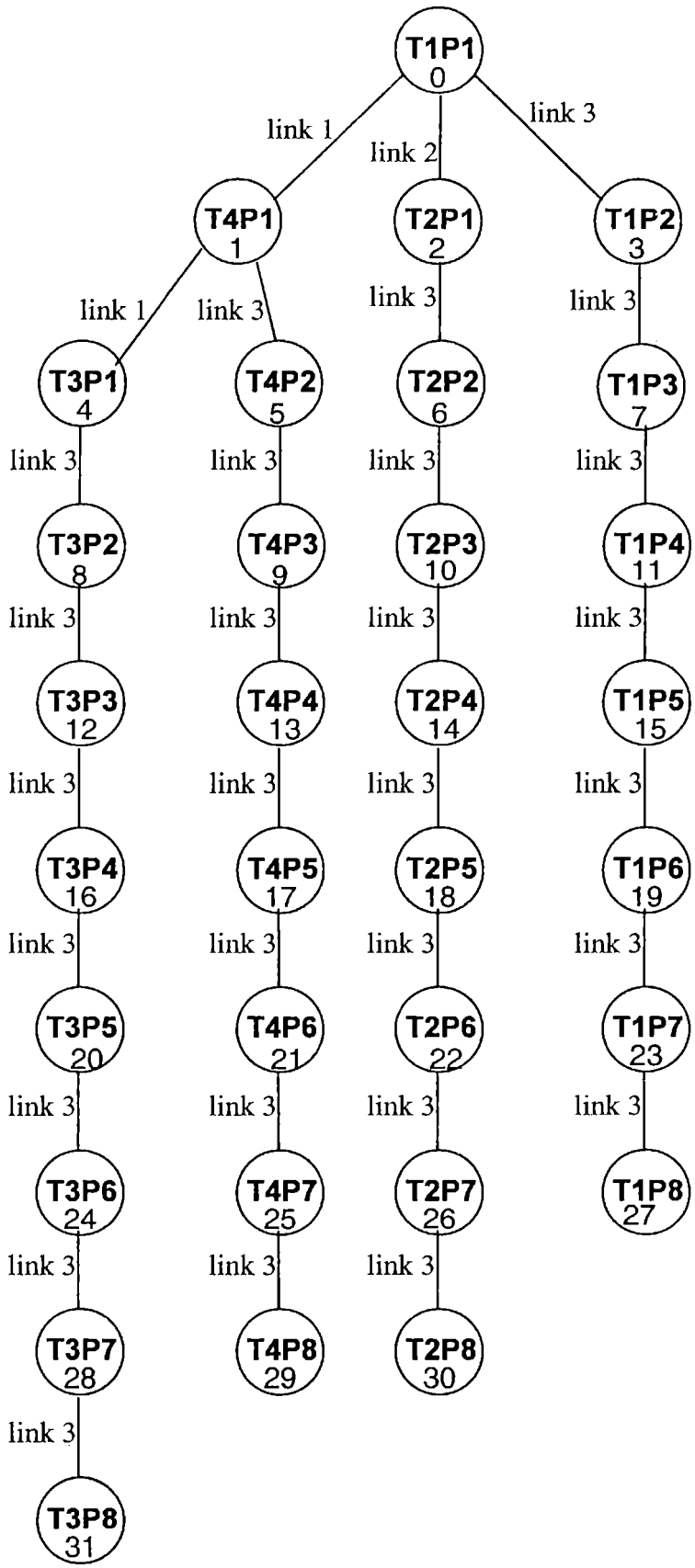

T1P1
0
link 1
link 2
link 3
T4P1
1
T2P1
2
T1P2
3
link 1
link 3
link 3
link 3
T3P1
4
T4P2
5
T2P2
6
T1P3
7
link 3
link 3
link 3
link 3
T3P2
8
T4P3
9
T2P3
10
T1P4
11
link 3
link 3
link 3
link 3
T3P3
12
T4P4
13
T2P4
14
T1P5
15
link 3
link 3
link 3
link 3
T3P4
16
T4P5
17
T2P5
18
T1P6
19
link 3
link 3
link 3
link 3
T3P5
20
T4P6
21
T2P6
22
T1P7
23
link 3
link 3
link 3
link 3
T3P6
24
T4P7
25
T2P7
26
T1P8
27
link 3
link 3
link 3
T3P7
28
T4P8
29
T2P8
30
link 3
T3P8
31

Figura C.10 : árbol generado para la numeración de la grilla.

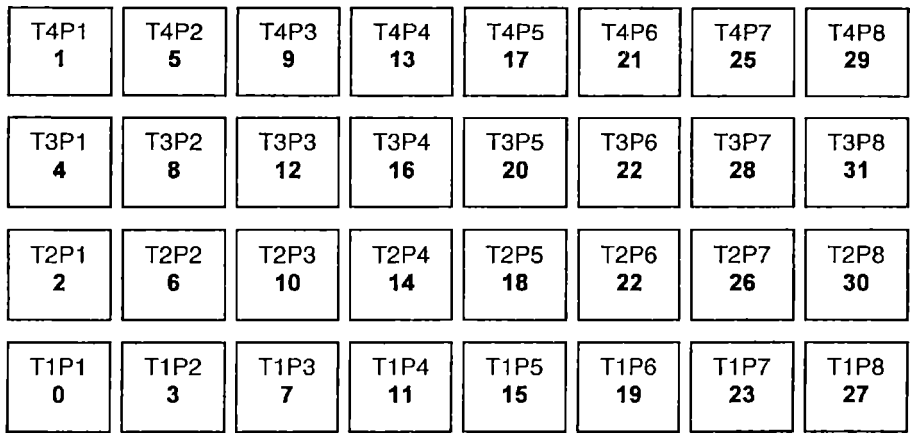


Figura C.11: numeración de la grilla obtenida del árbol de la figura 10.

Apéndice D: PVM para Windows 95/NT

Se describe en forma breve la implementación de PVM para los sistemas WIN32 [Mar96]. No existen restricciones para las aplicaciones que actualmente están usando PVM porque es completamente compatible con la versión actual de PVM3.

Nueva arquitectura

Actualmente, PVM está disponible en numerosas arquitecturas combinando estaciones de trabajo de Unix, máquinas con memoria compartida y procesadores de ejecución paralela masiva (MPP) en una única máquina paralela virtual. Estas arquitecturas son utilizadas, en la mayoría de los casos, en áreas científicas.

Otras compañías de software como Microsoft ofrecen sistemas operativos multiusuario (Windows NT) y multitarea (Windows 95) dominando el mercado. El creciente número de computadoras personales en las companías sugiere la necesidad de la utilización en forma conjunta de este recurso para el procesamiento paralelo.

Una de las principales características de PVM es que ofrece una interface de pasaje de mensajes de manera que las aplicaciones asuman que están ejecutando en una única máquina.

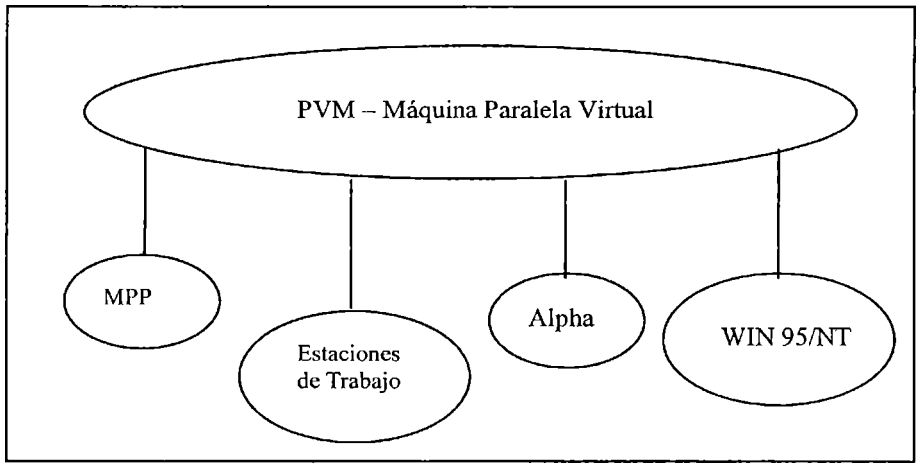


Fig D.1: Modelo de PVM

Implementación

Como el PVM original esta versión necesita una proceso daemon llamado pvmd, el cual es el encargado de "llevar el control de" (keep track) del administrador de tareas. El primer proceso pvmd se comportaba como un pvmd master. En la versión original, este proceso también es usado para comenzar daemons esclavos en otros hosts. En esta versión

usa un proceso hoster separado. Este es invisible al usuario, por lo que no notará ninguna diferencia.

El sistema PVM para WIN 32 utiliza tres procesos para implementar su característica de visión de única máquina: pvm-console, daemon pvm maestro y proceso hoster. Las principales funciones de estos procesos es la de comenzar o finalizar nuevas tareas, rutear mensajes entre tareas, como así también establecer una conexión directa para un mejor rendimiento.

Configuración de PVM para WIN32

Para el correcto uso de PVM para WIN32 requiere de las siguientes variables de ambiente,

- PVM_TMP especifica la locación de los archivos temporales (PVM_TMP=c:\temp)
- PVM_ROOT referencia a la instalación de pvm (PVM_ROOT=c:\pvm\pvm3)
- PVM_RSH establece la locación del comando rsh
- PVM_ARCH se le debe asignar WIN32

Tanto Windows 95 como Windows NT fueron diseñados para trabajo en red. Pero no proveen herramientas para el manejo de procesos remotos. Por la necesidad de que al menos un daemon adicional se ejecute en cada host, es necesario un remote shell deamon (rshd) para permitir el ingreso de hosts a la máquina virtual.

El proceso pvm-console provee una interface para una interacción más fácil del usuario con pvmd. A partir de esta interface es posible agregar nuevos hosts a la máquina virtual e ingresar claves. El usuario puede reiniciar la máquina paralela virtual si se producen errores en las tareas. Una vez iniciada la máquina virtual, nuevas tareas pueden ser comenzadas en la consola y el comando "ps –a" retorna información de las tareas que se están ejecutando en cada máquina ligada a PVM.

RSHD – Remote Shell Deamon

Provee acceso a computadoras remotas de manera que los procesos puedan ser localizados remotamente para su ejecución en el contexto del usuario [RSHD]. Es particularmente útil para la inter-operativilidad entre Unix y Windows 95/NT. Esto no permite comenzar una WIN32 GUI en una máquina remota y tener una pantalla gráfica. Esto no es parte de la funcionalidad de rshd.

El control de acceso generalmente se realiza por medio de un archivo en donde se explicitan los usuarios remotos que tendrán permiso para acceder a la PC. Se debe indicar el nombre del usuario con que ingresa a la máquina remota, el directorio al que tendrá acceso remotamente y una lista con los identificadores (nombre o dirección IP) de las máquinas que el usuario podrá ejecutar comandos. En el directorio indicado se ejecutarán los comandos enviados por el usuario remoto.

El programa rsh.exe es un comando de línea utilizado por los clientes que requieran la ejecución de un comando remoto. La sintaxis es la siguiente:

rsh *host* –l *usuario comando arg1 arg2* ...

Donde *host* es el nombre del host donde se desea ejecutar *comando* y *usuario*

APENDICE E: Regresión Lineal

Para el modelo determinístico $Y = \alpha + \beta^*X$ el valor observado de Y es una función lineal de X. Esto se generaliza a un modelo probabilístico, donde se asume que el valor esperado de Y es una función lineal de X; pero para X fijos, la variable Y difiere de su valor esperado en una cantidad aleatoria.

El modelo de regresión lineal simple utilizado es [Mey73][Dev87]:

$$Y = \alpha + \beta^*X + \varepsilon$$

donde Y (variable dependiente) es una variable aleatoria cuyo valor depende del valor de X (variable independiente); α y β son constantes (desconocidas); y ε es una variable aleatoria denominada **desviación aleatoria** o término de **error aleatorio**. Se realizan las siguientes hipótesis sobre ε:

$$E(\varepsilon) = 0; \qquad V(\varepsilon) = \sigma^2 \qquad \text{para todo } X,$$

donde $E(\varepsilon)$ es el valor esperado (esperanza) de ε; y $V(\varepsilon)$ es la varianza de ε (no se supone nada sobre la distribución de la variable aleatoria ε). Debido a que X es un valor fijo se cumple que:

$$E(Y) = \mathrm{E}(\alpha + \beta^*X + \varepsilon) = \alpha + \beta^*X + E(\varepsilon) = \alpha + \beta^*X, \text{ y}$$
$$V(Y) = \mathrm{V}(\alpha + \beta^*X + \varepsilon) = V(\alpha + \beta^*X) + V(\varepsilon) = 0 + \sigma^2 = \sigma^2$$

La relación $E(Y) = \alpha + \beta^*X$ indica que el valor medio de Y es una función lineal de X. La línea de regresión $Y = \alpha + \beta^*X$ será entonces la *línea de valores medios*; dado un valor de X en particular, Y será el valor esperado de X. La segunda relación indica que la variabilidad en la distribución de los valores de Y es la misma para diferentes valores de X.

Estimación de los parámetros del Modelo

Los valores α, β y σ^2 son desconocidos y serán estimados a partir de n observaciones $(x_1,y_1),..,(x_n,y_n)$. Se asume que estas observaciones fueron obtenidas independientemente una de las otras. Esto significa que y_i es el valor observado de la variable aleatoria Y_i, donde $Y_i=\alpha+\beta^*X_i+\varepsilon_i$; y las n desviaciones ε_1, $\varepsilon_{2,...,}$ ε_n son variables aleatorias independientes.

Por definición, los *estimadores de mínimos cuadrados* de α y β son los valores α y β que minimizan:

$$S(\alpha,\beta) = \sum_{i=1}^{n}\left[Y_i - (\alpha x_i + \beta)\right]^2$$

Para minimizar $S(\alpha,\beta)$, se deben resolver las ecuaciones:

$$\frac{\partial S}{\partial \alpha} = 0 \text{ y } \frac{\partial S}{\partial \beta} = 0.$$

obteniendo asi dos ecuaciones lineales con incógnitas α y β. Resolviendo estas ecuaciones se obtienen los estimadores de α y β: $\hat{\alpha}$ y $\hat{\beta}$ respectivamente.

$$\hat{\alpha} = \overline{Y} - \hat{\beta}\overline{x}, \text{ donde } \overline{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$$

$$\hat{\beta} = \frac{\sum_{i=1}^{n} Y_i (x_i - \bar{x})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}, \text{ donde } \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Estimación de σ^2

El parámetro σ^2 da una medida de la variabilidad inherente en el modelo de regresión. Un valor grande de σ^2 indica que las observaciones (x_i, y_i) están alejadas de la línea de regresión, en tanto que un valor pequeño de σ^2 señala que los puntos observados están muy cercanos a la línea de regresión. La estimación de σ^2 se efectúa teniendo en cuenta los **residuos** $y_i - \hat{y}_i$ donde $\hat{y}_i = \hat{\alpha} + \hat{\beta} x_i$ es el i-ésimo valor ajustado (cada residuo es la diferencia entre el valor observado y el predicho).

El error de sumas cuadradas *SEE* se define como

$$SEE = \sum (y_i - \hat{y}_i)^2 = \sum \left[y_i - \left(\hat{\alpha} + \hat{\beta} x_i \right) \right]^2$$

y a partir de este se define el estimador de σ^2

$$\hat{\sigma}^2 = s^2 = \frac{SSE}{n-2} = \frac{\sum (y_i - \hat{y}_i)^2}{n-2}$$

INTERVALOS DE CONFIANZA

Obtener sólo una estimación del valor de un parámetro generalmente es insatisfactorio. Si bien esta estimación representará nuestra mejor aproximación al valor real del parámetro, virtualmente nunca será igual a éste. Se necesita además de alguna medida que indique cuán cercano puede estar el valor estimado del valor real.

Un intervalo de confianza [Mey73][Dev87] reemplaza una estimación puntual (que es un valor simple) por un intervalo de posibles valores para el parámetro que se estima. El grado de plausibilidad de estos valores se especifica como el nivel de confidencia de un intervalo de confianza, y es la probabilidad de que el valor real pertenezca a este intervalo.

Intervalo de confianza de nivel (1-δ*) para* β

Se buscará un intervalo de confianza de nivel $(1-\delta)$ para β, $S(\beta)$, tal que

$$P(\beta \in S(\beta)) = 1 - \delta$$

Sabemos que $\hat{\beta} = \frac{\sum_{i=1}^{n} Y_i (x_i - \bar{x})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$, con $E(\hat{\beta}) = \beta$ y $Var(\hat{\beta}) = \frac{\sigma^2}{S_{XX}}$, donde

$S_{XX} = \sum_{i=1}^{n} (x_i - \bar{x})^2$. $\hat{\beta}$ se puede reescribir como $\sum_{i=1}^{n} c_i y_i$, con $c_i = \frac{(x_i - \bar{x})}{S_{XX}}$, entonces $\hat{\beta}$

es una función lineal de variables aleatorias independientes con distribución normal (porque c_i es una constante), por lo cual $\hat{\beta} \sim N(\beta, \frac{\sigma^2}{Sxx})$ ($\hat{\beta}$ tiene distribución Normal con $\mu=\beta$ y $\sigma^2=\frac{\sigma^2}{Sxx}$). Luego,

$$\frac{\hat{\beta}-\beta}{\frac{\sigma}{\sqrt{Sxx}}} \sim N(0,1) \quad \text{y} \quad \frac{SSE}{\sigma^2} \sim \chi^2(n-2)$$ (tiene distribución χ-cuadrado con n-2 grados de libertad)

Al ser estas últimas variables aleatorias independientes, se define T de la siguiente manera:

$$T = \frac{\frac{\hat{\beta}-\beta}{\frac{\sigma}{\sqrt{Sxx}}}}{\sqrt{\frac{SSE}{(n-2)\sigma^2}}} \sim t(n-2)$$ (tiene distribución *t* de student con n-2 grados de libertad)

reescribiendo llegamos a que $T = \frac{\hat{\beta}-\beta}{\sqrt{\frac{MSE}{Sxx}}}$, donde $MSE = \frac{SSE}{n-2}$.

El intervalo de confianza de nivel 1 - δ para β es:

$$S(\beta) = \left[-t_{1-\frac{\delta}{2}}(n-2) \le \frac{\hat{\beta}-\beta}{\sqrt{\frac{MSE}{Sxx}}} \le t_{1-\frac{\delta}{2}}(n-2) \right]$$

donde $t_{1-\delta}(n-2)$ es tal que $P(X \le t_{1-\delta}(n-2)) = 1-\delta$; con $X \sim t(n\text{-}2)$

despejando β obtenemos:

$$S(\beta) = \left[\hat{\beta} - \sqrt{\frac{MSE}{Sxx}} t_{1-\frac{\delta}{2}}(n-2) \le \beta \le \hat{\beta} + \sqrt{\frac{MSE}{Sxx}} t_{1-\frac{\delta}{2}}(n-2) \right]$$

Intervalo de confianza de nivel (1-δ) para α

Se buscará un intervalo de confianza de nivel (1-δ) para α: $S(\alpha)$ tal que

$$P(\alpha \in S(\alpha)) = 1 - \delta$$

Se procede en forma similar que con $\hat{\beta}$. Sabemos que $\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$, con $E(\hat{\alpha})=\alpha$ y $Var(\hat{\alpha})=\sigma^2\left(\frac{1}{n}+\frac{\bar{x}^2}{S_{XX}}\right)$; $\hat{\alpha}$ es combinación lineal de variables aleatorias con distribución Normal, entonces $\hat{\alpha} \sim N\left(\alpha, \sigma^2\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)\right)$. Luego,

$$\frac{\hat{\alpha}-\alpha}{\sigma^2\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)} \sim N(0,1) \quad \text{y} \quad \frac{SSE}{\sigma^2} \sim \chi^2(n-2) \quad \text{son variables aleatorias independientes.}$$

Se define T de la siguiente manera:

$$T = \frac{\dfrac{\hat{\alpha}-\alpha}{\sigma^2\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)}}{\sqrt{\dfrac{SSE}{(n-2)\sigma^2}}} \sim t(n-2)$$

reescribiendo llegamos a que $T = \dfrac{\hat{\alpha}-\alpha}{\sqrt{MSE\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)}}$.

El intervalo de confianza de nivel 1 - δ para α es:

$$S(\alpha) = \left[-t_{1-\frac{\delta}{2}}(n-2) \le \frac{\hat{\alpha}-\alpha}{\sqrt{MSE\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)}} \le t_{1-\frac{\delta}{2}}(n-2) \right]$$

despejando α obtenemos:

$$S(\alpha) = \left[\hat{\alpha} - \sqrt{MSE\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)}\, t_{1-\frac{\delta}{2}}(n-2) \le \alpha \le \hat{\alpha} + \sqrt{MSE\left(\frac{1}{n}+\frac{\bar{X}^2}{S_{XX}}\right)}\, t_{1-\frac{\delta}{2}}(n-2) \right]$$

Intervalo de confianza de nivel (1-δ) para σ^2

El estimador de σ^2 es $\hat{\sigma}^2 = \dfrac{SSE}{(n-2)}$. Se buscará un intervalo de confianza de nivel (1-δ) para σ^2: $S(\sigma^2)$ tal que $P(\sigma^2 \in S(\sigma^2)) = 1 - \delta$

A partir de $\frac{SSE}{\sigma^2} \sim \chi^2(n\text{-}2)$ se define $S(\sigma^2)$ de la siguiente manera:

$$S(\sigma^2) = \left[\chi^2_{\frac{\delta}{2}}(n-2) \leq \frac{SSE}{\sigma^2} \leq \chi^2_{1-\frac{\delta}{2}}(n-2) \right]$$

despejando σ^2 obtenemos:

$$S(\sigma^2) = \left[\frac{SSE}{\chi^2_{1-\frac{\delta}{2}}(n-2)} \leq \sigma^2 \leq \frac{SSE}{\chi^2_{\frac{\delta}{2}}(n-2)} \right]$$

donde $\chi^2_{\alpha}(n-2)$ es tal que $P(X \leq \chi^2_{\alpha}(n-2)) = 1-\delta$; con $X \sim \chi^2$ (n-2)

Bibliografía

[Aho] Aho A, Hopcroft J, Ullman J, "Estructuras de Datos y Algoritmos", Addison-Wesley Iberoamericana S.A., Wilmington, Delaware, E.U.A.

[Alt94] Alta Technology, "VME-XP: Installation Guide and User Reference Manual" Version 1.3, November 1994.

[And91] Andrews G, " Concurrent Programming: Principles and Practice", The Benjamin/Cummings Publlishing Company, Inc., 1991.

[Cho] Choi J. "A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers", School of Computing Soongsil University, Seoul, KOREA.

[Cho92] Choi J, Dongarra J, Walker D. "The design of scalable software libraries for distributed memory concurrent computers". In Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilairedu Touvet , France). Elsevier Science Publishers, September 7-8, 1992.

[Cho94] Choi J, Dongarra J, Walker D. "PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers". Concurrency: Practice and Experience, 1994.

[Dev87] Devore J, "Probability and Statistic for Engeneering and Sciences", Brooks/Cole Publishing Company, Segunda Edición, 1987.

[Gei94] Geist Al, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V, "PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing", MIT Press, Scientific and Engineering ComputationJanusz Kowalik, 1994.

[Gei95] Van de Geijn R, Watts J. "SUMMA Scalable Universal Matrix Multiplication Algorithm". LAPACK Working Note 99, Technical Report CS-95-286, University ofTennessee, 1995.

[Hip95] Hipper G, Tavangarian D, "A Concurrent Communication Architecture for Workstation Clusters", Proceedings of the ISMM International Conference on Intelligent Information Management Systems, Washington, D.C., 1995.

[Hoc82] Hockney R, "Characterization of Parallel Computers and Algorithms", Computer Physics Communications, 26:285-29, 1982.

[Hoc88] Hockney R, Jesshope C, Parallel Computers 2: Architecture, Programming and Algorithms", Adam Hilger/IOP Publishing, Bristol&Philadelphia, Second Edition, 1988.

[Ker85] Kernighan B, Ritchie D, "El lenguaje de Programación C", Prentice Hall, 1985.

[Kum94] Kumar V, Grama A, Gupta A, Karypis G, "Introduction to Parallel Computing. Design and Analysis of Algorithms", The Benjamin/Cummings Publishing Company, Inc., 1994.

[Lei92] Leighton F, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes", Morgan Kaufman Publishers, 1992.

[Log94a] Logical System, "Parallel 'C' for the Transputer: Toolset Reference", 1994.

[Log94b] Logical System, "Parallel 'C' for the Transputer: C Library Reference", 1994.

[Mar96] Markus F, Dongarra J, "Another architecture: PVM on Windows 95/NT", 1996.

[Mey73] Meyer P, "Probabilidad y Aplicaciones Estadísticas", Editorial Fondo Educativo Iberoamericano, Segunda Edición, 1973.

[Sil91] Silberschatz A, Peterson J, Galvin P, "Sistemas operativos. Conceptos Fundamentales", Addison Wesley, 1991.

[Tra94] Transputer Education Kit, Distribution Notes, CHECK2V0.DOC, 1994.

[Thi95] Thiébaut D, "Parallel Programing in C for the transputer" (http://cs.smith.edu/ ~thiebaut), 1995.

[Tin98] Tinetti F, De Guisti A, "Procesamiento Paralelo. Conceptos de Arquitecturas y Algoritmos", Editorial Exacta, 1998.

[Tin98a] Tinetti F, "Aplicaciones Paralelas de Cómputo Intensivo en NOW Heterogéneas".