Centro de Investigación y de Estudios Avanzados del

Instituto Politécnico Nacional

CINVESTAV-IPN

Departamento De Ingeniería Eléctrica

Sección Computación

KHipeR:

KerneL Escalable Para Una Red de Transputer

En Topología Hipercubo

Tesis que presenta el Ing. Gustavo Sergio Téllez Rangel para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERÍA ELÉCTRICA con opción en COMPUTACIÓN.

Trabajo dirigido por el M.en C. José Oscar Olmedo.

México D.F., Mayo de 1995.

INDICE

La necesidad de una mayor potencia computacional crece continuamente. La solución tradicional de crear procesadores más rápidos se muestra cada vez más inadecuada El costo de las supercomputadoras, sistemas que usan estos procesadores especiales, es sumamente restrictivo. Además con la cercanía cada vez mayor al límite de la velocidad en los circuitos de silicio se ha visto la necesidad de investigar intensamente sobre otras alternativas. Por los resultados que se han obtenido hasta este momento se observa que los sistemas concurrentes pueden ser una excelente solución.

Una solución al requerimiento de computadoras más veloces se encuentra en formar redes de procesadores, una que ha mostrado amplias ventajas es el hipercubo. Un procesador diseñado para este tipo de redes se llama transputer.

Conectar varios transputers para formar una red es extremadamente simple, sin embargo esta simplicidad en hardware se paga al momento de diseñar y codificar la comunicación y sincronización de los procesos que constituyen al programa.

Dentro de la solución desarrollada se encuentra una extensión al lenguaje de programación C con sintaxis que permite el uso de construcciones concurrentes, esto bajo el modelo c.s.p. el cual ha mostrado amplias ventajas.

Se puede disminuir el tiempo de ejecución de un programa si este se divide en tareas y cada una de estas se asigna a un procesador diferente, cada tarea se llama proceso. Para lograr lo anterior se han desarrollado tecnologías a nivel hardware y software.

En este trabajo para lograr la comunicación entre los procesadores se utiliza una red que se puede dibujar como un cubo, en cada esquina se coloca un procesador el cual solo se puede comunicar con aquellos que se encuentran en los extremos de los arcos que inciden en él. A esta red se le conoce como hipercubo.

Los lenguajes actuales para programar procesadores transputer proporcionan mecanismos de comunicación y control simple de ellos. Ambos sufren del problema de poder ser usados de una forma directa solo si su alcance está limitado al procesador. Para usar una red el programador debe desarrollar sus propios mecanismos de comunicación y control de procesos a lo largo de ella, una actividad que es necesaria pero no productiva.

Otra actividad no productiva es el mapeo, la colocación de los procesos en los procesadores, que dado que depende de la arquitectura con la que se esté trabajando fija el programa a la topología, disminuyendo su portabilidad. Además dado que el programador no conoce con exactitud la ejecución de los procesos puede ser ineficiente.

Para programar una red es necesario un lenguaje capaz de expresar la comunicación entre los procesos y su sincronización con independencia de su colocación. Este lenguaje debe ser sustentado con un modelo eficiente, uno que se ha mostrado bastante adecuado y de amplia aceptación es conocido como c.s.p.

Además la implantación de este lenguaje debe ocultar detalles de la arquitectura sobre la que se está trabajando a fin de permitir que los programas sean portables.

Los objetivos de este trabajo son:

- *Desarrollar extensiones al lenguaje C para que soporte el modelo c.s.p.*
- *Ofrecer una capa de software para facilitar la programación de un hipercubo de transputers.*
- *El mapeo debe ser realizado por la mencionada capa de software de una manera eficiente.*
- *El lenguaje debe ocultar la arquitectura proporcionando portabilidad de los programas.*

1 CONCEPTOS BASICOS

1.1 Hipercubo

Un sistema concurrente se puede implementar como una red de computadoras o como una red de procesadores dentro de una computadora, llamadas multicomputadoras o multiprocesadores, respectivamente.

Se ha demostrado que el hipercubo es una de las más versátiles y eficientes redes descubiertas. El de orden d está formado por 2^d nodos y (d x 2^d) líneas de comunicación. El número que identifica a uno de sus procesadores se establece a través de una cadena de d bits. Definido recursivamente como:

a) El hipercubo de orden cero está formado por un solo procesador. La cadena que lo identifica es un 0.

b) Un hipercubo de orden d se forma con dos hipercubos de orden d-1. La cadena de los procesadores del primero inicia con cero y los del segundo con uno.

Lo cual explica dos características atractivas: su diámetro pequeño y su ancho de bisección alto.

Una forma de representar a un hipercubo es darle una apariencia geométrica.
en tercera dimensión. Otra es mostrar sus grupos de enlaces. Se elige la representación que facilite la comprensión del algoritmo utilizado. A continuación vemos ambas para un hipercubo de orden 4.

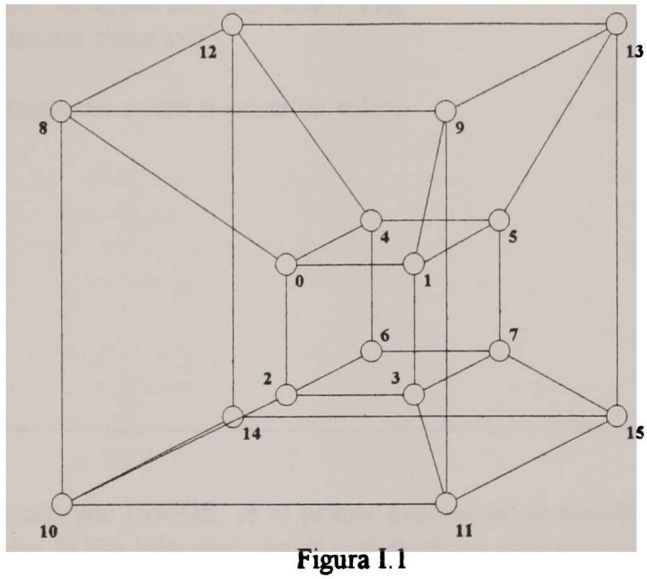


Figura I.1

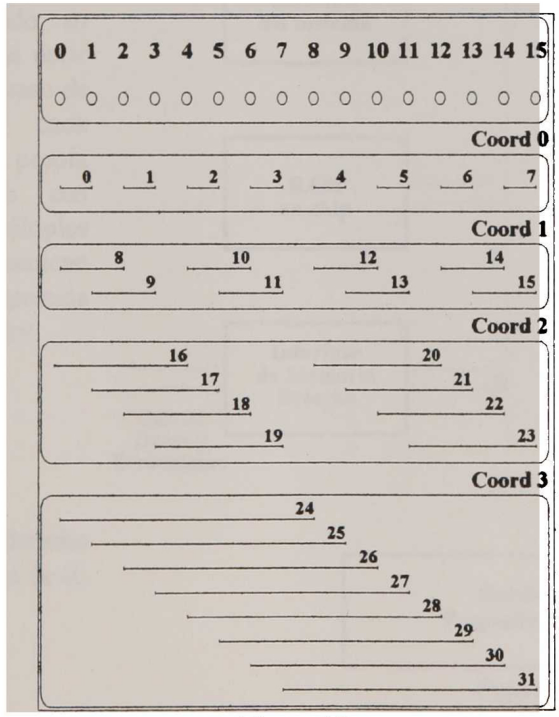


Figura I.2

Una característica importante del hipercubo es la manera como se pueden representar sus características.

Un ejemplo es como cada cara tiene su especificación a nivel de bits. En la figura de la derecha, la cara izquierda está formada por los nodos 000, 100, 010 y 110, por lo cual la podemos denotar como xx0.

Un procesador es adyacente a otro si existe un enlace que los conecte.

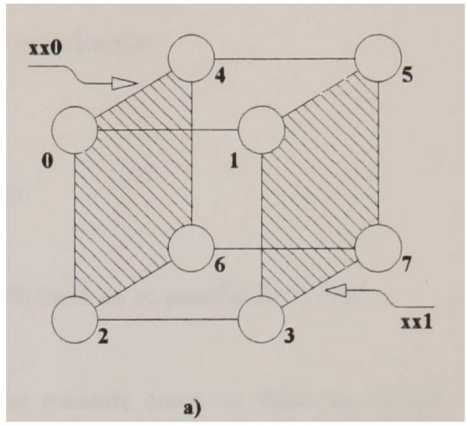


Figura I.3

I.2. Transputer

El transputer, diseñado por INMOS, es el primer procesador encapsulado en una sola pastilla que proporciona un procesador de alta velocidad, rápida comunicación entre procesadores y un soporte explícito para sistemas de múltiples procesos y múltiples procesadores.

Fue diseñado con la idea básica de lograr un dispositivo que pudiera ser usado en arquitecturas MIMD, es decir aquellas basadas en el paso de mensajes, donde cada procesador tiene su propia memoria física, pero con soporte para múltiples procesos que se comunican por memoria compartida dentro de cada procesador.

En la figura de la derecha podemos ver el diagrama de un transputer genérico.

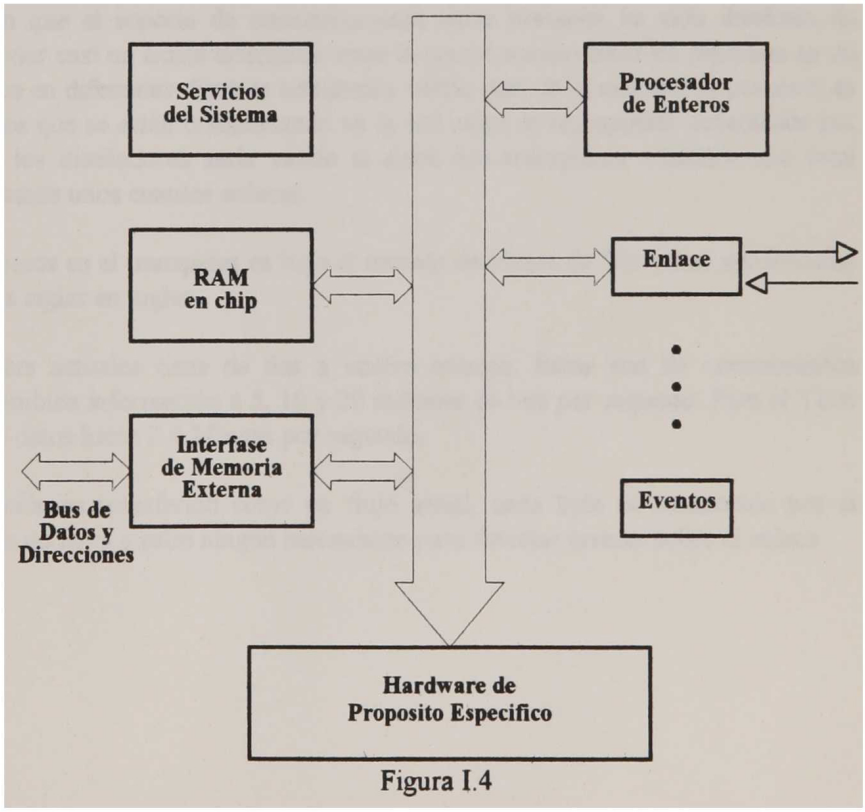


Figura I.4

Las características comunes de todos los procesadores transputer actuales son:

- Procesador de enteros de alta velocidad.
- Procesador con calendarizador de procesos en micro código.
- Memoria estática rápida inmersa en la pastilla.
- Hasta cuatro enlaces para comunicación con otros transputers.
- Temporizadores internos.
- Interfase de memoria externa.
- Adicionalmente el modelo T805 cuenta con unidad de punto flotante en su pastilla de 32 bits.

El concepto central del transputer es el proceso. Un proceso se entiende como un flujo de control individual, *hilo*. El transputer conmuta entre los procesos que está ejecutando, es decir proporciona multitarea. Tradicionalmente esta es manejada por un sistema operativo, en el caso de computadoras basadas en el transputer es totalmente manejada por el calendarizador de este.

El calendarizador, en micro código, mantiene dos colas de procesos: una de alta y otra de baja prioridad. Los procesos de alta prioridad son ejecutados hasta que terminan, los de baja prioridad solo durante una rebanada de tiempo de aproximadamente 1 milisegundo y pueden ser interrumpidos por los de alta prioridad. En ambos casos dejan de ejecutarse si llegan a requerir mandar o recibir información.

El transputer fue diseñado para ser usado en multiprocesadores basados en paso de mensajes por lo cual tiene un fuerte soporte para las comunicaciones entre procesos.

Los diseñadores insisten en que el soporte de comunicaciones entre procesos ha sido diseñado de manera tal que para el programador casi no existe diferencia entre la comunicación entre los procesos en un mismo transputer y los colocados en diferentes. Esto es totalmente válido cuando el número de procesos es uno por transputer y los procesos que se están comunicando en la red están en transputers conectados por sus enlaces. La afirmación de los diseñadores sería válida si entre dos transputers existiera una total conectividad no únicamente mediante unos cuantos enlaces.

La comunicación entre procesos en el transputer es bajo el modelo de Hoare de "Procesos secuenciales que se comunican", c.s.p. por sus siglas en ingles.

Cada uno de los transputers actuales tiene de dos a cuatro enlaces. Estos son de comunicación completamente doble que intercambian información a 5, 10 y 20 millones de bits por segundo. Para el T800 puede lograrse el intercambio de datos hasta 2.4 Mbytes por segundo.

El intercambio de información es transferido como un flujo serial, cada byte es reconocido por el transputer receptor. El transputer no lleva a cabo ningún mecanismo para detectar errores sobre el enlace.

I.3. Breve Nomenclatura Gráfica

Los dibujos que acompañan a las explicaciones de este trabajo siguen algunas reglas que uniformizan su significado.

- Un proceso y una función serán expresados mediante círculos, elipses o una figura similar. La creación de un proceso estará indicada mediante una línea punteada entre el y su creador.

- La invocación de una función se expresa mediante una línea continua.

- Así, un árbol formado con líneas continuas es llamado un árbol de activaciones en tanto que a un árbol formado con líneas punteadas, que representa la creación de procesos, se le llama árbol de creaciones.

- Cuando se requiera separar el ambiente de un procesador se hará mediante líneas a rayas.

2 PRESENTACION DEL SISTEMA

II.1. Elección de los componentes del sistema

II.1.1 Elección de la topología

Dentro de las diferentes topologías débilmente acopladas, de amplia granularidad, que se conocen para la implantación de multiprocesadores, una de las que ha mostrado mayores beneficios es el hipercubo.

Es eficiente tanto para llevar a cabo tareas de propósito general como tareas de propósito específico. Debido a que puede simular eficientemente cualquier arreglo de O (N) árboles binarios o mallas de árboles con una pequeña pérdida de rendimiento [4].

El hecho de que las redes mencionadas pueden ser contenidas por el hipercubo, ya sea completa o parcialmente, permite que sean simuladas por este.

Otras características sobresalientes del hipercubo son:

- Diámetro pequeño
- Facilidad para ser escalado
- Gran cantidad de algorítmos que por su propia naturaleza son directamente mapeables [7] a esta topología, entre ellos la FFT, diferentes clases de ordenamientos y operaciones de matrices.

Es una topología, que por sus características, ha obtenido éxito comercial. Varios fabricantes han diseñado computadoras paralelas con esta arquitectura.

II.1.2 Elección del modelo

La elección del transputer conlleva la elección del modelo c.s.p., ya que fue diseñado basado en él, por lo cual se mencionan primero las características del modelo.

C.s.p. es uno de los modelos más recientes que ha demostrado su poder para plasmar algorítmos concurrentes y ser una excelente solución a la comunicación y sincronización de procesos. A continuación se mencionan sus características relevantes.

- Usa guardias y el comando alternativo para la declaración y control del no determinismo.
- Como mecanismo de control de procesos concurrentes propone el comando "par" donde se especifica la ejecución concurrente de procesos.
- El modelo c.s.p. comunica procesos por paso de mensajes, mediante canales.
  Un canal es unidireccional y su uso es exclusivo al par de procesos que comunica.
  El paso de mensajes tiene la función adicional de sincronizar los procesos.
- El uso de comandos de entrada y de salida para lograr la comunicación entre procesos.
  La comunicación ocurre cuando un proceso nombra a otro. Lo puede nombrar como origen para la entrada o como destino para la salida de información. Esta es copiada de un proceso al otro.
- Plantea la ausencia de buffering automático. Es invisible el retardo que sufre un proceso en espera de la disponibilidad de la correspondiente salida o entrada del otro proceso.
- Los comandos de entrada pueden ser utilizados como guardias
- Un comando repetitivo puede tener comandos protegidos, terminando cuando todos los procesos componentes han terminado.
- Las construcciones que plantea son paralela, alterna y secuencia, conocidas como par, alt y seq, respectivamente.
- La construcción par genera los procesos que la constituyen y termina cuando el último de ellos termina.
- La construcción alt crea un proceso si existe un guardia con su condición cierta o con su entrada lista, en caso de que sean varios los guardias listos alt elige uno al azar.
- La construcción seq ejecuta secuencialmente los procesos que lo constituyen.

II.1.3 Elección de procesador

El transputer es un procesador específicamente creado para el desarrollo de computadoras paralelas. Con él se pueden formar redes fácilmente sin tener conocimientos de electrónica.

A continuación enumero las características por las cuales elegí al transputer:

- ❑ Su enorme facilidad para formar redes.
- ❑ Se puede adquirir en una tarjeta, la cual se inserta en alguna de las ranuras de la PC.
- ❑ Se puede formar una red de transputer utilizando una o varias PC.
- ❑ Kernel integrado en hardware, eficiente en el cambio de contexto.
- ❑ Enlaces bidireccionales de alta velocidad

El transputer proporciona canales software y hardware, de acuerdo a la denominación de los diseñadores, canales y enlaces (links) respectivamente. Solo cuando se requiera de claridad se usará la denominación de canales software o hardware.

Un canal se implementa como una localidad de memoria, un enlace es un recurso físico. La memoria es la única limitación al número de canales. El número de enlaces es finito, el modelo T805 cuenta con 4.

Se utiliza un canal o un enlace de acuerdo a la colocación de los procesos que comunica, los primeros para procesos en el mismo procesador, enlaces en caso contrario.

Se crea un algoritmo considerando canales abstractos. Cuando se implementa se decide cuales se convierten en canales y cuales en enlaces.

II.1.4 Elección del lenguaje

Entre los enfoques para programar computadoras paralelas destacan dos:

- ❑ Desarrollo de lenguajes nuevos

  Da la posibilidad de una total claridad para las operaciones concurrentes, su desventaja reside en la resistencia que presentan los programadores antiguos a aprender lenguajes nuevos, así como la gran cantidad de software en lenguajes ya existentes.

- ❑ Ampliación de la sintaxis de lenguajes ya en uso.

  Ofrece la facilidad de trabajar con lenguajes conocidos, su desventaja consiste en que no existen todavía estándares para la sintaxis de concurrencia. Opté por esta última.

Elegí al C como el lenguaje al cual ampliarle su sintaxis por las siguientes razones:

- las ventajas que proporciona a la programación de sistemas
- la gran cantidad de software existente
- la amplia aceptación que goza por parte de programadores
- la existencia de compiladores para todas las computadoras comerciales.

Para elegir el compilador comparé el de "Logical Systems" y el de "3L". Opté por el segundo debido a su claridad para configurar topologías. Y aún cuando el primero ofrece una construcción par esta controla únicamente procesos locales, lo cual no marca una diferencia importante.

Los compiladores mencionados ofrecen funciones para comunicar a dos procesos por medio de un canal o un enlace. Para comunicar a dos procesos separados por varios enlaces, se requieren procesos adicionales en cada procesador intermedio para recibir y pasar los mensajes.

Entre dos procesadores que requieran múltiples canales se tendrá que compartir el enlace asignado por rebanadas de tiempo, lo que se llama multiplexaje. Los compiladores ofrecen funciones para la creación de procesos multiplexores, sin embargo estos acaparan el uso de los canales.

Cuando se espera comunicación de uno de varios procesos se usa la función alt. Sólo cuando todos los procesos están en el mismo procesador se puede usar la función proporcionada por el compilador.

De las construcciones que el modelo c.s.p. ofrece son par, alt y seq.
La construcción seq es la única que proporciona, implícitamente, el compilador 3L y es la que los lenguajes tipo algol ofrecen.

II.2. Problemas

II.2.1 Portabilidad y Abstracción

Los compiladores mencionados ofrecen las funciones necesarias para la programación concurrente y los mecanismos para trabajar con redes, sin embargo estos últimos fijan fuertemente el programa a la topología, por lo cual el programador cuando desarrolla una aplicación debe cuidar los siguientes aspectos acerca de los procesos:

- Comunicación entre ellos.
- El mapeo de procesos, es decir la relación proceso-procesador.
- El control de ellos de acuerdo a las construcciones concurrentes requeridas en la aplicación.

Cada uno de los puntos expuestos se desarrolla en las siguientes subsecciones y se indican a continuación las desventajas que crean:

- Resta abstracción a los algorítmos
- Disminuye la portabilidad del programa.

Sin embargo, es el sistema operativo, o una capa de software más simple, quien en última instancia debe encargarse de los aspectos mencionados. De esta manera el programador dirige su esfuerzo únicamente en la búsqueda de una solución independiente de la arquitectura.

II.2.2 Comunicación entre procesos

Conectar varios transputers para formar una red es extremadamente simple, sin embargo, el precio que se paga es en la programación al desarrollar la comunicación y sincronización de los procesos colocados en diferentes procesadores. En c.s.p. comunicación y sincronización son dos aspectos de un mismo proceso.

Con cada implementación de un algoritmo se deben crear los multiplexores necesarios. Un trabajo no inherente al algoritmo mismo, necesario pero no creativo, que además disminuye la portabilidad del programa.

Son dos los problemas que el programador debe resolver en ***cada nueva*** aplicación con respecto a la creación de procesos adicionales:

- Para retransmitir mensajes, los ruteadores.
- Los encargados del multiplexaje.

En la práctica ambos tipos de procesos se funden en uno solo que se encarga de las funciones de ambos. La complejidad de este se incrementa debido a las características de sincronización del modelo, para lo cual se deben llevar tablas con la información del estado de los canales.

II.2.3 Mapeo de los procesos

Un sistema basado en paso de mensajes se degrada rápidamente si el tráfico de comunicaciones es excesivo, por lo cual un buen mapeo tiene como consecuencia un buen rendimiento.

El mapeo debe lograr que los procesos que se comunican queden lo más cerca posible, teniendo en cuenta que la carga de trabajo por cada procesador esté balanceada, es decir utilizar al máximo cada uno de los procesadores.

Algunos algoritmos paralelos consisten de procesos cuyo tiempo de ejecución no es uniforme, en estos casos la disponibilidad de los procesadores tampoco es uniforme. Por su carácter dinámico esta disponibilidad no se puede planear y sólo a tiempo de ejecución se obtiene la información necesaria. Esta es la razón por la cual un mapeo estático, en general, no es óptimo.

Además un mapeo estático disminuye la portabilidad, puesto que esta actividad en sí misma es ajustarse a la arquitectura. Por lo cual es deseable un mapeo dinámico.

II.2.4 Control de los procesos

El alcance explícito de las construcciones par y alt que ofrecen los compiladores mencionados [11] es local, de manera que si se desea que estas construcciones controlen procesos corriendo en otros procesadores se deberá recurrir a estrategias que como desventaja disminuirán la portabilidad de la aplicación.

Sí las construcciones par y alt funcionan únicamente para procesos corriendo en el mismo procesador no se puede contar con ellas cuando se trabaja con redes.

Dichas construcciones sólo pueden ser implantadas en una red si previamente se tiene una capa de comunicaciones que permita el control sobre toda la red.

Cada topología tiene su propio algoritmo de ruteo. Dado que el número de topologías posibles es enorme es obvio que ningún compilador pueda proporcionar una capa de comunicaciones estándar.

II.3. Solución planteada

De acuerdo a lo planteado los problemas a resolver son:

- Abstracción de los algoritmos.
- Portabilidad de las aplicaciones.

Para lograr lo anterior se deben resolver los siguientes problemas:

- Comunicación entre procesos.
- Mapeo de procesos.
- Control de procesos de acuerdo a las construcciones concurrentes.

La portabilidad y abstracción se resuelve eligiendo un modelo que tenga amplia aceptación y después el software que soporte este modelo. De manera que son dos las capas de software a implementar:

- Capa de Comunicaciones Abstractas
  La comunicación se resuelve implementando una capa que enlace a los diferentes procesadores, dando facilidades de alto nivel a la comunicación. Se denomina *abstracta* para referirse a que el usuario de ella no requiere del conocimiento de la topología.

- Capa de Control de procesos.
  Para el mapeo y control de las construcciones concurrentes.

Se denominan ***KHiper-Com*** y ***KHiper-Control*** respectivamente, en conjunto ***KHiper***. En la literatura a este conjunto se le conoce como kernel.

Además, la capa de control puede proporcionar facilidades adicionales para el control de los canales de KHiper-Com y sobre la entrada-salida.

El kernel del transputer responde únicamente a las necesidades de los procesos corriendo dentro del procesador, no en cuanto a los procesos corriendo en los diferentes procesadores de la red. Este trabajo consiste en proporcionar un control global.

II.3.1 Extensiones a la sintaxis del lenguaje C

Se muestra el modelo c.s.p. y luego se plantea como extensiones al lenguaje C. Se muestran alternadamente parte del modelo c.s.p. y las extensiones correspondientes para legibilidad de la sintaxis propuesta.

Para describir las gramáticas del modelo y las extensiones al lenguaje C, se utiliza la forma normal de Backus-Naur.

Para lograr consistencia con el lenguaje C los identificadores de las extensiones están en inglés.

Modelo c.s.p.

<comando paralelo>	::=	[<proceso> { | | <proceso>}]
<proceso>	::=	<etiqueta del proceso> <lista de comandos>
<etiqueta del proceso>	::=	<vacío> | <identificador> ::
	|	<identificador> (<etiqueta del subíndice> {,<etiqueta del subíndice>}) ::
<etiqueta del subíndice>	::=	<constante entera> | <rango>
<constante entera>	::=	<numeral> | <variable limitada>
<variable limitada>	::=	<identificador>
<rango>	::=	<variable limitada> : <límite inferior> .. <límite superior>
<límite inferior>	::=	<constante entera>
<límite superior>	::=	<constante entera>

Extensión al lenguaje C.

<comando paralelo>	::=	'parbegin' '{'
		{ {<declaración de proceso>} | {<lista de comandos>} | { 'seq' } }
		'parend' '}'
<declaración de proceso>	::=	'paritem' '(' <pid>, <pid>, <número de argumentos> {, <argumentos> } ')'
<número de argumentos>	::=	<int>
<argumentos>	::=	<char>
		<int>
		<float>
		<double>

Modelo c.s.p.

<comando de entrada>	::=	<origen> ? <variable>
<comando de salida>	::=	<destino> ! <expresión>
<origen>	::=	<nombre del proceso>
<destino>	::=	<nombre del proceso>
<nombre del proceso>	::=	<identificador> | <identificador> (<subíndices>)
<subíndices>	::=	<expresión entera> {,<expresión entera>}

Extensión al lenguaje C.

<comando de entrada>	::=	<entrada de palabra>
	|	<entrada de mensaje>
	|	<entrada guardia de palabra>
	|	<entrada guardia de mensaje>
<comando de salida>	::=	<salida de palabra>
	|	<salida de mensaje>
<entrada de palabra>	::=	'inword' '(' <chan>, <variable destino> ')'
<entrada de mensaje>	::=	'inmess' '(' <chan>, <mensaje destino> ')'
<entrada guardia de palabra>	::=	'ginword' '(' <chan>, <variable destino> ')'
<entrada guardia de mensaje>	::=	'ginmess' '(' <chan>, <mensaje destino> ')'
<salida de palabra>	::=	'outword' '(' <chan>, <variable origen> ')'
<salida de mensaje>	::=	'outmess' '(' <chan>, <tamaño>, <mensaje origen> ')'
<variable destino>	::=	<apuntador a enteros>
<mensaje destino>	::=	<apuntador a char>
<variable origen>	::=	<int>
<mensaje origen>	::=	<apuntador a char>
<tamaño>	::=	<int>

Modelo c.s.p.

<comando alternativo>	::=	[<comando protegido> {□<comando protegido>}]
<commando protegido>	::=	<guardia> <lista de commandos >
	|	(<rango> {, <rango>}) <guardia> <lista de comandos >
<guardia>	::=	<lista de guardias>
	|	<lista de guardias> ; <comando de entrada>
	|	<comando de entrada>
<elemento protegido>	::=	<expresion booleana> | <declaración>

Extensión al lenguaje C.

<comando alternativo>	::=	'altbegin' '{' { {<declaración de proceso alt>} | {<listadecomandos>} } 'altend' '}'
<declaración de proceso alt>	::=	'alitem' '(' <pid>, <guardia>, <pid>, <número de argumentos> {,<argumentos>} ')'
<guardia>	::=	<expresión booleana> | <entrada guardia>

Las principales correspondencias son las siguientes:

Modelo c.s.p.:		
<comando paralelo>	::=	[<proceso> { || <proceso>}]
Extensión al lenguaje C:		
<comando paralelo>	::=	'parbegin' '{' { {<declaración de proceso>} | {<lista de comandos>} | { 'seq' } } 'parend' '}'

Modelo c.s.p.:		
<proceso>	::=	<etiqueta del proceso> <lista de comandos>
Extensión al lenguaje C:		
<declaración de proceso>	::=	'paritem' '(' <pid>, <pid>, <número de argumentos> {, <argumentos> } ')'

Modelo c.s.p.:		
<comando de entrada>	::=	<origen> ? <variable>
Extensión al lenguaje C:		
<comando de entrada>	::=	<entrada de palabra>
	|	. .
<entrada de palabra>	::=	'inword' '(' <chan>, <variable destino> ')'

Modelo c.s.p.:		
<comando de salida>	::=	<destino> ! <expresión>
Extensión al lenguaje C:		
<comando de salida>	::=	<salida de palabra>
	|	.
<salida de palabra>	::=	'outword' '(' <chan>, <variable origen> ')'

Modelo c.s.p.:		
<comando alternativo>	::=	[<comando protegido> {□<comando protegido>}]
Extensión al lenguaje C:		
<comando alternativo>	::=	'altbegin' '{' { {<declaración de proceso alt>} | {<listadecomandos>} } 'altend' '}'

Ejemplos:

Modelo c.s.p.:
startend | getgive
Extensión al lenguaje C:

parbegin {
paritem (pid, pid_startend, code_startend, 0);
paritem (pid, pid_getgive, code_getgive, 0);
}

Modelo c.s.p.:
oeste?dato
Extensión al lenguaje C:
inword (canalesteoeste, &dato)

Modelo c.s.p.:
este!(resultado+1)
Extensión al lenguaje C:
outword (canalesteoeste, resultado+1)

Modelo c.s.p.:
$exchange_1$ □ $exchange_2$ □ $exchange_3$
Extensión al lenguaje C:

altbegin {
paritem (q1>q2, pid, pid_exchange_1, code_exchange, 0);
paritem (q2>q3, pid, pid_exchange_2, code_exchange, 0);
paritem (q3>q4, pid, pid_exchange_3, code_exchange, 0);
}

C.A.R. Hoare [1] indica que su trabajo ignora, entre otros, el problema de una eficiente implementación y que es probable que una solución a este problema requiera:

1) Imposición de restricciones en el uso de las características propuestas;
2) Re introducción de una notación diferente;
4) El diseño de un hardware apropiado.

Mi propuesta da solución al problema con las características que Hoare menciona, plasmando su modelo como una extensión al lenguaje C sobre una hardware específico, una topología hipercubo usando procesadores transputer.

II.3.2 Kernel

A continuación se mencionan algunas características que tiene KHiper.

- Se encarga de la colocación de los procesos sin intervención del programador. De manera que del algoritmo a la implementación no habrá mucha diferencia.

- Permite la comunicación de los procesos sin importar donde se encuentren, será KHiper quien se encargue de investigar en donde se encuentran para lograr la comunicación. El trabajo de mandar la información por entre los diferentes nodos de la red será transparente al programador.

- KHiper tiene la versatilidad de correrse en un hipercubo de orden 0, 1, 2 ó 3, por lo cual recibe la denominación de escalable.

- No es función de KHiper el evitar deadlocks en el programa del usuario. Esto es responsabilidad del programador.

- KHiper hace un eficiente distribución de procesos evitando que existan procesadores inactivos mientras existan procesos por ser ejecutados. Sin embargo, no en todos los casos KHiper reducirá el tiempo de ejecución de un programa, más aún, es posible que ese tiempo se incremente, debido a la sobrecarga ocasionada por las funciones de KHiper.

- Las facilidades que ofrece KHiper al usuario, aún cuando facilitan su labor, no le evitan ciertas molestias que pueden ser disminuidas por un precompilador. El desarrollo de este no es el objetivo de la tesis.

- Este trabajo da la plataforma para el futuro desarrollo de un sistema operativo y la implementación de kerneles para otras redes, ya sea de topologías diferentes o con otro tipo de procesadores.

II.3.3 Modelación del tráfico en la red

Para desarrollar los algoritmos de mapeo se necesita un modelo que permita analizar el tiempo que tardan en comunicarse los procesos. Así, utilizando al modelo se colocan los procesos de manera tal que se minimice el tiempo de comunicaciones del patrón.

A continuación planteo dos modelos de comportamiento del flujo de información dentro de una topología, para obtener información del rendimiento de un programa. Se pueden comparar diferentes mapeos para un mismo patrón en una topología particular y aún en topologías diferentes.

Definición. Una topología es un grafo, $T = (P, A)$, donde $P = \{p_0, p_1, p_2, ..., p_{n-1}\}$ es el conjunto de vértices, llamados procesadores y $A = \{a_0, a_1, a_2, ..., a_{m-1}\}$ es el conjunto de arcos, llamados conexiones, donde cada conexión $a_k = (p_i, p_j)$ es un par no ordenado de nodos. Podemos referirnos a la conexión que une al nodo p_i con el nodo p_j como a_{ij}.

Definición. Un hipercubo H de dimensión d, es un grafo, $H = (N, E)$, donde $N = \{n_0, n_1, n_2, ..., n_{\tilde{n}-1}\}$ es el conjunto de vértices, llamados nodos, siendo $\tilde{n} = 2^d$ y $d >= 0$, y $E = \{e_0, e_1, e_2, ..., e_{m-1}\}$, siendo $m = d\ 2^{d-1}$, es el conjunto de arcos, llamados enlaces, donde cada enlace $e_k = (p_i, p_j)$ es un par no ordenado de nodos, donde i y j, numerados en base dos, solo son diferentes en un bit.

Definición. Un patrón de comunicaciones R es un grafo, $R = (O, C)$, donde $O = \{o_0, o_1, o_2, ..., o_{q-1}\}$ es el conjunto de vértices, llamados procesos, siendo $q > 0$, y $C = \{c_0, c_1, c_2, ..., c_{r-1}\}$, siendo $r > 0$, es el conjunto de arcos, llamados canales, donde cada canal $c_k = (o_i, o_j)$ es un par no ordenado de procesos, tal que $i <> j$.

El concepto de patrón de comunicaciones es importante ya que cuando se diseña un programa concurrente se hace bajo esta idea.

Un patrón de comunicaciones se puede implantar en hardware, con lo cual se convierte en topología, o se pueden mapear sus procesos en los procesadores de una topología, que es lo más usual.

Por ejemplo considerese el siguiente patrón de comunicaciones, en forma de árbol

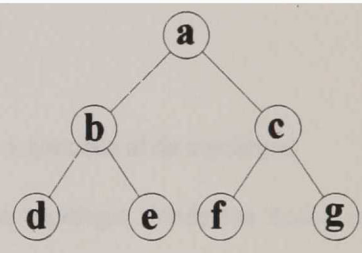


figura II.1

el cual se puede mapear en un par de procesadores

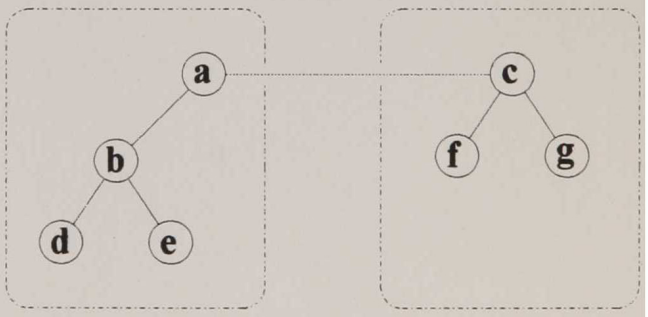


figura II.2

o sobre un hipercubo

Y aún cuando el conjunto de canales permanece, algunos de estos tienen que atravesar varios enlaces.

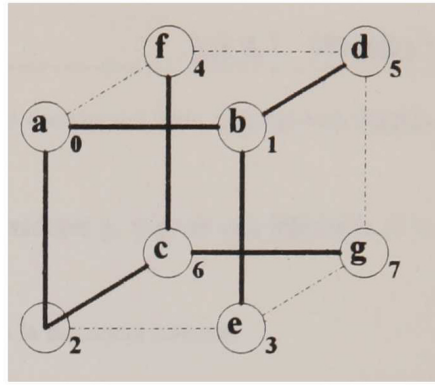


figura II.3

Considerese formar una topología en forma de árbol, de manera que cuando mapemos el patrón original se tendría una relación uno a uno en el mapeo.

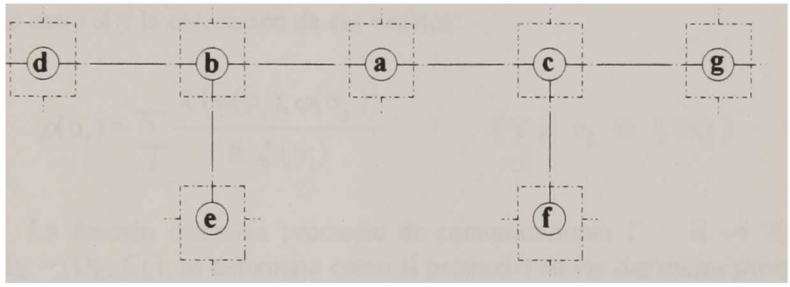


figura II.4

Como vemos el concepto de patrón de comunicaciones es más amplio y contiene al de topología.

Para la topología específica que estoy tratando, el hipercubo es una topología donde sus nodos son procesadores y sus enlaces son conexiones.

Este primer modelo considera las distancias entre procesos que se comunican para obtener una medida de la comunicación entre ellos.

Definición. La función distancia $\lambda : P \times P \rightarrow I$ entre los procesadores p_i y p_j de una topología es el número de enlaces entre los procesadores p_i y p_j.

A menos que especifique lo contrario me referiré como distancia a la distancia mínima.

Definición. La función vecindad $\xi : O \rightarrow 2^O$ para o_i es el conjunto de todos sus vértices adyacentes, denotado como $\xi(o_i) = \{o_{i0}, o_{i1}, o_{i2}, \ldots, o_{ik-1}\}$.

Definición. La función colocación $\varphi : O \rightarrow P$, indica en que nodo se coloca el proceso o_i.

Definición. La función distancia promedio $\rho : O \rightarrow \Re$, para el proceso o_i se determina como la distancia promedio entre él y la colocación de sus vecinos:

$$\rho(o_i) = \sum_j \frac{\lambda\left(\varphi(o_i), \varphi(o_j)\right)}{\# \xi(o_i)} \qquad \{ \forall j \mid o_j \in \xi(o_i) \} \qquad \text{ec.II.1}$$

Definición. La función distancia promedio de comunicaciones $\Gamma : R \rightarrow \Re$, para un patrón de comunicaciones $R_k = (O_k, C_k)$, se determina como el promedio de las distancias promedio de cada uno de los procesos en O_k:

$$\Gamma(R_k) = \frac{1}{2} \sum_i \frac{\rho(o_i)}{\#C} \qquad \{ \forall i \mid o_i \in O_k \} \qquad \text{ec.II.2}$$

Esta función será menor a uno sí la cantidad de procesos por procesador no es uniforme, en el caso extremo valdrá cero cuando todos los procesos se encuentren colocados en el mismo procesador.

De la teoría de grafos entendemos como vértices adyacentes a aquellos extremos de un arco.

Consideremos el patrón de comunicaciones de la siguiente figura.

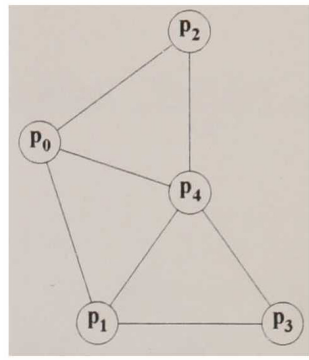


figura II.5

Las vecindades están definidas como:

$\xi(p_0) = \{ p_1, p_2, p_4 \}$
$\xi(p_1) = \{ p_0, p_3, p_4 \}$
$\xi(p_2) = \{ p_0, p_4 \}$
$\xi(p_3) = \{ p_1, p_4 \}$
$\xi(p_4) = \{ p_0, p_1, p_2, p_3 \}$

Consideremos un mapeo específico donde cada uno de los procesos es colocado sobre un hipercubo de orden 3 como en la sig. fig.

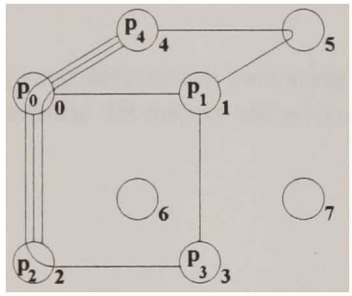


figura II.6

La función posición adquiere los siguientes valores:

$\varphi(p_0) = 0 \quad \varphi(p_1) = 1 \quad \varphi(p_2) = 2 \quad \varphi(p_3) = 3 \quad \varphi(p_4) = 4$

La función distancia:

$\lambda(j(p0), j(p_1)) = 1 \quad \lambda(\varphi(p_0), \varphi(p_2)) = 1 \quad \lambda(\varphi(p_0), \varphi(p_4)) = 1$
$\lambda(\varphi(p_1), \varphi(p_3)) = 1 \quad \lambda(\varphi(p_1), \varphi(p_4)) = 2 \quad \lambda(\varphi(p_2), \varphi(p_4)) = 2$
$\lambda(\varphi(p_3), \varphi(p_4)) = 3$

$$\rho(0) = \frac{\lambda(\varphi(p_0), \varphi(p_1)) + \lambda(\varphi(p_0), \varphi(p_2)) + \lambda(\varphi(p_0), \varphi(p_4))}{3} = \frac{1+1+1}{3} = 1$$

$$\rho(1) = \frac{\lambda(\varphi(p_1), \varphi(p_0)) + \lambda(\varphi(p_1), \varphi(p_3)) + \lambda(\varphi(p_1), \varphi(p_4))}{3} = \frac{1+1+2}{3} = 1.33$$

$$\rho(2) = \frac{\lambda(\varphi(p_2), \varphi(p_0)) + \lambda(\varphi(p_2), \varphi(p_4))}{2} = \frac{1+2}{2} = 1.5$$

$$\rho(3) = \frac{\lambda(\varphi(p_3), \varphi(p_1)) + \lambda(\varphi(p_3), \varphi(p_4))}{2} = \frac{1+3}{2} = 2$$

$$\rho(4) = \frac{\lambda(\varphi(p_4), \varphi(p_0)) + \lambda(\varphi(p_4), \varphi(p_1)) + \lambda(\varphi(p_4), \varphi(p_2)) + \lambda(\varphi(p_4), \varphi(p_3))}{4} = \frac{1+2+2+3}{4} = 2$$

$$\Gamma = \frac{1+1.33+1.5+2+2}{5} = 1.57$$

Ahora consideremos otro mapeo para el mismo patrón de comunicaciones sobre el mismo hipercubo, como vemos en la sig. fig.

Este mapeo nos da una distancia promedio Γ de 1.29 por lo cual es mejor que el anterior.

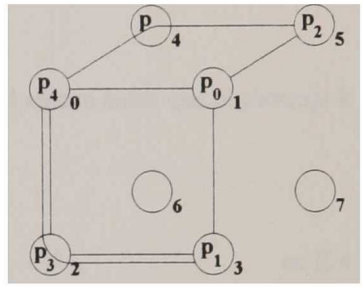


figura II.7

Así como se utilizaron estas definiciones para comparar dos mapeos para un patrón sobre una misma topología, así se podrían comparar dos mapeos sobre topologías diferentes.

II.3.3.2 Modelo II

El primer modelo considera sólo las distancias entre procesos que se comunican, sin considerar el tráfico en la red. Por lo cual se desarrollo un segundo modelo, el cual penaliza el desbalance en el uso de los enlaces, buscando que el sistema no se degrade por el uso excesivo de unos cuantos enlaces.

Para desarrollar este modelo consideremos dos grupos de n procesos:

$pa = \{pa_0, pa_1, \ldots, pa_{n-1}\}$ colocados en el nodo A y
$pb = \{pb_0, pb_1, \ldots, pb_{n-1}\}$ colocados en el nodo B

El proceso pa_k manda un mensaje al proceso pb_k, el tamaño del mensaje es de l_k bytes.
La velocidad a través del enlace que une a A con B es de v bytes/seg.

Ahora se plantea la pregunta ¿ Cuanto tiempo tardará en llegar el mensaje k a su destino ?

La solución a esta pregunta es la suma de los siguientes elementos:

- El tiempo que tarda el mensaje en atravesar el enlace.
  (s_k / v) segundos.

- El momento en que inicia el mensaje.

El segundo es una medida probabilística, la cual desarrollo a continuación:
El mensaje k debe esperar que 0 ó n-1 mensajes atraviesen el canal antes que él, en promedio debe esperar que ocurran (n - 1) / 2 mensajes previos. El tamaño promedio, dado que son diferentes, es

$$\frac{\sum_{i}^{n-1} \ell_i}{(n-1)} \qquad i \neq k \qquad \text{ec.II.3}$$

Por lo cual la cantidad de información promedio que atraviesa el enlace antes que el mensaje k es

$$\frac{(n-1)}{2} \times \frac{\sum_{i=0}^{n-1} \ell_i}{(n-1)} \qquad i \neq k \qquad \text{ec.II.4}$$

$$\sum_{i=0}^{n-1} \frac{\ell_i}{2} \qquad i \neq k \qquad \text{ec.II.5}$$

De lo anterior se concluye que el mensaje k tarda en atravesar el enlace en un tiempo promedio

$$\sum_{i=0}^{n-1} \frac{\ell_i}{2v} + \frac{\ell_k}{v} \qquad i \neq k \qquad \text{ec.II.6}$$

La velocidad es constante para todos los enlaces por lo cual se normalizo la ecuación anterior, la cual queda de la siguiente forma:

$$\sum_{i=0}^{n-1} \frac{\ell_i}{2} + \ell_k \qquad i \neq k \qquad \text{ec.II.7}$$

A esta cantidad se le llamo ***tiempo de entrega para el enlace*** a, denotada como te_a.

El canal i atraviesa los enlaces e_1, e_2, ... de manera que el tiempo que tarda en llegar el mensaje k a su destino es la suma de los tiempos que tarda en atravesar cada uno de ellos, a este tiempo promedio normalizado se le llamo ***tiempo de entrega para el canal*** i. Expresado en base a lo definido, será la suma de los tiempos de entrega para cada uno de los enlaces que atraviesa el canal i.

Lo que realmente se logra con esta expresión es **comparar** las cantidades de información que fluyen por la red, es decir se obtienen **relaciones** entre los tamaños de los mensajes.

Como ejemplo de como se utilizan las definiciones anteriores en la colocación de un proceso consideremos el siguiente patrón que está siendo mapeado a un hipercubo d-3. Este forma parte de uno más amplio, el cual no se muestra.

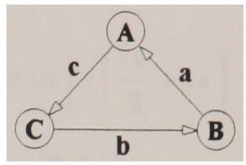


figura II.8

De ese patrón ya han sido mapeados A y B, localizados en los nodos 0 y 3 respectivamente, como vemos en la figura de la derecha.

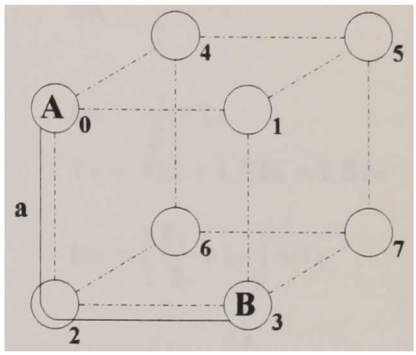


figura II.9

Sólo falta colocar C.

Parece obvio que el lugar ideal es el nodo 2, la cuestión es saber si ese es realmente el más adecuado. Vemos esta posibilidad a continuación.

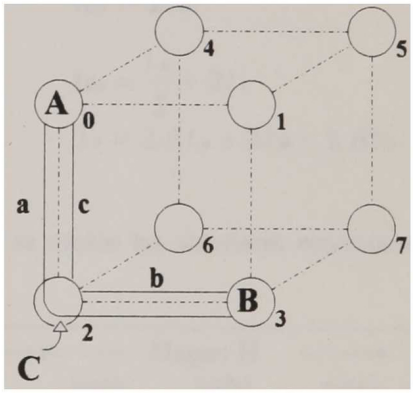


figura II.10

Se observa que los enlaces e03 y e32 son los que soportan todo el tráfico de comunicaciones. Se necesita determinar sí colocando a C en otro nodo, el tráfico fuera menos denso sin que el tiempo que tardan las comunicaciones sea mayor.

Se elije el nodo 5 que no está lejano a A y B y despeja a e03 y e32.

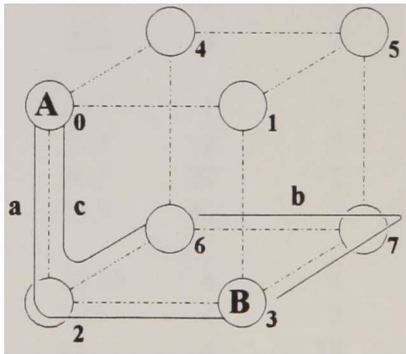


figura II.11

Utilizando las ecuaciones ec.II.6 y ec.II.7, muestro los tiempos de entrega de cada canal.

a) Para el mapeo I:

$$te_a = \left(\frac{l_c}{2} + l_a\right) + \left(\frac{l_b}{2} + l_a\right)$$

$$te_a = 2l_a + \frac{l_b + l_c}{2}$$

$$te_b = \frac{l_a}{2} + l_b$$

$$te_c = \frac{l_a}{2} + l_c$$

$$T_r = 3l_a + 1.5l_b + 1.5l_c$$

b) Para el mapeo II:

$$te_a = \left(\frac{l_c}{2} + l_a\right) + l_a$$

$$te_a = 2l_a + \frac{l_c}{2}$$

$$te_b = 2l_b$$

$$te_c = \frac{l_a}{2} + 2l_c$$

$$T_r = 2.5l_a + 2l_b + 2.5l_c$$

Para observar como se comportan estos tiempos de entrega se evalúo las anteriores ecuaciones con diferentes conjuntos de valores para tamaños de mensaje.

				← MapeoI		→	← Mapeo II		→
No.	**la**	**lb**	**lc**	**te(a)**	**te(b)**	**te(c)**	**te(a)**	**te(b)**	**te(c)**
1	3	2	1	7.5	3.5	2.5	6.5	4	3.5
2	4	2	1	9.5	4	3	8.5	4	4
3	5	2	1	11.5	4.5	3.5	10.5	4	4.5
4	10	2	1	21.5	7	6	20.5	4	7
5	10	0.5	1	20.75	5.5	6	20.5	1	7
6	10	1	1	21	6	6	20.5	2	7
7	10	10	1	25.5	15	6	20.5	20	7
8	10	20	1	30.5	25	6	20.5	40	7
9	10	2	0.5	21.25	7	5.5	20.25	4	6
10	10	2	1	21.5	7	6	20.5	4	7
11	10	2	10	26	7	15	25	4	25
12	10	2	20	31	7	25	30	4	45

tabla 1

Si se mantienen constantes l_b y l_c en tanto aumenta l_a el tiempo de entrega del mapeo II es mejor que el del mapeo I, renglones 1 al 4.

En cambio, conforme l_b crece con respecto a l_a y l_c el tiempo de entrega va disminuyendo. Similar a cuando l_c crece.

El planteamiento anterior considera condiciones extremas, cuando todos los procesos mandan simultáneamente sus mensajes.

El exceso de tráfico sobre algunos enlaces puede degradar el sistema. Con este modelo esa degradación puede ser considerada ya que el tiempo de entrega es una medida de la eficiencia de la red. Tenemos ahora una medida que relaciona el tráfico y la velocidad de los mensajes sobre la red.

Sintetizando: ***Para lograr un mapeo efectivo se debe minimizar el tiempo de entrega.***

A continuación se formaliza lo expresado anteriormente.

Definición. La función Ec: $C \rightarrow 2^E$, es el conjunto de enlaces que un canal atraviesa cuando es mapeado.

Definición. La función Ce: $E \rightarrow 2^C$, es el conjunto de canales mapeados que atraviesan al enlace e_k.

Por ejemplo para la siguiente figura:

Ec (a)= $\{e_{02}, e_{23}\}$
Ec (b)= $\{e_{67}, e_{73}\}$
Ec (c)= $\{e_{62}, e_{20}\}$

Ce (e_{02})= { a, c}
Ce (e_{23})= { a }
Ce (e_{26})= { c }
Ce (e_{67})= { b }
Ce (e_{73})= { b }

Definición. La función caudal l : $C \rightarrow \Re$ indica, en forma normalizada, la cantidad de comunicaciones que va a efectuar un canal.

Para la entrada 3 de la tabla presentada

l (a)= 5; l (b)= 2; l (c)= 1;

Definición. La función caudal restante lr : $C \rightarrow \Re$ para el canal ck sobre el enlace ei es la cantidad de información que atravesará el enlace e_i por parte de todos los canales que están mapeados a él excepto el canal ck, es decir la información de todos los canales restantes a ck.

$$\mathbf{lr(ck,\ ei)} = \sum_{i} \mathbf{l(c_i)} \qquad \{\forall\, i \mid c_i \in Ce\,(e_k) \wedge c_i \neq c_k\} \qquad \text{(ecII.8)}$$

Definición. La función caudal restante promedio lrp : $C \rightarrow \Re$, es la cantidad promedio de información que atravesará el enlace e_i antes que la información del canal c_k.

$$\mathbf{lrp(ck,\ ei) = \frac{lr(ck,\ ei)}{2}} \qquad \text{(ecII.9)}$$

Para el mapeo de la figura al lado obtenemos las siguientes ecuaciones para la función caudal restante y función caudal restante promedio para el canal c_a a través del enlace 01

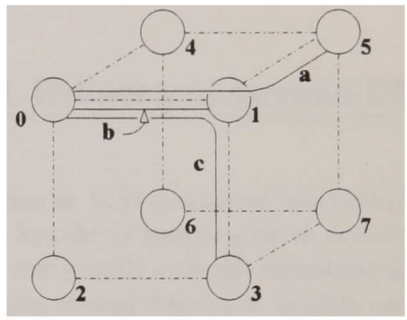


figura II.12

$$\mathbf{lr(c_a,\ e_{01}) = \sum (l(c_b), l(c_c))}$$

$$\mathbf{lrp(c_a,\ e_{01}) = \frac{l(c_b) + l(c_c)}{2}}$$

Definición. El tiempo de entrega te: $C \rightarrow \Re$, para el canal c_j. Esta función está normalizada, por lo que aún cuando no aparece la variable velocidad, dimensionalmente es correcta.

$$\mathbf{te(ck) = \sum_i lrp(ck,\ ei) + \#\left(Ec\left(c_k\right)\right) l(ck)} \qquad \{\forall\, i \mid e_i \in Ec\,(c_k)\} \qquad \text{(ecII.10)}$$

Para la figura anterior tenemos que para el canal ca el tiempo de entrega es

$$\mathbf{te(c_a) = \frac{l(c_b) + l(c_c)}{4} + 2l(c_a)}$$

Definición. El tiempo de entrega tr: $R \rightarrow \Re$, para el patrón de comunicaciones R es

$$\mathbf{tr(R) = \frac{\sum_i te(c_i)}{\#C}} \qquad \{\forall\, i \mid c_i \in C\} \qquad \text{(ec.II.11)}$$

Utilizando la figura previa, el tiempo de entrega para el patrón de comunicaciones R es

$$\mathbf{Tr(R) = \left(\frac{l(c_b) + l(c_c)}{4} + 2l(c_a)\right) + \left(\frac{l(c_a) + l(c_b)}{4} + 2l(c_c)\right) + \left(\frac{l(c_a) + l(c_c)}{4} + l(c_b)\right)}$$

Reduciendo obtenemos

$$\mathbf{Tr(R) = 2.5l(c_a) + 2l(c_b) + 2.5l(c_c)}$$

En la definición de la función caudal restante promedio el factor 2 en el denominador es una cantidad que podemos evaluar estadísticamente, midiendo los tiempos de ejecución de una serie de programas. Esta puede ser una constante o tal vez una función.

Las definiciones que se plantean son independientes de la topología, por lo cual se pueden aplicar a diferentes topologías.

II.3.4. KHiper y el modelo OSI

El modelo de Referencia para la Interconexión de Sistemas Abiertos de la Organización Internacional de Estándares (ISO OSI, Day y Zimmerman 1983) es importante en su función de comparación de modelos. Sirve para redes de computadoras, es decir computación distribuida. Aún cuando podemos considerar que la computación paralela es un caso particular de los sistemas distribuidos existen diferencias amplias entre ambos.

Específicamente en mi trabajo estoy tratando con un conjunto de procesadores idénticos, los cuales se encuentran a una distancia muy corta entre ellos. Además todos los procesadores trabajan en la solución de un sólo problema. Lo cual elimina algunos de los requerimientos del modelo OSI.

A continuación KHiper y el modelo OSI.

- Para Khiper el modelo OSI es útil únicamente para referenciarlo con la capa de comunicaciones abstractas.
- La capa física se refiere a las interfases eléctricas y mecánicas, a la transmisión más simple de bits. Esto se logra usando los enlaces que proporciona el transputer.
- La capa de enlace de datos se encarga de proporcionar una línea libre de errores. En el caso de KHiper no es necesaria, dado que los procesadores se encuentran en el mismo gabinete. Para que existan errores la distancia entre los procesadores debe exceder los 60 cm.
- Resolver la diferencia entre velocidades de transmisión de los procesadores es otro punto que debe resolver la capa de enlace, Khiper no la necesita.
- Finalmente, en sistemas distribuidos esta capa debe resolver el problema cuando la comunicación es bidireccional. Dado que la comunicación que ofrecen los enlaces del transputer son completamente bidireccional (full duplex), no existen conflictos que resolver.
- Las funciones que ofrece la capa de comunicaciones abstractas de KHiper abarca las que en OSI deben ofrecer las capa de red y de transporte.
- KHiper como capa de red se encarga de rutear la información de un nodo a otro.
- Los canales que ofrece KHiper ofrecen una conexión extremo a extremo, como deben ofrecer las comunicaciones de la capa de transporte.

- La sincronización y manejo de tokens que debe ofrecer la capa de sesión no son necesarios dado que KHiper es mono usuario, corriendo sobre una topología conocida.

- En este trabajo las capas superiores de O.S.I. no tienen sentido, dado que los procesadores involucrados son idénticos.

A continuación se puede observar como se relacionan KHiper y OSI.

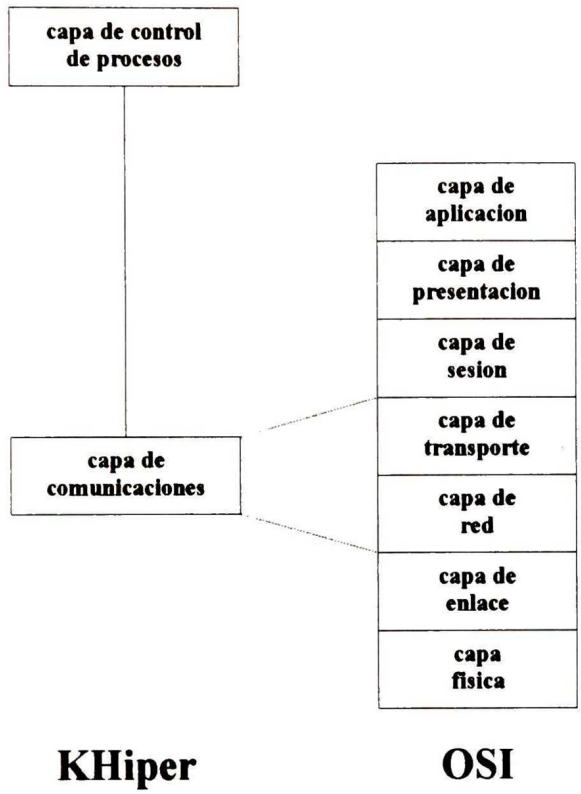


figura II.13

3 VISTA DEL USUARIO

III.1 Descripción

El sistema consiste de varias capas de software. La capa hardware, la capa de comunicaciones abstractas, y la capa de control de procesos.

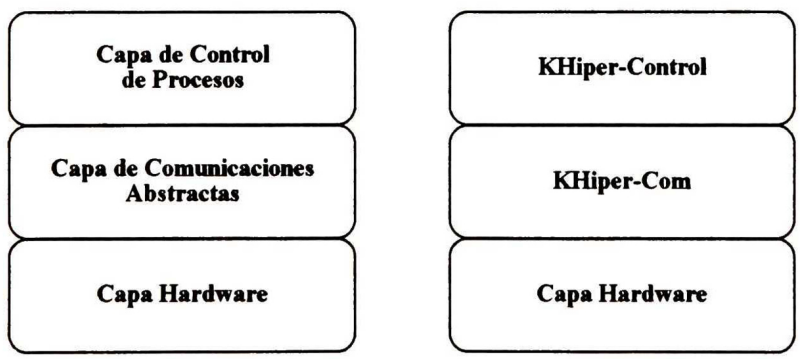


figura III.1

Se consideran dentro de la capa hardware a los procesadores, a las funciones de comunicación que ofrece el compilador, dado que su alcance es limitado, y a los archivos de configuración.

La capa de comunicaciones abstractas proporciona funciones de acuerdo al modelo c.s.p. Su alcance se extiende a cualquier procesador. Los canales que ofrece esta capa se les llamo virtuales.

En otros modelos de kernel una capa únicamente hace uso de las funciones proporcionadas por la capa inmediata inferior. Sin embargo, dado que el modelo c.s.p. funciona por medio de paso de mensajes las funciones que ofrece la capa de comunicaciones las utiliza tanto la capa de control como el usuario.

La capa de control de procesos está diseñada para

- Soportar las construcciones par y alt.
- Buscar un mapeo óptimo para la construcción par.
- Ocultar la arquitectura al usuario.
- Dar facilidades para mandar información a la pantalla y recibir información de teclado.

Este trabajo tiene dos alternativas de uso.

- Utilizar únicamente la capa de comunicaciones.
  Cuando los programas son simples y no requieren las construcciones alt y par.
  A este caso se le llamó utilizar «KHiper-Com».

- Usar la segunda y tercer capas cuando se requiere usar el modelo c.s.p.
  A este se le llamó, por comodidad, utilizar «KHiper-Control», ya que va implícito el uso de la capa de comunicaciones, KHiper-Com.

III.2 Uso de la capa de control de procesos

III.2.1 Construcción Par

A continuación se muestra un programa usando todas las capas de KHiper.

```
1   #include "hkernel.h"
2   proc Main (Pid pid) {
3
4           Pid procA, procB, procC, procD;
5           tscr screen;
6
7    newpids (pid, 4, &procA, &procB, &procC, &procD);
8
9    parbegin
10   {
11    paritem (pid, procB, 2, 0);
12    paritem (pid, procC, 2, 0);
13    paritem (pid, procD, 2, 0);
14   } parend;
15   toscreen ((SCR, "proceso %3d con codigo 1 ejecutado por (%d)\n",pid, node));
16
17   end ();
18  }
```

```
19
20  proc code2 (PId pId) {
21
22          tscr screen;
23
24   toscreen((SCR, "proceso %3d con codigo 2 ejecutado por (%d)\n", pId, node));
25   end ();
26  }
27
28  initCodes ( ) {
29
30   initcodes (2, Main, code2);
31  }
```

Todos los códigos deben ser previamente identificados mediante la función initCodes( ), esta es llamada por el sistema para formar una tabla en cada procesador.

Es obligación del usuario definir initCodes( ). La cual llama a initcodes.
El primer parámetro indica cuantos códigos van a ser declarados y a continuación el nombre de cada uno de ellos. La posición de un nombre nos da su índice.

En el programa de ejemplo el usuario define la función initCodes en la línea 28. En initcodes, línea 30, el usuario indica que son 2 códigos, Main y code2. Entonces, el índice de Main es 1 y el de code2 es 2.

A partir de ese momento el usuario se referirá a los códigos por su índice, siendo su responsabilidad el uso correcto de ellos. Es recomendable usar macros define.

Si existe un error al definir initCodes, o si se usan incorrectamente los índices, pueden haber fallas, como crear un proceso no deseado, con parámetros extraños, o que el sistema se estanque, en caso de usar un índice no declarado.

Un proceso le informa al sistema cuando llega al fin de sus actividades mediante la función end( ).

III.2.2 Como funciona KHiper

La primera acción de KHiper es crear el primer proceso del usuario, cuyo índice es uno. Después se dedica a esperar solicitudes. Las primeras provienen de ese primer proceso, lo que lo convierte en el proceso principal a la manera de la función main de C. Para realzar este aspecto en el ejemplo lo llamé Main.

Para que un proceso pueda solicitar servicios a KHiper debe tener un identificador, el cual se le pasa como parámetro. Al primer proceso es el único a quien Khiper le da identificador.

Un proceso antes de crear procesos solicita a Khiper sus identificadores, mediante la función newpids, línea 7. Los identificadores son del tipo PId, línea 5.

Las palabras reservadas parbegin y parend delimitan el cuerpo de la construcción.

Los procesos componentes se indican usando paritem (...), los parámetros son: el identificador del padre, el identificador del hijo, el número de código que se va a usar, el número de parámetros y finalmente la lista de estos. En la siguiente línea se va a instanciar al proceso con identificador procB, código 2 y 0 argumentos.

```
11    paritem (pId, procB, 2, 0);
```

El primer parámetro del código que usará un proceso es el identificador, el cual debe llamarse 'pId'. Todas funciones del kernel requieren este valor. Por comodidad se definieron una serie de macros que le pasan este valor a la función. Es por esta razón la restricción en cuanto al nombre del identificador.

El número de parámetros al que se refiere la función paritem no considera la existencia de pId. De manera que el número mínimo de parámetros formales que se debe declarar es uno.

De las líneas 12 a 14 tenemos la declaración de tres procesos con identificadores procB, procC y procD. Los tres van a ejecutar el mismo código, 'code2'. Después del índice se encuentra el valor 0 indicando que los procesos no tienen parámetros.

El tipo que regresa una función que defina el código de un proceso debe ser void. Existe un macro que substituye el texto proc por void.

KHiper al encontrar el parend busca los procesadores con menor carga de trabajo y le indica a cada uno de ellos que ejecute uno de los procesos. Hará esperar al proceso padre a que todos los procesos hijos terminen, cuando esto suceda lo recalendarisará.

Gráficamente se puede ver la construcción par de las líneas 12 a 14 como muestro a continuación.

De acuerdo a II.4 no se debe confundir este grafo con un árbol de activaciones. La gráfica anterior está indicando que Main da origen simultáneo a proB, proC y proD.

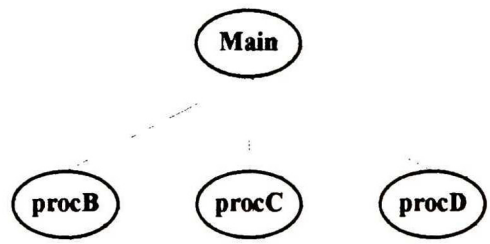


figura III.2

Sólo cuando se alcanza el final de la construcción par los procesos paralelos empiezan a ejecutarse.

Dicho de otra manera, el código indicado dentro de la construcción par se ejecuta antes que los procesos indicados por las funciones paritem.

Resaltando, el árbol de creaciones para el programa anterior sería el siguiente.

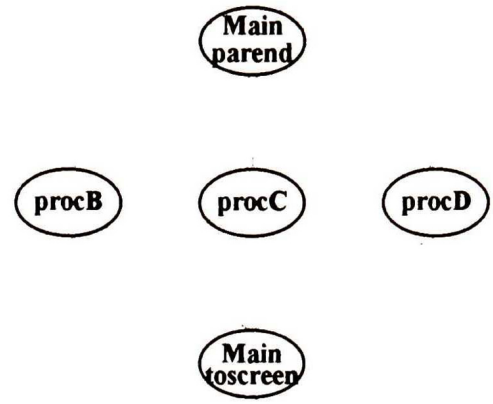


figura III.3

Los procesos constituyentes se ejecutan hasta encontrar el parend, esto no significa que los parámetros se actualicen en ese momento. Revisese el siguiente fragmento.

```
 8    dato1= 10;
 9    dato2= 20;
10    parbegin {
11       paritem (pId, procA, CodeA, 1, dato1);
12       paritem (pId, procB, CodeB, 1, dato2);
13       dato1= 1000;
14       dato2= 2000;
15    } parend;
```

El proceso procA va a ejecutarse con un parámetro de 10 no de 1000.

Todos los hijos ejecutan el mismo código. Lo que realiza cada uno de ellos es simple, mandar a la pantalla su identificador y el procesador que lo está ejecutando, utilizando el macro toscreen, línea 24.

El uso del macro toscreen requiere de la previa declaración de la variable tscr screen. Este programa lo puede encontrar en el directorio \khiper\ejemplos con el nombre kerx02.c.

III.2.3 Paso de parámetros

A continuación se muestra otro programa en el cual se observan características adicionales del kernel.

```
1  /*        KER X03 .C                    */
2  #include "hkernel.h"
3  #define FIRST 5
4  #define INC 4
5  enum {Code1=1, Code2, Code3, Code4, Code5);
6
7  proc code1 (PId pId) {
8            PId  procA, procB, procC, procD;
9            chari, deepth,san [50];
10           tscr screen;
```

```
11
12  deepth= FIRST;
13  newpids (PRO, 4, &A, &B, &C, &D);
14  parbegin
15  { paritem (pid, A, Code2, 1, deepth+ INC);
16   paritem (pid, B, Code3, 1, deepth+ INC);
17   paritem (pid, C, Code3, 1, deepth+ INC);
18   paritem (pid, D, Code3, 1, deepth+ INC);
19  } parend;
20  tab (deepth, san);
21  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, node));
22  end ();
23 }
24
25 proc code2 (Pid pid, Small deepth) {
26          Pid E, F;
27          chari, san [50];
28          tscr screen;
29
30  params (&deepth);
31  newpids (pid, 2, &E, &F);
32  parbegin
33  { paritem (pid, E, Code4, 1, deepth+ INC);
34   paritem (pid, F, Code5, 1, deepth+ INC);
35  } parend;
36  tab (deepth, san);
37  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, node));
38  end ();
39 }
40
41 proc code3 (Pid pid, Small deepth) {
42          chari, san [50];
43          tscr screen;
44
45  params (&deepth);
46  tab (deepth, san);
47  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, node));
48  end ();
49 }
50
51 proc code4 (Pid pid, Small deepth) {
52          chari, san [50];
53          tscr screen;
54
55  params (&deepth);
56  tab (deepth, san);
57  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, node));
58  end ();
59 }
60
61 proc code5 (Pid pid, Small deepth) {
62          Pid G, H, I;
63          chari, san [50];
64          tscr screen;
65
66  params (&deepth);
```

```
67  newpids (pId, 3, &G, &H, &I);
68  parbegin
69    {     paritem (pId, G,   Code3, 1, deepth+ INC);
70          paritem (pId, H,   Code3, 1, deepth+ INC);
71          paritem (pId, I,   Code3, 1, deepth+ INC);
72  } parend;
73  tab (deepth, san);
74  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pId, node));
75  end ();
76 }
77
78 tab (int deepth, char* san) {
79            int i;
80  for (i= 0; i < deepth; i++)
81       san [i]= ' ';
82  san [i]= '\0';
83 }
84
85 initCodes ( )
86 { initcodes (6, code1, code2, code3, code4, code5, code6);
87 }
88
```

La primera característica adicional que se encuentra es en el paritem, líneas 15 a 18.

Después del índice code2 se indica el número de parámetros, en este caso uno, el cual es la variable deepth incrementada en INC unidades.

La segunda característica aparece en la línea 30.

Cuando KHiper crea un proceso no conoce donde se van a almacenar sus parámetros, por lo que es necesario que sea el proceso quien los solicite mediante la función params.

El parámetro de params es la dirección del primer parámetro del proceso posterior a pId.

A continuación se muestra el árbol de creaciones correspondiente.

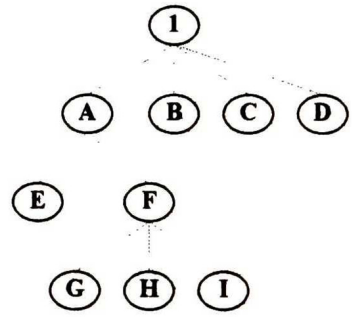


figura III.4

III.2.4 Colocación en procesadores específicos

Continuemos con un programa que permite ajustar el programa a la topología subyacente.

```
1  /*              KER X08.C                    */
2  #include "hkernel.h"
3  #define INC 4
4  #define Code2 2
5
6  proc code1 (Pid pid) {
7            Pid pA, pB, pC, pD, pE, pF, pG, pH;
8            chari, deepth, san [50];
9            tscr screen;
10
11  newpids (pid, 8, &pA, &pB, &pC, &pD);
12  parbegin {
13   paritem (pid, pA, Code2, 1, INC);
14   paritem (pid, pB, Code2, 1, INC);
15   paritem (pid, pC, Code2, 1, INC);
16   paritem (pid, pD, Code2, 1, INC);
17   placeat (pA, 0); placeat (pB, 1);   placeat (pC, 2);   placeat (pD, 3);
18  } parend;
19
20  tab (deepth, san);
21  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, node));
22  end ();
23 }
24
25 proc code2 (Pid pid, Small deepth) {
26           chari, san [50];
27           tscr screen;
28
29  params (&deepth);
30  tab (deepth, san);
31  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, node));
32
33  end ();
34 }
35
36 tab (int deepth, char* san) {
37           int i;
38  for (i= 0; i < deepth; i++)
39       san [i]= ' ';
40  san [i]= '\0';
41 }
42
43 initCodes ()
44 { initcodes (2, code1, code2);
45 }
```

Sí a pesar de perder portabilidad, se desea mapear en procesadores específicos se puede efectuar mediante la función placeat. Se le encuentra en la línea 17.

El primer parámetro indica que proceso se va a colocar y el segundo en donde. Sí el usuario proporciona el número de un procesador que no existe puede bloquear el sistema. Lo más adecuado es que inserte código que verifique la dimensión.

De manera que los procesos quedan colocados en el hipercubo como a continuación se muestra.

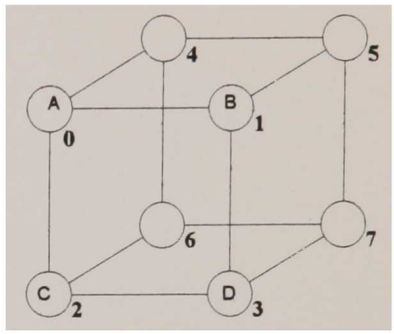


figura III.5

III.2.5 Patrón de Comunicaciones y Mapeo

El patrón de comunicaciones le informa a KHiper como se relacionan los procesos para que pueda calcular su colocación, basado en el rendimiento.

En el Modelo I, de la sección II.3.4.2, se encuentra la definición de vecindad.

Las ξ-vecindades del patrón de la siguiente figura son:

$x(a) = \{ b, c, e \}$
$x(b) = \{ a, d, e \}$
$x(c) = \{ a, e \}$
$x(d) = \{ b, e \}$
$x(e) = \{ a, b, c, d \}$

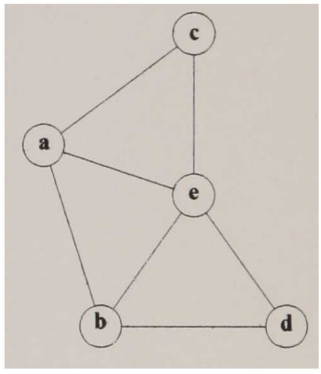


figura III.6

Se muestra ahora un programa que hace uso del anterior patrón de comunicaciones.

```
1  /*              KER X11.C        */
2  #include "hkernel.h"
3  #define FIRST 5
4  #define INC 4
5  enum {Code1=1, Code2};
6
7  proc code1 (Pid pid) {
8          Pid  a, b, c, d, e, f, g, h;
9          char  i, deepth, san [50];
10         tscr  screen;
11
12  deepth= FIRST;
```

```
13  newpids (pId, 8, &a, &b, &c, &d, &e, &f, &g, &h);
14  parbegin {
15   paritem (pId, a, Code2, 1, deepth+ INC);
16   paritem (pId, b, Code2, 1, deepth+ INC);
17   paritem (pId, c, Code2, 1, deepth+ INC);
18   paritem (pId, d, Code2, 1, deepth+ INC);
19   paritem (pId, e, Code2, 1, deepth+ INC);
20   neigh (pId, a, 3, c, b, e);
21   neigh (pId, b, 3, a, d, e);
22   neigh (pId, c, 2, a, e);
23   neigh (pId, d, 2, b, e);
24   neigh (pId, e, 4, a, b, c, d);
25  } parend;
26
27  tab (deepth, san);
28  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pId(), node));
29  end ();
30 }
31
32 proc code2 (PId pId, Small deepth)
33 {
34            char  i, san [50];
35            tscr  screen;
36
37  params (&deepth);
38  tab (deepth, san);
39  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pId(), node));
40  end ();
41 }
42
43 tab (int deepth, char* san) {
44            int i;
45  for (i= 0; i < deepth; i++)
46       san [i]= ' ';
47  san [i]= '\0';
48 }
49 .
50 initCodes ( ) {
51  initcodes (2, code1, code2);
52 }
```

De la línea 20 a 24 se encuentra a neigh que es la contraparte de la función ξ-vecindad. Su primer parámetro es el pId del proceso, el segundo la cardinalidad de la ξ-vecindad, y a continuación los identificadores que forman dicha vecindad.

En este programa el patrón de comunicaciones es simplemente ilustrativo, puesto que los procesos no se comunican entre sí.

III.2.6 Canales

KHiper comunica procesos no importa donde se encuentren. La comunicación se establece por medio de canales que se les llamo ***virtuales***, dado que son aparentes.

Se necesitan dos canales extremos para formar uno virtual, en el origen y en el destino. Uno apunta al otro. Este direccionamiento lo especifica el proceso indicando a que procesador y a que número de canal extremo apunta. ***Las funciones de comunicación utilizan sólo canales extremos.***

Los canales se deben declarar, los extremos de tipo EChan, los virtuales VChan.

En KHiper-Control los canales virtuales son un recurso del sistema, adquieren su valor cuando lo solicitan mediante la función getchan. El parámetro de esta es la dirección de la variable tipo VChan.

Cuando se va a requerir comunicación entre un par de procesos que aún no se han creado, el proceso padre solicita un canal virtual y pasa su valor como parámetro a las funciones que declaran a los futuros hijos. Los hijos inicializan sus canales extremos en base a este, mediante chaninit.

El primer parámetro de chaninit es el canal extremo que forma al virtual, el cual es el segundo parámetro.

La función para mandar un word es outword, inword para recibir. El primer parámetro de outword es el canal extremo y el segundo es el valor que manda. El primero de inword es el canal extremo y el segundo la dirección de la variable que va a recibir.

El siguiente programa está basado en el patrón de al lado, donde en cada vértice se encuentra un proceso que recibe y manda un dato como se observa en la figura.

Está representado por las funciones siguientes:

$\xi(a) = \{ b, e \}$
$\xi(b) = \{ a, c \}$
$\xi(c) = \{ b, d \}$
$\xi(d) = \{ c, e \}$
$\xi(e) = \{ d, a \}$

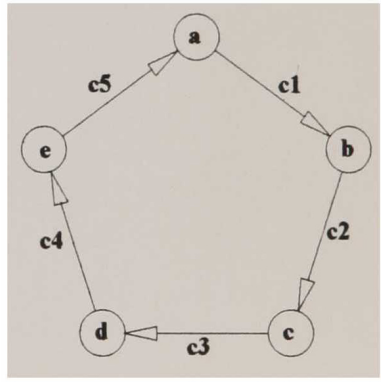


figura III.7

```
1  /*              KER X19.C              */
2  #include "hkernel.h"
3  enum codes {Main=1, Start_end, Get_give};
4
5  proc Main (PRS) {
6          Pid  a, b, c, d, e;
7          Vchan  c1, c2, c3, c4, c5;
8          tscr  screen;
9
10 newpids (PRO, 5, &a, &b, &c, &d, &e);
```

```
11  parbegin {
12   getchan (&c1); getchan (&c2); getchan (&c3); getchan (&c4); getchan  (&c5);
13
14      paritem (PRO, a, Start_end,      3, c5, c1, 'a');
15      paritem (PRO, b, Get_give,       3, c1, c2, 'b');
16      paritem (PRO, c, Get_give,       3, c2, c3, 'c');
17      paritem (PRO, d, Get_give,       3, c3, c4, 'd');
18      paritem (PRO, e, Get_give,       3, c4, c5, 'e');
19
20   neigh (PRO, a, 2, e, b);  neigh (PRO, b, 2, a, c);
21   neigh (PRO, c, 2, b, d);  neigh (PRO, d, 2, c, e);  neigh (PRO, e, 2, d, a);
22  } parend;
23
24  toscreen ((SCR, "\n\nproceso %d ejecutado por (%d)\n", pId, node));
25  end ();
26 }
27
28 proc start_end (PRS, Vchan IN, Vchan OUT, char letra) {
29            int  dato;
30            tscr  screen;
31            Echan  in, out;
32
33  params (&IN);
34  chaninit (&in,  IN);
35  chaninit (&out, OUT);
36
37  outword (out, 1);
38  inword  (in, &dato);
39  toscreen ((SCR, "proceso %c en %d con dato == %d\n", letra, node, dato));
40  end ();
41 }
42
43 proc get_give (PRS, Vchan IN, Vchan OUT, char letra) {
44            int  dato;
45            tscr  screen;
46            Echan  in, out;
47
48  params (&IN);
49  chaninit (&in,  IN);
50  chaninit (&out, OUT);
51
52  inword  (in, &dato);
53  outword (out, dato+ 1);
54  toscreen ((SCR, "proceso %c en %d con dato == %d\n", letra, node, dato));
55  end ();
56 }
57
58 initCodes ( ) {
59  initcodes (3, Main, start_end, get_give);
60 }
```

Todos los procesos son iguales excepto start_end que es quien inicia y termina el ciclo, en los restantes vértices se encuentra a get_give. Los crea Main.

Se requieren 5 canales virtuales, Main los declara en la línea 7 y solicita su valor por medio de la función getchan, línea 12.

En la línea 28 se encuentra el código de start_end. Su primer parámetro es el canal virtual que necesita para recibir, el segundo para mandar, c5 y c1 respectivamente. El tercero es una letra que lo va a identificar en pantalla.

Los parámetros de get_give son iguales a los de start_end.

En la línea 34 start_end inicializa a in para ser extremo de IN. En la 35 a out.

start_end inicia la comunicación mandando un '1' al proceso de su izquierda a través de out.

Los procesos b, c, d, y e esperan hasta recibir un word de su derecha a través de in, línea 52. Después lo manda, incrementado en 1, a la izquierda por medio de out. Al final manda a pantalla una leyenda.

El grafo de activaciones para este programa es el siguiente.

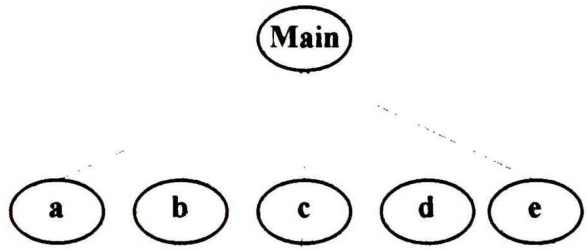


figura III.8

III.2.7 Arreglos de identificadores y de canales

A continuación se comenta un par de funciones útiles cuando se realizan múltiples instancias de un código para generar un grupo de procesos. En el programa anterior nos dimos cuenta que los códigos de los dos hijos básicamente son idénticos, de manera que podemos sintetizarlos.

En este programa se observa el patrón del anterior pero a diferencia de aquel este entra en un ciclo de comunicaciones infinito.

```
1  /*              KER X23.C       */
2  #include "hkernel.h"
3  #define yes 1
4
5  proc Main (Pid pid) {
6          int  i;
7          Pid  soon [5];
8          Vchan  chan [5];
```

```
9               tscr  screen;
10
11  newpidarray  (5, soon);
12  getchanarray (5, chan);
13  parbegin
14  { for (i= 0; i < 5; i++)
15    paritem (pId, soon [i], 2, 4, chan [(i+4)%5], chan [i], !i%5, 'a'+i);
16
17   for (i= 0; i < 5; i++)
18    neigh (pId, soon [i], 2, soon [(i+4)%5], soon [(i+1)%5]);
19
20   bestmap ();
21  } parend;
22 }
23
24 proc cycleitem (PId pId, Vchan In, Vchan Out, Small start, char letter) {
25              int  dato;
26              Echan  in, out;
27              tscr  screen;
28
29  params (&In);
30  chaninit (&in,  In);
31  chaninit (&out, Out);
32
33  if (start == yes) outword (out, 1);
34  while (1) {
35   inword  (in, &dato);
36   outword (out, dato+ 1);
37   toscreen ((SCR, "proceso %c en %d con dato == %d\n", letter, node, dato));
38  }
39 }
40
41 initCodes ( ) {
42  initcodes (2, Main, cycleitem);
43 }
```

Una facilidad que proporciona KHiper es la de solicitar identificadores en un arreglo usando la función newpidarray. En la línea 11 newpidarray solicita 5 identificadores en el arreglo 'soon'.

De manera similar en vez de utilizar la función getchan se puede utilizar la función getchans, línea 12.

En la línea 12 se encuentra la manera de solicitar un arreglo de 5 canales virtuales. ***Manejando los canales mediante arreglos podemos invocar a paritem como cuerpo de un ciclo for***, líneas 14 y 15.

III.2.8 Obtención del mejor mapeo

KHiper logra un mapeo óptimo, no siempre el mejor.

El algoritmo de mapeo inicia determinando la colocación de un primer proceso, el resultado puede ser diferente si ese primer proceso es otro.

Para obtener el mejor mapeo lo se calcula tomando como primer proceso a cada uno de ellos, lo cual implica un mayor tiempo de ejecución del algoritmo.

Sólo si el usuario considera que este tiempo adicional se compensa y que el tiempo de ejecución de su programa disminuye entonces puede incluir la función bestmap( ) dentro del cuerpo de la construcción par. En el programa anterior se le encuentra en la línea 26.

III.2.9 Construcción Seq

KHiper no proporciona facilidades para el manejo de variables globales, por lo que en ocasiones es necesario que el proceso padre se ejecute simultaneamente con sus hijos. Para ello KHiper ofrece la construcción seq dentro del cuerpo de la construcción par.

Vease el siguiente programa

```
1  /*              KER X25.C              */
2  #include "hkernel.h"
3  #define yes 1
4  #define Cycleitem 2
5
6  proc Main (Pid pid) {
7          int i;
8          Pid soon [5];
9          Echan etomysoon [5];
10         Vchan vtomysoon [5], girar [5];
11         float dpcV;
12          tscr screen;
13
14 getpidarray (5, soon);
15 getchanarray (5, vtomysoon);
16 getchanarray (5, girar);
17
18 parbegin {
19  for (i= 0; i < 5; i++) {
20   paritem (pid, soon [i], Cycleitem, 4, girar [(i+4)%5], girar [i], vtomysoon [i], 'a'+i);
21   neigh (pid, soon [i], 2, soon [(i+4)%5], soon [(i+1)%5]);
22  }
23  bestmap ();
24
25  seq {
26   for (i= 0; i < 5; i++)
27    chaninit (& etomysoon [i], vtomysoon [i]);
28
```

```
29    for (i= 0; i < 5; i++)
30     outword (etomysoon [i], !i%5);
31    }
32   } parend;
33
34   dpcV= dpc (node, pId);
35   toscreen ((SCR, "\n\nproceso %d ejecutado por (%d) con dpcV= %4.2f\n",
36                  pId, node, dpcV));
37   end ();
38  }
39
40  proc cycleitem (PId pId, Vchan In, Vchan Out, Small Broad, char letter) {
41              int  dato, start;
42              Echan  in, out, broad;
43              tscr  screen;
44
45   params (&In);
46   chaninit (&in,  In);
47   chaninit (&out, Out);
48   chaninit (&broad, Broad);
49
50   inword (broad, &start);
51   if (start == yes) {
52    outword (out, 1);
53    inword  (in, & dato);
54   }
55   else {
56    inword  (in, & dato);
57    outword (out, dato+ 1);
58   }
59   toscreen ((SCR, "proceso %c en %d con dato == %d\n", letter, node, dato));
60   end ();
61  }
62
63  initCodes ( ) {
64   initcodes (2, Main, cycleitem);
65  }
```

Este programa usa un patrón pentágono con una ligera diferencia: el padre se comunica con sus hijos, los cuales sólo se pueden comunicar hasta que el padre les haya mandado un word de información.

De la línea 25 a la 31 se encuentra la construcción seq, que nos indica que esa sección de código se ejecutará simultaneamente con los hijos. Por lo cual el el padre les puede mandar información a sus hijos usando las líneas 29 y 30, como vemos a continuación.

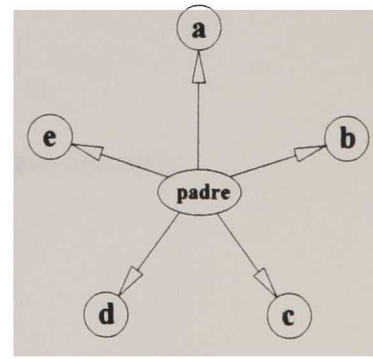


figura III.9

III.2.10 Cálculo del Mapeo según el Tráfico

El mapeo logrado con las funciones neigh se basa en cálculo de la distancia entre los hijos, sin tomar en consideración el tráfico por los enlaces. De manera que, en ocasiones, se puede tener un mapeo excelente, pero que, sin embargo congestiona excesivamente a algunos enlaces.

Por lo cual un buen algoritmo de mapeo debe tener en consideración el tráfico. Por esa razón se desarrolló un segundo modelo, el cual considera la cantidad de comunicaciones de cada canal.

La función que se utiliza para indicarle a KHiper el patrón además el caudal de cada canal es neighp. Estas características se observan en el siguiente programa.

```
1  /*                    KER X50.C                */
2  #include "hkernel.h"
3  #define FIRST 5
4  #define INC 4
5  #define Code2 2
6
7  proc code1 (Pid pid) {
8            Pid  a, b, c, d, e, f, g, h;
9            char  i, san [50];
10           Small  deepth;
11           tscr  screen;
12
13  deepth= FIRST;
14  getpids (pid, 8, &a, &b, &c, &d, &e, &f, &g, &h);
15  parbegin {
16   paritem (pid, a, Code2, 2, deepth+ INC, 'a');
17   paritem (pid, b, Code2, 2, deepth+ INC, 'b');
18   paritem (pid, c, Code2, 2, deepth+ INC, 'c');
19   paritem (pid, d, Code2, 2, deepth+ INC, 'd');
20   paritem (pid, e, Code2, 2, deepth+ INC, 'e');
21
22   neighp (pid, a, 1, 2, e, 30, c, 50, b, 50);
23   neighp (pid, b, 1, 2, a, 50, d, 20, e, 70);
24   neighp (pid, c, 1, 1, a, 50, e, 50);
25   neighp (pid, d, 1, 1, b, 20, e, 50);
26   neighp (pid, e, 3, 1, b, 70, c, 50, d, 50, a, 30);
27  } parend;
28
29  tab (deepth, san);
30  toscreen ((SCR, "%sproceso %3d ejecutado por (%d)\n", san, pid, mynode));
31  end ();
32 }
33
34 proc code2 (Pid pid, Small deepth, char letter) {
35           char  i, san [50];
36           tscr  screen;
37
38  params (&deepth);
39  tab (deepth, san);
40  toscreen ((SCR, "%sproceso %2d  %c ejecutado por (%d)\n", san, pid, letter,
41 node));
```

```
42  end ();
43 }
44
45 tab (int deepth, char* san) {
46            int i;
47  for (i= 0; i < deepth; i++)
48       san [i]= ' ';
49  san [i]= '\0';
50 }
51
52 initCodes ( ) {
53  initcodes (2, code1, code2);
54 }
```

En este programa tenemos el patrón de la figura III.5. Pero ahora la información contenida en esa figura no nos es suficiente. Ahora en el patrón debe ir indicado el caudal de cada canal y el sentido de la comunicación.

A KHiper no le resulta lo mismo viajar del procesador A al B que en sentido opuesto cuando la distancia es mayor de un enlace.

El patrón que ahora nos sirve es el que se muestra a continuación.

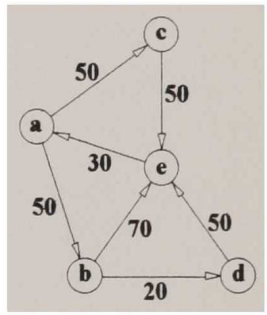


figura III.10

III.2.11 Construcción Alt

La función altitem sirve para declarar los procesos potenciales, los parámetros que usa son los mismos que los de la función paritem, la diferencia es el guardia que se coloca antes de todos. El guardia consiste de una expresión cierta o falsa de acuerdo al lenguaje C. Líneas 14,15 y 16.

Cuando de entre los guardias varios tienen el valor cierto la regla de desempate consiste en consider al primero de ellos.

A continuación se muestra un programa para ordenar en forma ascendente un conjunto de datos.

```
1  /*              KER X43.C              */
2  #include "hkernel.h"
3  enum ncodes {_Main= 1, _exchange};
4
5  proc Main (Pid pid) {
6            Pid procA, procB, procC;
7            tscr screen;
8            int  q1, q2, q3, q4;
9
```

```
10  getpids (pId, 3, &procA, &procB, &procC);
11
12  q1= 7; q2=3; q3= 5; q4= 1;
13  do {
14   altbegin {
15    altplace (node);
16    cond1= altitem  (q1 > q2, pId, procA, _exchange, 2, &q2, &q1);
17    cond2= altitem  (q2 > q3, pId, procB, _exchange, 2, &q3, &q2);
18    cond3= altitem  (q3 > q4, pId, procC, _exchange, 2, &q4, &q3);
19   } altend;
20  } while (cond1 || cond2 || cond3);
21
22  toscreen ((SCR, "%3d %3d %3d %3d\n", q1, q2, q3, q4));
23  end ();
24 }
25
26 proc exchange (PId pId, int *val1, int *val2) {
27            int temp;
28
29  params (&val1);
30
31  temp = *val1;
32  *val1= *val2;
33  *val2= temp;
34  end ();
35 }
36
37 initCodes () {
38  initcodes (2, Main, exchange);
39 }
```

La construcción alt puede generar a lo más un solo proceso. Cuando es necesario colocar a este en el mismo procesador que el padre se utiliza la función altplace. Es preferible esta función al conjunto de placeat que lo substituye.

El parámetro de altplace es el procesador donde se desea ejecutar al hijo, línea 15.

En el programa anterior tenemos la necesidad de colocar al hijo en el mismo procesador que el padre, para que le retorne valores mediante paso de parámetros por referencia.

Una función que proporciona KHiper para una programación más cómoda es el uso de altresult() que devuelve un valor de 1 si algún proceso fue elegido. Una alternativa es como podemos observar a continuación.

```
13  do {
14   altbegin {
15    altplace (node);
16    altitem  (q1 > q2, pId, procA, _exchange, 2, &q2, &q1);
17    altitem  (q2 > q3, pId, procB, _exchange, 2, &q3, &q2);
18    altitem  (q3 > q4, pId, procC, _exchange, 2, &q4, &q3);
19   } altend;
20  } while (altresult ( ) );
```

III.3 Uso de la capa de comunicaciones abstractas

III.3.1 Un ejemplo sencillo

Cuando el usuario va ejecutar programas sin las construcciones par y alt y está dispuesto a tratar ligeramente con la arquitectura puede usar solo la capa de comunicaciones. La ventaja consiste en evitar el overhead correspondiente KHiper-Control.

Los programas del usuario que usan KHiper-Com deben estar en dos archivos, uno con los códigos a ser ejecutados en la raíz y otro con los que van a ser ejecutados en todos los demás. Sólo los que corren en la raíz pueden accesar los servicios de la computadora anfitrión, por lo que es necesario dos compilaciones.

Los dos archivos deben tener el mismo nombre, excepto en la última letra. La letra para el que va a correr en la raíz es R, N para el otro.

A continuación se muestra un programa usando KHiper-Com. En el primer archivo se especifica el comportamiento de un proceso que recibe comunicación de otro que se encuentra en el procesador 1. Solo uno de los procesos no raíz llamará a la función sender, el del procesador 1.

```
1  /*              X 00 R.C                */
2  #include "ipc.h"
3  #define IN  4
4  #define OUT 12
5  #define END 1
6  extern int node;
7
8  receiver ( ) {
9              int data;
10
11 setEnd  (IN, END, OUT);
12 preEnable (IN);
13
14 inword (IN, &data);
15 printf ("La Raiz recibio un %d\n", data);
16 }
17
18 Main ( ) {
19  receiver ( );
20 }
```

```
1  /*          X 00 N.C                */
2  #define SHIFT 2
3  #define IN  4
4  #define OUT 12
5  #define END 0
6  extern int node;
7
8  sender () {
```

```
9   setEnd  (OUT, END, IN);
10  preEnable (OUT);
11
12  outword (OUT, 69+ node);
13 }
14
15 Main () {
16  if (node == 1)
17   sender ();
18 }
```

En los dos se encuentra la función Main, que es la primera que se va a ejecutar. Cuando se usa KHiper-Com la función principal se debe llamar Main.

A continuación se comentan como se da la comunicación en KHiper-Com.

Para KHiper-Com los canales virtuales no son un recurso del sistema. Cuando se quiere que dos procesos se comuniquen se declara un canal extremo por cada uno de ellos, estos se deben direccionar mutuamente. Para su identificación los canales extremos tienen un número, el cual funciona como índice dentro de una tabla que mantiene su información.

Un canal extremo "apunta" (direcciona) a otro cuando su destino es ese, el cual se encuentra en un determinado procesador.

Para que un canal extremo direccione a otro usa la función setEnd quien llena los valores destino en la tabla. Su parámetros son el canal extremo y el número de nodo y canal extremo destinos, en ese orden.

En el dibujo se encuentran los campos de la tabla de canales extremos con los valores para el programa anterior, además en línea punteada el canal virtual entre *receiver* y *sender*.

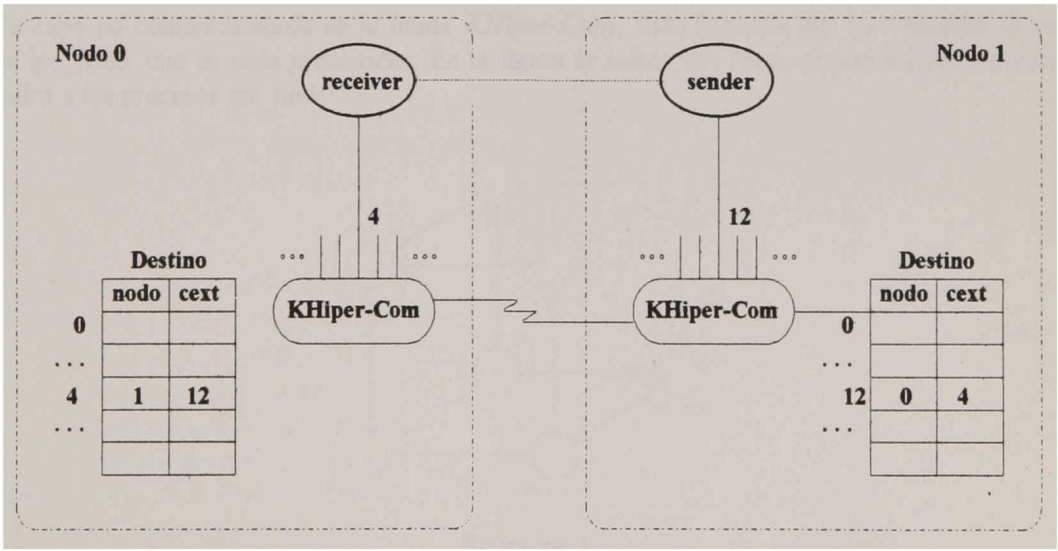


figura III.11

4 Capa de Comunicaciones

IV.1. Estructura

A la capa de comunicaciones se le llama ***KHiper-Com***. Está formada por un conjunto de procesos llamados ipc_node, uno en cada procesador. En la figura se indica con líneas punteadas los procesadores y con círculos a los procesos ipc_node.

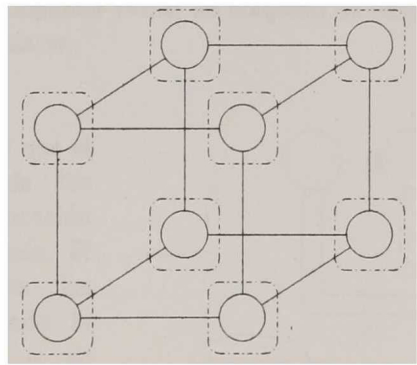


figura ipc.1

KHiper-Com es una capa de software completamente distribuida. No existen procesadores distinguidos. Cada ipc_node tiene un comportamiento diferente, conociendo su número de identificación y la dimensión del hipercubo puede determinar que enlaces puede ocupar y con que procesadores se comunica.

Un ipc_node está formado por un proceso que se encarga de recibir y otro que se encarga de mandar información, ipc_in e ipc_out respectivamente, su existencia es simultanea a los procesos del usuario, en forma genérica se refiere a ellos como ***ipc_node***.

Está característica es la propuesta de este trabajo para evitar que un sólo proceso deba manejar los canales de entrada y salida y esto llegue a ser causa de deadlocks. Esta situación se aclara en detalle en la sección *Deadlock*. Además de esta manera se evita que los enlaces se conviertan en regiones críticas por ser recursos compartidos. Esto se puede ver en la figura.

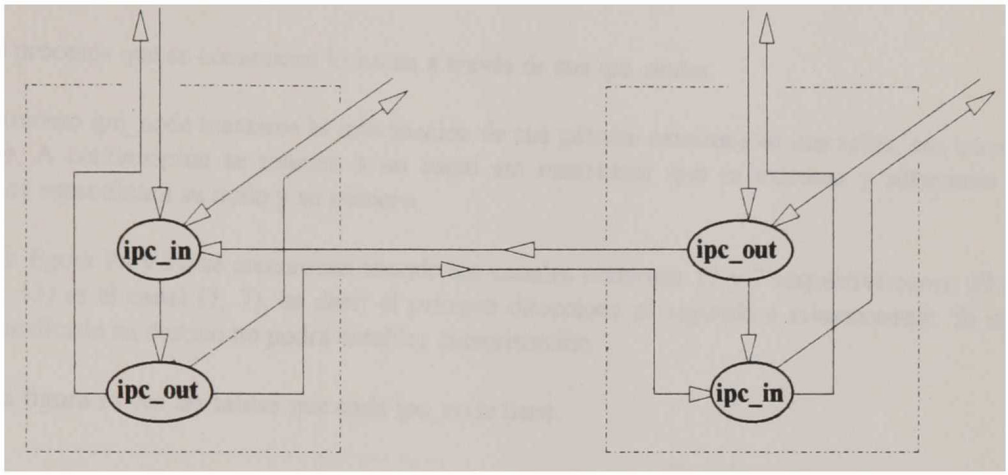


figura ipc.2

En la figura se tiene dos procesadores de un hipercubo orden 3 donde los enlaces de entrada son manejados por ipc_in y los de salida por ipc_node. También se observa que existe un canal de un ipc_in a un ipc_out.

Un ruteador se encarga de determinar a través de que enlace se debe mandar la información para que llegue a su procesador destino.

Esencialmente un ipc_in es un multiplexor y un ruteador; el ipc_out un demultiplexor y un ruteador. Tanto el multiplexor como el demultiplexor tienen un conjunto de canales asociado para mandar y recibir información con los procesos del usuario.

En la figura se puede observar que el ipc_in recibe las solicitudes de los procesos del usuario y la comunicación que llega por los enlaces de entrada. El proceso ipc_in demultiplexa las solicitudes que le llegan mandando la información al proceso destino o de mandarlas a la red de enlaces vía ipc_out quien rutea la información.

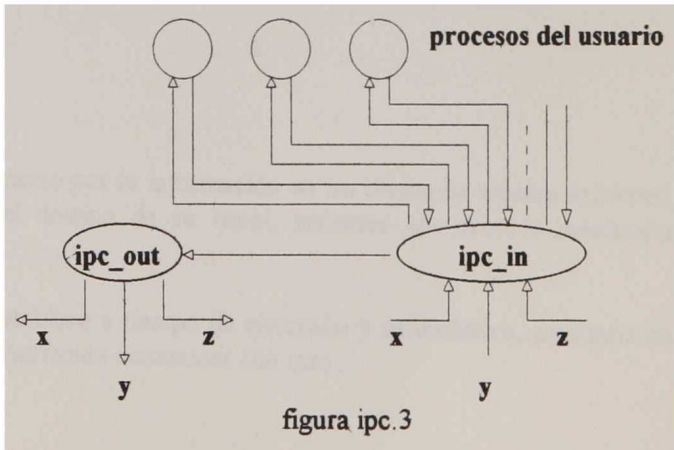


figura ipc.3

Al par formado por un canal que va del ipc_node a un proceso del usuario y a uno que viaja en sentido contrario se le llama un canal extremo. En la figura podemos ver como cada proceso del usuario tiene uno.

IV.2. Canales Virtuales y Extremos

Los procesos que se comunican lo hacen a través de sus ipc_nodes.

El proceso ipc_node mantiene la información de sus canales extremos en una tabla, sus números sirven de índice. A continuación se referirá a un canal sin mencionar que es extremo y sólamente cuando se requiera se especificará su nodo y su número.

En la figura Pa y Pb se comunican usando los canales extremos 13 y 7 respectivamente. El destino del canal (2, 13) es el canal (5, 7), es decir el primero direcciona al segundo e inversamente. Sí un canal no tiene especificado su destino no podrá entablar comunicación.

En la figura se ven las tablas que cada ipc_node tiene.

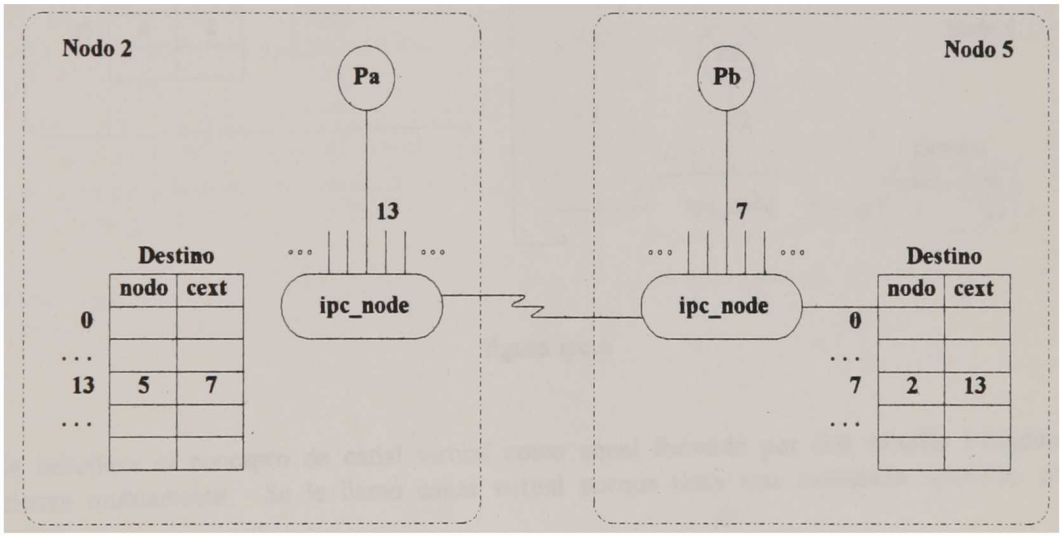


figura ipc.4

La relación entre dos procesos se da únicamente por la información en las tablas de canales extremos, es decir sí en una de las tablas se cambia el destino de su canal, entonces ese proceso mantendrá comunicación con otro diferente.

La información del destino en la tabla se establece a tiempo de ejecución y es dinámica, pudiendo un canal comunicar primero con un proceso y posteriormente comunicar con otro.

Un proceso puede tener comunicación con otros para lo cual va a necesitar tantos canales como procesos destino existan. En la siguiente figura Pa se comunica con Pb, Pc y Pd.

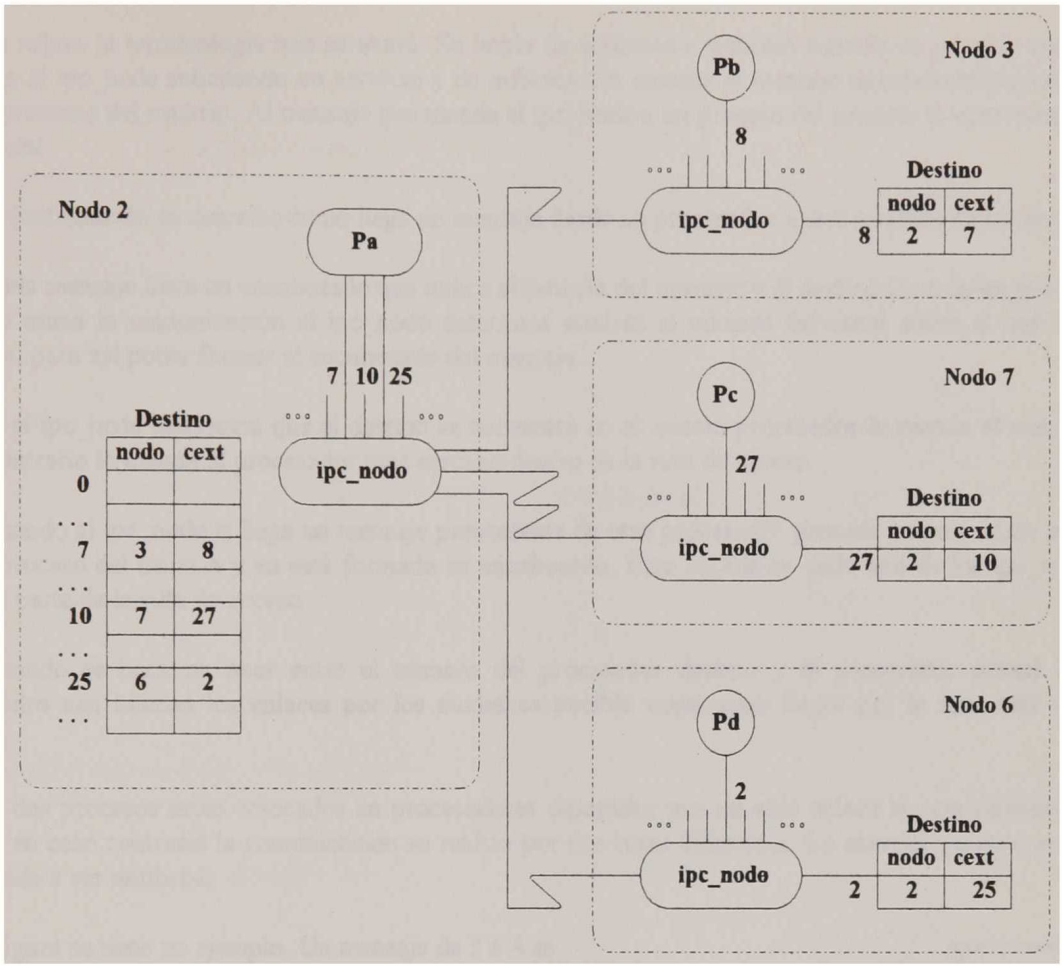


figura ipc.6

Se introduce el concepto de canal virtual como aquel formado por dos canales extremos que se direccionan mutuamente. Se le llamó canal virtual porque tiene una existencia aparente, puramente conceptual.

Los canales virtuales no tienen la restricción de ser unidireccionales. Incluso, un canal virtual puede ser utilizado primero en un sentido y posteriormente en el contrario, con la restricción de no intentar usarlo simultáneamente en ambos sentidos, que además resultaría absurdo.

Es responsabilidad del usuario el uso adecuado de los canales virtuales.

IV.3. Ruteo

Se aclara la terminología que se usará. Se habla de solicitud o petición cuando un proceso manda un mensaje al ipc_node solicitando un servicio y de información cuando el mensaje mandado transporta datos de los procesos del usuario. Al mensaje que manda el ipc_node a un proceso del usuario le conocemos como indicación.

A continuación se describe como llega un mensaje desde un procesador a otro a través de los ipc_node.

Cada mensaje lleva un encabezado que indica el tamaño del mensaje y el destino.Cuando un proceso del usuario inicia la comunicación el ipc_nodo determina cual es el número del canal sobre el que llegó la petición, para así poder formar el encabezado del mensaje.

Sí el ipc_node determina que el destino se encuentra en el mismo procesador le manda el mensaje, en caso contrario lo manda al procesador más cercano dentro de la ruta de acceso.

Cuando al ipc_node le llega un mensaje proveniente de otro procesador procede como cuando proviene de un proceso del usuario y ya está formado su encabezado. Esto sucede en cada uno de los ipc_nodo que forman parte de la ruta de acceso.

Cuando se hace un exor entre el número del procesador destino y el procesador actual los bits encendidos nos indican los enlaces por los cuales es posible viajar para llegar por la ruta más corta al destino.

Sí dos procesos están colocados en procesadores separados por un sólo enlace la ruta consiste en ese enlace, en caso contrario la comunicación se realiza por dos rutas diferentes. Lo anterior permite un tráfico que tiende a ser uniforme.

En la figura se tiene un ejemplo. Un mensaje de 3 a 4 se da por la ruta 3-2-0-4 en cambio de 4 a 3 se da por 4-5-7-3.

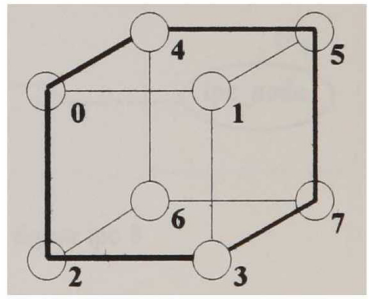


figura ipc.7

IV.4. Protocolo de comunicaciones

El paso de un mensaje consiste en copiarlo del proceso que realiza la operación de salida al proceso que hace la entrada. Si el destino de un proceso no está listo para comunicarse el kernel desencalendariza al proceso, en caso contrario realiza la copia y recalendariza al destino.

A un canal software proporcionado por el compilador se le llamo canal, básicamente es una localidad de memoria donde a través de una serie de marcas el kernel se informa de su estado.

Para sincronizar a los procesos que se comunican mediante un canal virtual se implementó el protocolo que a continuación se describe. Para efectos de claridad se considerará que in y out son llamados en procesadores diferentes.

Se utiliza un campo de la tabla de canales como campo de marca, su estado puede ser libre o anotado o marcado. De los dos campos de marcas correspondientes a los canales extremos se decidio utilizar el que corresponde con el proceso que hace la entrada. Como se observará en la función alt esto mejora la eficiencia.

Para efectos de claridad se va a llamar genéricamente in y out a las funciones que trabajan con los canales extremos, chan_in y chan_out a los que trabajan con los naturales.

Existen dos secuencias posibles de in-out, de acuerdo a quien sucede primero.

Esencialmente ambos casos son iguales.

Se revisará el caso in antes de out, para la explicación se utilizará la siguiente figura.

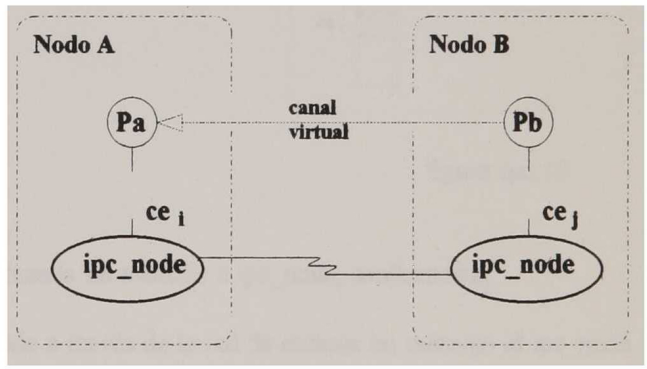


figura ipc.8

En la llamada a la función in va indicado el número de canal extremo a ser utilizado, en este caso ce_i.

Cuando a ipc_node le llega el mensaje de la función in determina el estado del campo de marca de ce_i, lo encuentra limpio, lo marca y continua atendiendo solicitudes de otros procesos. Como se puede ver en la figura.

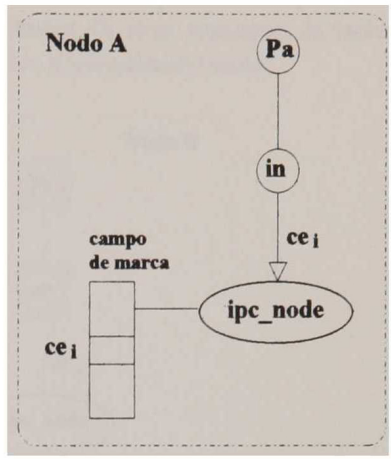


figura ipc.9

A continuación la función in espera un mensje proveniente de ipc_node, dado que este no le manda nada, Pa es desencandelarizada vía la función in. En la figura se observa este hecho.

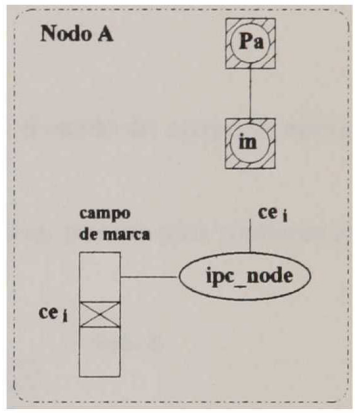


figura ipc.10

Posteriormente es llamada la función out quien manda un mensaje a ipc_node, mediante ce_j.

El ipc_node como respuesta a este mensaje manda a través de la red de enlaces un mensaje al ipc_node del procesador A indicando que el out ya está dispuesto a la comunicación.

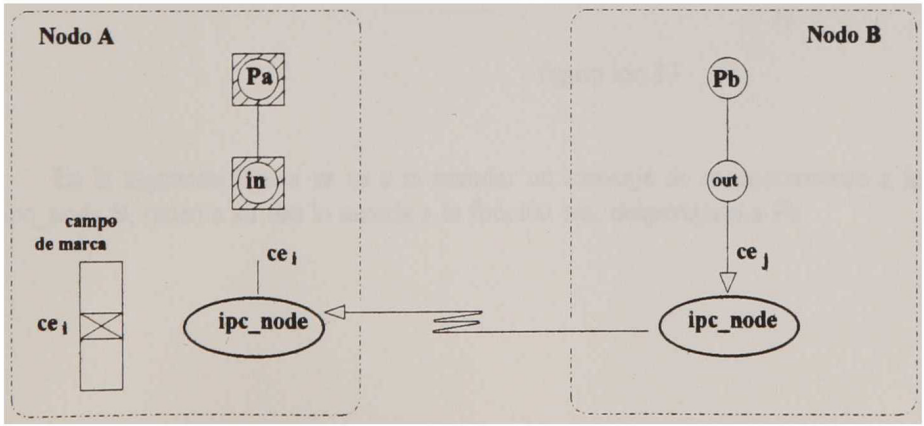


figura ipc.11

Después de mandar el mensaje el ipc_node continua atendiendo solicitudes de otros procesos, en tanto que la función out espera un mensaje proveniente del ipc_node, por lo cual es desencalendarizado.

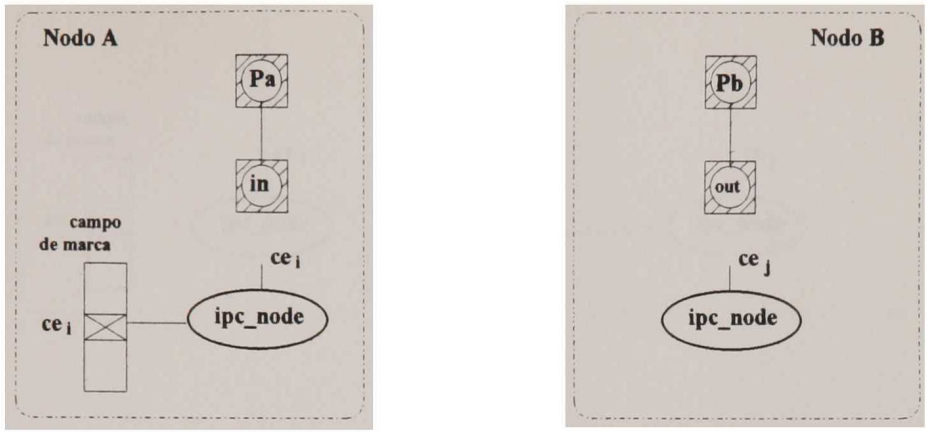


figura ipc.12

En la figura anterior se encuentra que mientras el ipc_node A determina el estado del campo de marca de ce_i, Pa y Pb se encuentran desencalendarizados.

El campo anotado indica que in ya ocurrio y que está dormido esperando un mensaje para continuar el protocolo, por lo cual ipc_in le manda ese mensaje.

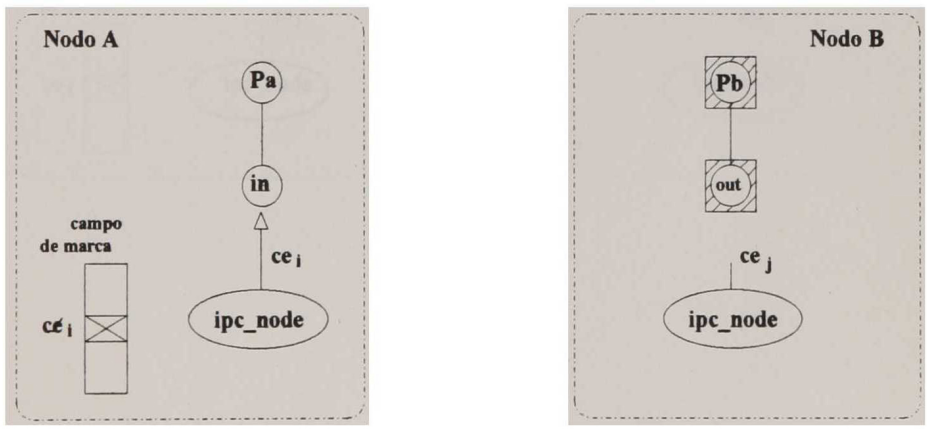


figura ipc.13

En la siguiente figura se ve a in mandar un mensaje de reconocimiento a ipc_node. Quien lo pasa al ipc_node B, quien a su vez lo manda a la función out, despertando a Pb.

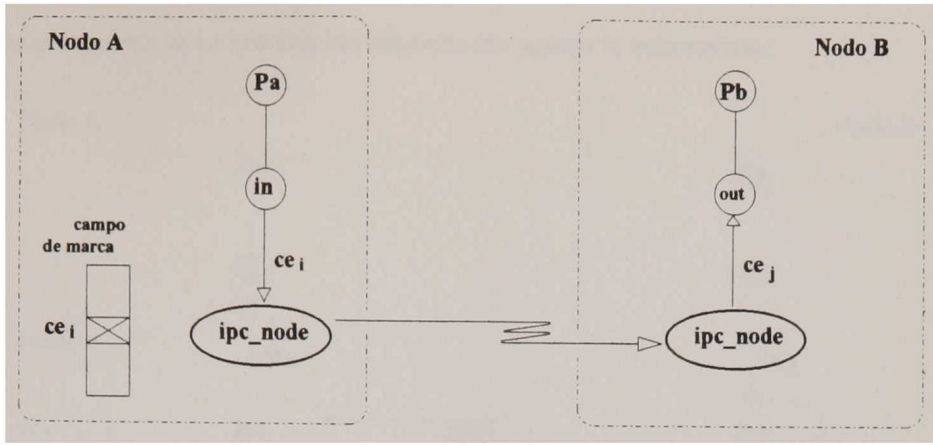


figura ipc.14

Después de que in manda el mensaje de reconocimiento el proceso Pa vuelve a ser desencalendarizado, siguiente figura, en esta ocasión porque in espera la información.

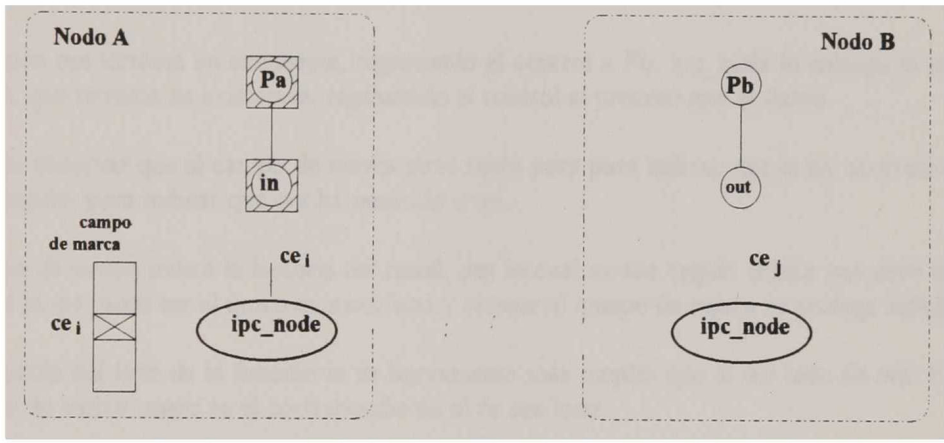


figura ipc.15

En la figura siguiente se ve que una vez despierto out manda la información.

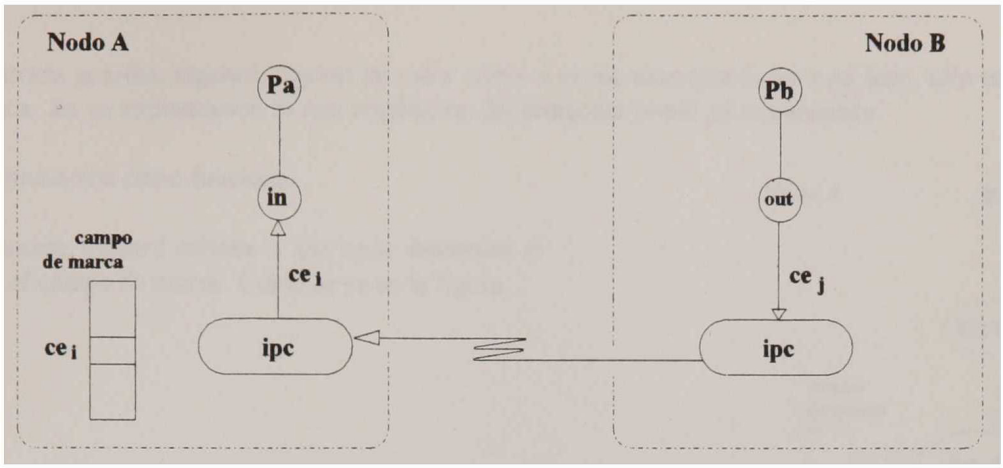


figura ipc.16

La función out termina su existencia, regresando el control a Pb. ipc_node le entrega la información a la función in, que termina su existencia, regresando el control al proceso que la llamó.

Se puede observar que el campo de marca sirve tanto para para indicar que in ha ocurrido o no, o visto desde otro ángulo, para indicar que out ha ocurrido o no.

El campo de marca indica la historia del canal, por lo cual es una región crítica que debe ser protegida. Al permitirle al ipc_node ser el único en modificar y revisar al campo de marca se protege dicho campo.

El protocolo del lado de la función in es ligeramente más amplio que el del lado de out. Esto debido a que el campo de marca usado es el correspondiente al de ese lado.

Por brevedad algunas figuras no están del todo desglosadas.

IV.5. Entrada guardia

La entrada guardia, inguard, regresa un valor cierto si el out correspondiente está listo, sólo en ese caso se comunica. En su implantación es una ampliación del protocolo in-out ya mencionado.

A contiunación como funciona.

La función inguard solicita a ipc_node determine el estado del el campo de marca. Como se ve en la figura.

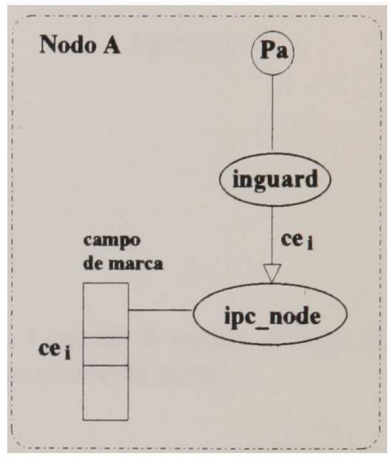


figura ipc.17

Si el ipc_node encuentra el campo de marca libre se lo comunica al inguard. A diferencia la función in, no lo marca, puesto que ello significaría que va a esperar a su out.

Con esta información el inguard finaliza su ejecución.

En caso de que el campo de marca no se encuentra libre inguard se comporta exactamente igual que in.

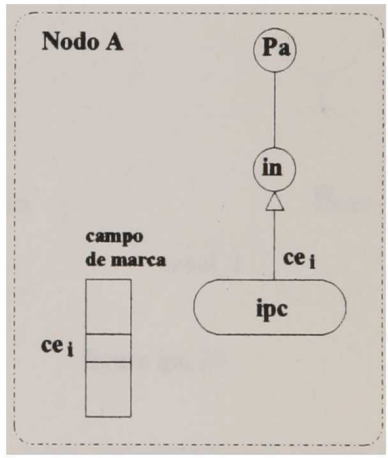


figura ipc.18

IV.6. Deadlock

En un sistema basado en c.s.p. puede ocurrir un deadlock cuando dos procesos intentan comunicarse entre sí por diferentes canales.

Considerese los procesos A y B del siguiente fragmento y la gráfica que le corresponde.

```
1   proc A ( ) {
2     . . .
3     out (canal_1, dato1);
4     . . .
5     in (canal_2, &dato2);
6     . . .
7   }
8
9   proc B ( ) {
10    . . .
11    in (canal_1, &valor_1);
12    . . .
13    out (canal_2, valor_2);
14    . . .
15  }
```

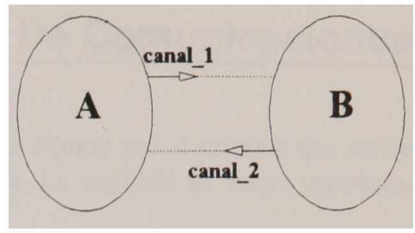


figura ipc.19

En este fragmento puede ocurrir un deadlock. Si A ejecuta su línea 3 cuando B ejecuta su línea 11 ninguno de los dos podrá completar su comunicación, entrando el sistema en un abrazo mortal.

Originalmente todas las actividades de KHiper-Com las realizaba un solo proceso por nodo. En ocasiones entraba el sistema en abrazo mortal por una situación igual a la mostrada. La solución que se encontró en este trabajo es la que se muestra en la siguiente figura.

La solución consiste en dividir a cada proceso en dos procesos. Cada parte maneja únicamente un sentido de la comunicación.

Así en el caso de A, ahora Ain se encarga de la entrada y Aout de la salida. Los dos se comunican entre sí, para mantener la información que antes se manejaba en un solo proceso.

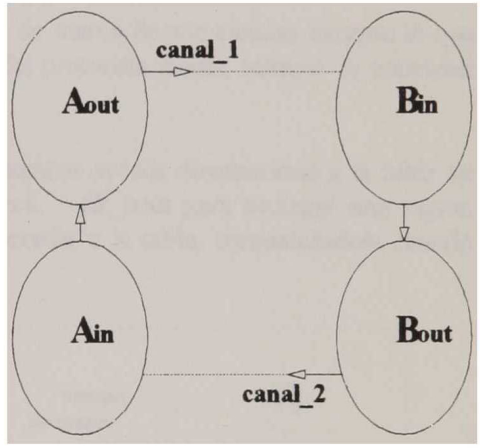


figura ipc.20

Esta solución proporciona flexibilidad ya que el proceso dividido puede simultáneamente recibir y mandar información. Esta es la razón por la cual existe ipc_in e ipc_out.

Cuando un proceso necesita conocer que destinos out están listos o esperar por el primero que arribe llama a alt_wait, o si no requiere esperar entonces llama a alt_no_wait. La implanté en varias versiones, genéricamente las agrupo en las dos mencionadas.

A continuación se explica la **función alt_wait**.

Esta función determina el estado de los campos de marca de aquellos canales extremos que le indicaron, informa al proceso que la llamo el número de canal si alguno de ellos está anotado. En caso de que todos los campos están limpios los marca. Esta es una marca diferente a la que indica que un in está esperando.

A continuación la función alt_wait espera una indicación proveniente del ipc_node, razón por la cual el proceso que llamó queda desencalendarizado. Cuando a ipc_node llega un mensaje de inicio de protocolo de parte de un out, revisa el campo de marca si este está indicado con marca de alt manda un mensaje al alt_wait.

Después de recibir el mensaje el alt_wait limpia los campos de marca de sus canales excepto el que corresponde al out que ha ocurrido, a quien le deja la anotación del protocolo in-out, termina su actividad regresando el número de canal que está listo.

alt_wait a diferencia de las funciones de KHiper-Com ya expuestas accede directamente a la tabla de canales extremos, este hecho convierte a la tabla en una región crítica. alt_wait para proteger esta región solicita al ipc_node que detenga su actividad cada vez que va a acceder a la tabla, comunicándole cuando termina. Como se puede ver en la siguiente figura.

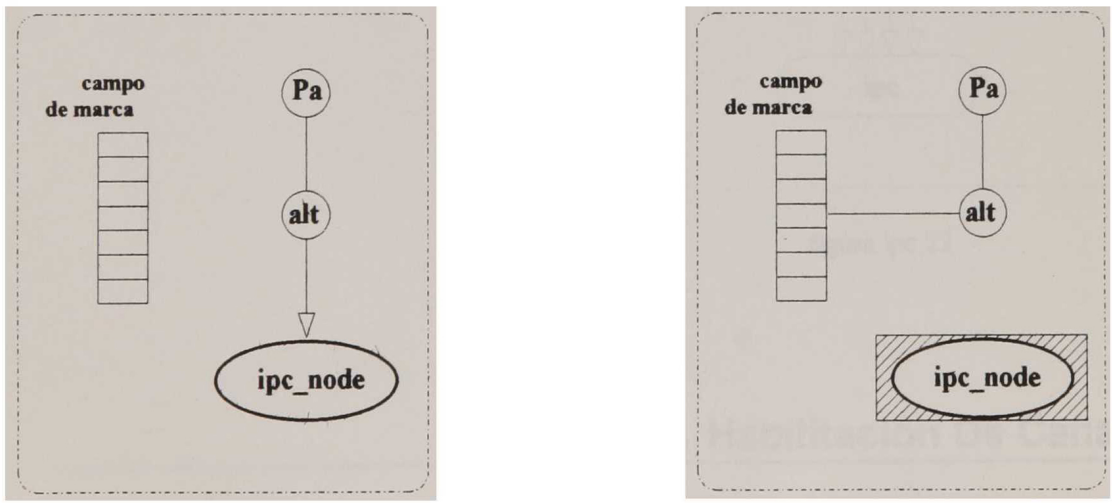


figura ipc.21

La función alt_wait se beneficia de que el campo de marca utilizado es el perteneciente al canal de la operación in, por lo cual solo necesita investigar campos localizados en la tabla del procesador local.

A continuación se explica la **función alt_no_wait**.

Las actividades de alt_no_wait son un subconjunto de las de alt_wait. Su acción se reduce a determinar si existe un campo marcado, regresando su número. En caso de no existir alguno regresa un valor negativo.

IV.8. Broadcast

La función broadcast se encarga de mandar un mensaje sobre un conjunto de canales, el tamaño de este lo elige el usuario, es una denominación genérica a un conjunto de funciones.

Para efectuar la comunicación la función broadcast genera un hilo que lleva a cabo un out por cada canal de su conjunto. Así en la figura se ve como Pa manda un mensaje por varios canales.

Esta función debe ejecutar las salidas en forma concurrente. Si se realizan en forma secuencial pueden provocarse deadlocks.

Se implantó la función en dos versiones. La primera termina sí y solo sí todas las operaciones de entrada han sido realizadas. La segunda únicamente crea los hilos y termina sin verificar quien se ha comunicado.

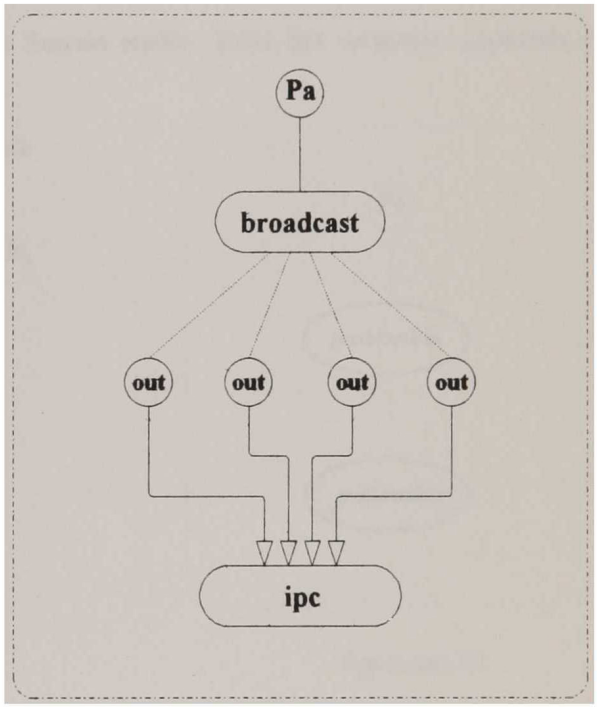


figura ipc.22

IV.9. Habilitación De Canales

Existen una serie de funciones que ofrece la capa de comunicaciones que son sólo de utilidad a KHiper-Control, dentro de estas se encuentran las funciones de habilitación del canal.

Una de las actividades de KHiper-Control es el manejo de los canales virtuales, y a este le corresponde inicializarlos.

Al iniciarse el protocolo de comunicación sí alguno de los canales extremos no ha direccionado a su compañero se provocará un estancamiento. Se debe, entonces, inhibir las comunicaciones por un canal virtual que no este inicializado. A este permiso se le llamó habilitación.

Para llevar el control de la habilitación del canal extremo se incrementó un campo en la tabla de canales extremos que informa si el canal está habilitado, dado que el canal virtual está compuesto por dos canales extremos entonces la habilitación de este se da por la habilitación de sus componentes.

Para el control se incrementó como primer actividad de una función in o out el determinar el estado del campo de habilitación de su canal. Sí está habilitado inicia el protocolo de comunicación, en caso contrario queda en espera de un mensaje de habilitación, es decir queda desencalendarizado.

La habilitación de un canal extremo la realiza la función enable. Tiene dos versiones: preenable y postenable.

La función preenable es muy simple, sólo declara como cierto el campo de habilitación.

La función postenable crea el proceso _postenable, como se ve en la figura.

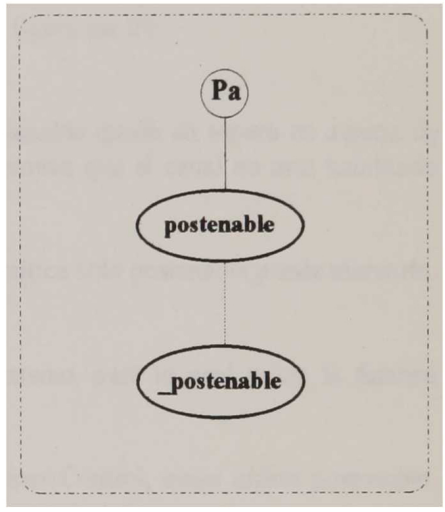


figura ipc.23

La función _postenable está pensada para ser utilizada cuando no se sabe el momento en que ocurrirán la función in o out con respecto a esta.

Sí alguna función in-out ha intentado usar el canal debe estar en espera de un mensaje de habilitación, en tal caso _postenable se lo manda, permitiendo que inicie su protocolo. En la figura se ve este evento.

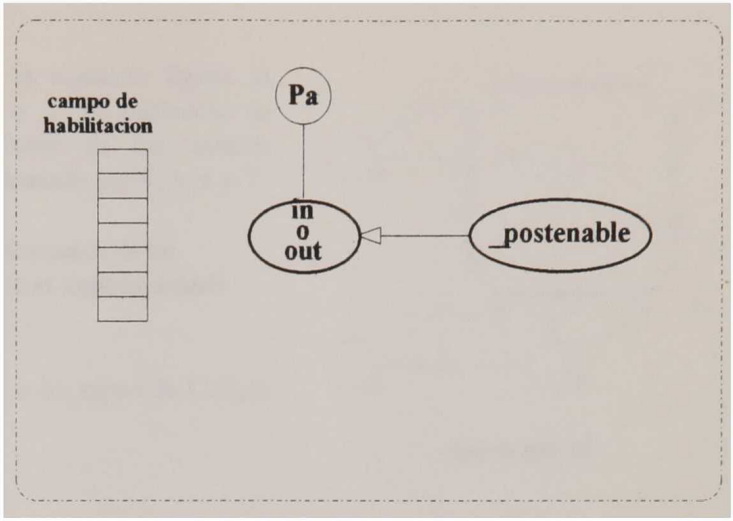


figura ipc.24

En caso de que no haya ocurrido alguna función in o out _postenable queda en espera de alguna de ellas para mandar el mensaje de habilitación. Cuando el in o out determine que el canal no esta habilitado solicitará el mensaje.

Para que el campo de habilitación no se convierta en una región crítica solo postenable puede marcarlo. Realizando este marcaje entre dos mensajes con in o out.

El control implica también retirar la habilitación a un canal extremo, para lo cual existe la función inhibit.

La habilitación de los canales surge como una necesidad de KHiper-Control, quien utiliza postenable, razón por la cual la función preenable el usuario de KHiper-Com la ve innecesaria.

IV.10. Escalamiento De La Capa De Comunicaciones

La dimensión máxima de un hipercubo está determinada por el número de enlaces disponibles **en todos y cada uno** de los procesadores.

El T800 ofrece 4 enlaces, el que va conectado a la PC utiliza uno para este efecto, siendo el que limita la dimensión de el hipercubo a un máximo de 3.

KHiper funciona sobre hipercubos de orden hasta de 3, para lograr esta escalabilidad se aprovechó la característica recursiva de formación de un hipercubo.

En un hipercubo de orden d el protocolo de comunicaciones entre procesadores que pertenecen a un hipercubo de orden menor se da exclusivamente a través de los enlaces de este último.

Por ejemplo en el hipercubo d-3 de la siguiente figura se comunican dos procesos colocados en 4 y 7. El protocolo de comunicaciones se da únicamente a través de los enlaces resaltados, pertenecientes al hipercubo d-2 formado por 4, 5, 6 y 7.

Esta característica de la capa de comunicaciones de no anclarse a un orden determinado de hipercubo es imprescindible para lograr que este sea escalable.

Se usó variables globales que informan a las capas de KHiper sobre la dimensión actual.

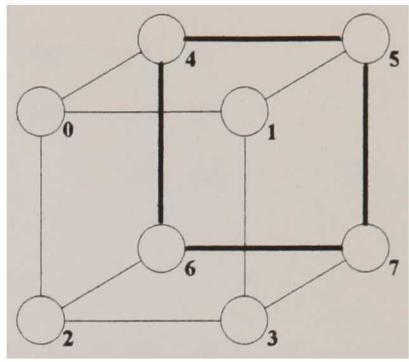


figura ipc.25

IV.11. Medida de la sobrecarga

KHiper proporciona funciones que permitan analizar su rendimiento. La capa de comunicaciones lleva cuenta del número de bytes por enlace de salida.

KHiper-Com clarifica esta información discriminando entre el uso de los enlaces por parte del programa del usuario así como por parte de KHiper-Control.

Esto lo logra la capa de comunicaciones proporcionando un par de funciones in-out de uso exclusivo de la capa de control de procesos.

IV.12. Inicialización y Finalización

Se inicia la capa de comunicaciones llamando a la función iPCInit quien inicializa la tabla de canales extremos, los canales software, los contadores de bytes por enlace de salida y crea los procesos ipc_in e ipc_out.

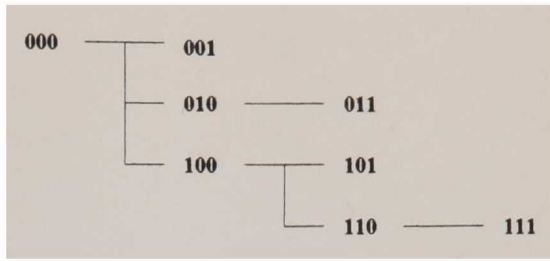
La finalización de KHiper-Com se da un orden específico. El fin de un ipc_node no debe impedir que otros se enteren del mensaje de fin de actividades.

La solución que se implantó consiste en desactivar ipc_node's en forma de árbol span. El ipc_in finaliza su actividad cuando le llega el mensaje de fin pero antes le avisa a su ipc_out. ipc_out le comunica al ipc_in correspondiente.

Se inicia este árbol invocando a la función ipcEnd en el procesador raíz. En caso de ser invocada en otro procesador no todos los ipc_node finalizan.

En la siguiente figura se ve el árbol span para un hipercubo d-3.

En el inciso a) se muestra como se va dando el movimiento de bits para generar el número del siguiente nodo, en b) se encuentran los números en base diez.



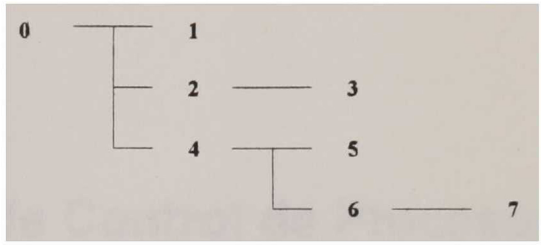
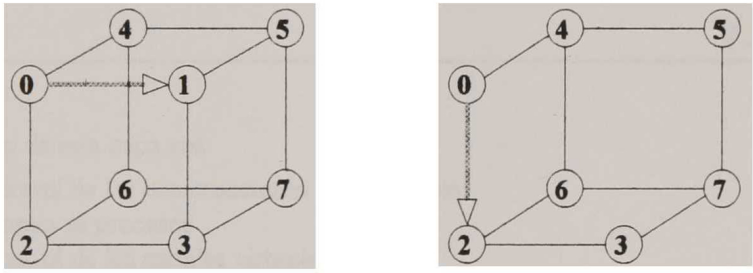


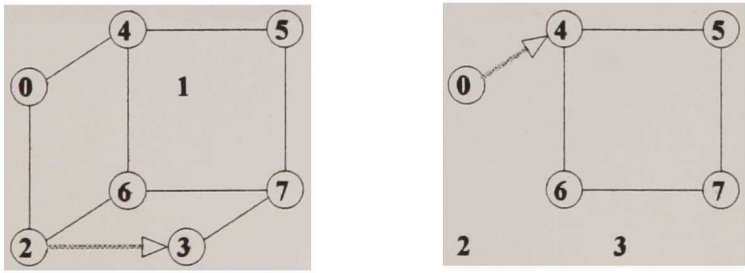
figura ipc.26

A continuación se puede ver como se da la transmisión de los mensajes de finalización en un hipercubo de orden 3 para los primeros cuatro ipc_node's. En la figura aparece un circulo indicando que existe el ipc_node. Con una flecha se indica que ipc_node comunica el mensaje de finalización.



a) **b)**

figura ipc.27



c) **d)**

figura ipc.28

Para que el main que esté informado de su terminación implanté la función *waitForEnd*, termina cuando los procesos del ipc_node le informan de su terminación.

5 Capa de Control de Procesos

V.1. Estructura

Las funciones de esta capa son:

- Control de las construcciones par, alt y seq.
- Mapeo de procesos
- Control de los canales virtuales
- Facilidades para mandar a pantalla y recibir del teclado.

Dentro del sistema el raíz es un procesador distinguido, puede hacer uso de los recursos de la computadora anfitrión, además siempre va a tener el mismo número en cualquier orden de hipercubo, es decir se mantiene invariante al escalamiento.

KHiper-Control no se plantea distribuido, la razón de esto es debido a que el máximo tamaño de la red es pequeño. El gasto de coordinación, así como la complejidad de esta capa no justifica el esfuerzo de diseño y posiblemente tampoco el overhead involucrado [13].

La encargada de llevar las actividades centrales es la función *ss_ctrl*, quien junto con un proceso *slv_ctrl* en cada procesador forman la capa de control. Cada *slv_ctrl* le comunica al *ss_control* las solicitudes de los procesos locales.

En la figura se ve como se comunica *ss_ctrl* con los *slv_ctrl*, mediante canales virtuales.

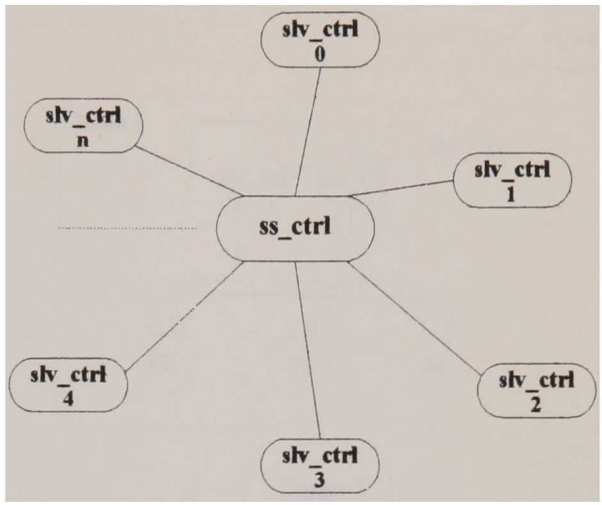


figura V.1

En la figura se ve la relación de la capa de control la de comunicaciones y los procesos del usuario.

Sólo se dibujó un proceso de usuario en cada nodo, Pi y Pj. También se ve la diferencia entre el procesador raíz y los demás.

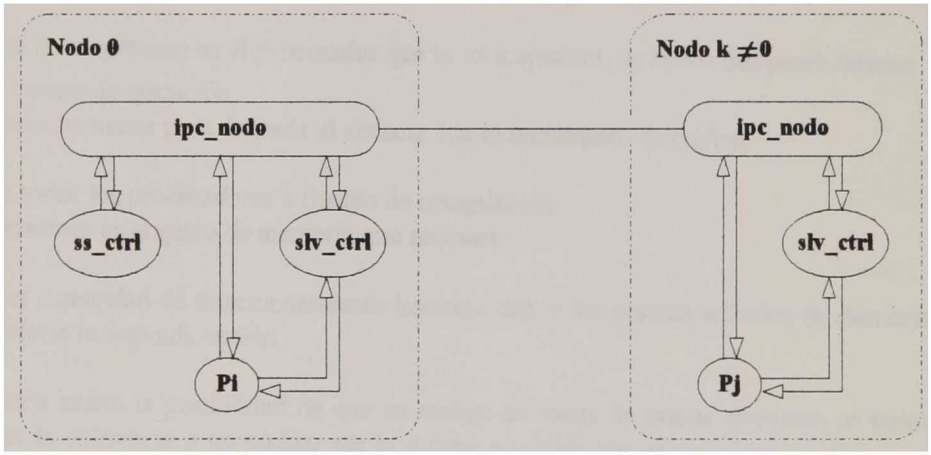


figura V.2

Por las razones que en este trabajo se dividieron en dos a ipc_node se divide a slv_ctrl en slv_IO y slv_OI, el primero recibe las solicitudes de los procesos del usuario y las manda *ss_ctrl*, el segundo de recibir las indicaciones de control por parte del *ss_ctrl*. Como se puede ver a continuación.

De las figuras anteriores se ve que cada proceso del usuario se comunica con la capa de control mediante dos canales naturales.

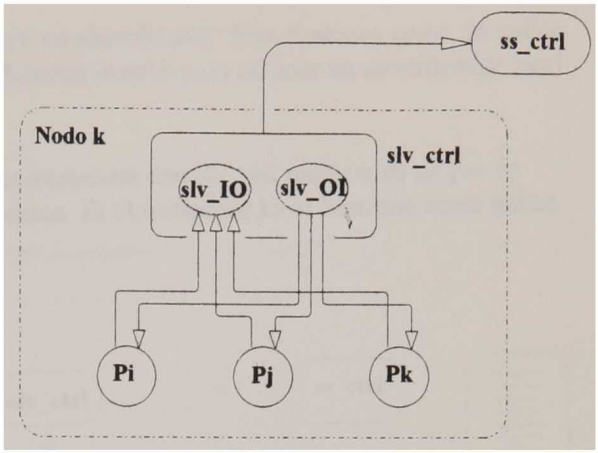


figura V.3

V.2. Código simétrico

Para colocar el código de un proceso en el procesador que lo va a ejecutar, se tienen dos posibilidades.

- Mandarlo a tiempo de ejecución
  Requiere menos memoria pero degrada al sistema con el incremento de tráfico.

- Colocarlo en todos los procesadores a tiempo de compilación.
  Su única desventaja es el gasto de memoria que requiere.

El transputer tiene una capacidad de direccionamiento bastante alta y los precios actuales de memoria son bajos por lo que se preferio la segunda opción.

Usando código simétrico existe la posibilidad de que un código no tenga la misma dirección en todos los procesadores. Para que la referencia a un código sea la misma en todos los procesadores se forma en cada uno de ellos una tabla de apuntadores a función, donde un cierto código ocupa la misma entrada en todas las tablas. Para referirse a un código se refiere a su índice.

V.3. Identificador del proceso

La función main, en cada procesador, crea el proceso slv_ctrl y en la raíz llama a la función ss_ctrl iniciando con ello la capa de control de procesos.

La primera acción de ss_ctrl es crear al proceso con código cuyo índice es uno. Después se dedica a esperar información por parte de los slv_ctrl y responderles en caso necesario.

KHiper-Control lleva el control de un proceso mediante su identificador. Este funciona como un índice en la tabla de procesos, accedida únicamente por ss_ctrl. Además le sirve para obtener un identificador local para comunicarse con la capa.

En la figura anterior se observa que un proceso para comunicarse con slv_ctrl requiere de un par de canales naturales. Estos forman parte la tabla local de procesos. El identificador local funciona como índice de la tabla.

Se solicita un identificador llamando a *newtpid*, quien comunica esa petición a ss_ctrl.

El slv_ctrl manda esta solicitud al ss_ctrl, con la información de quien es el proceso que lo esta solicitando.

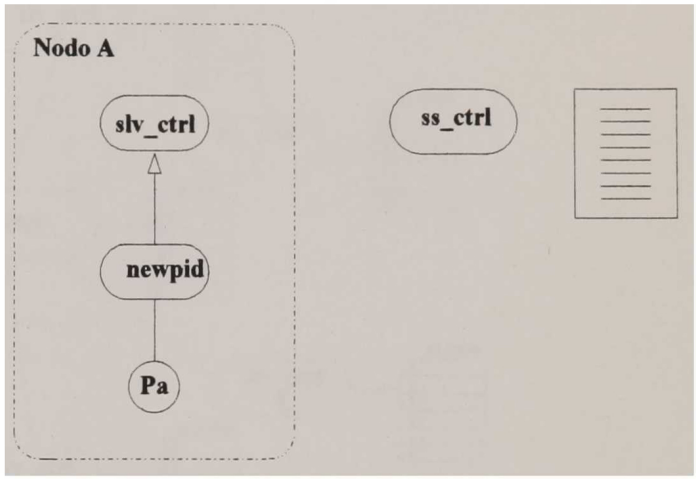


figura V.4

Después de pasar el mensaje Pa es desencalendarizado ya que la función newpid espera la respuesta. slv_ctrl continua su actividad. Como se ve en la figura.

Por su parte ss_ctrl llama a una función que le informa que número de identificador esta disponible, en cuanto lo tiene se lo manda al slv_ctrl solicitante.

A su vez slv_ctrl pasa el mensaje a newpid quien finaliza retornando el valor Pa.

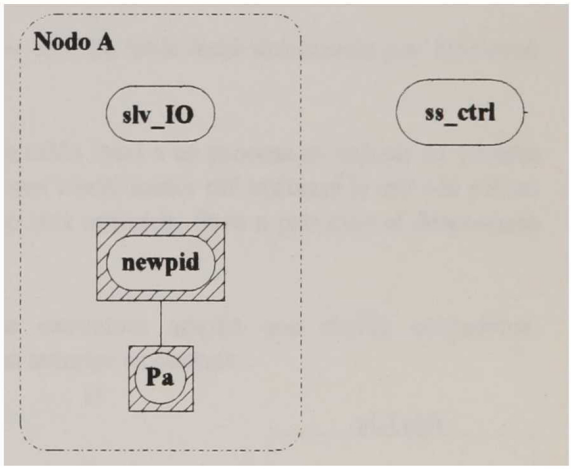


figura V.5

La liberación de un identificador de proceso se da cuando este avisa de su terminación.

V.4. Control de procesos concurrentes

Para el control de los procesos se utiliza una tabla central de procesos, llamada pTable y colocada en la raíz, y una tabla local de procesos en cada procesador llamada pLTable. Como se ve en la figura.

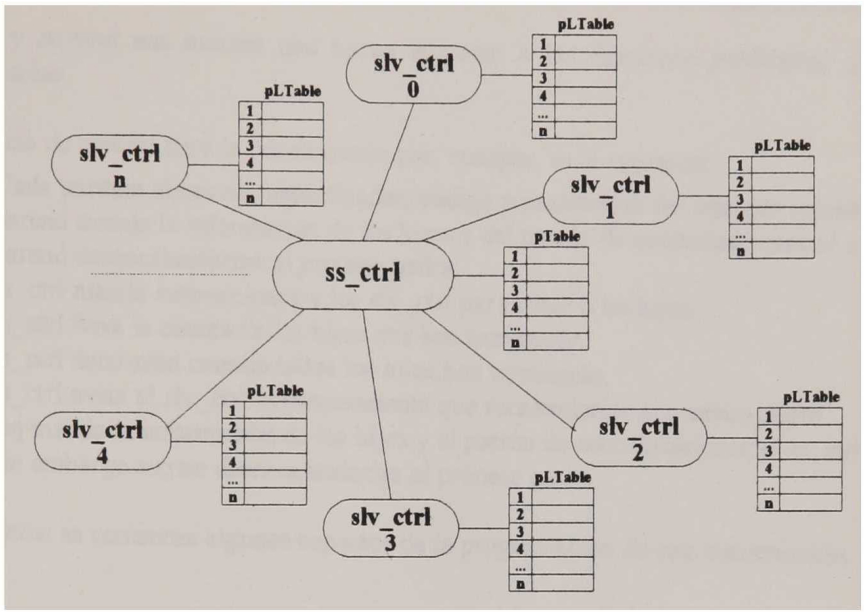


figura V.6

La tabla central es accedida únicamente por funciones ss_ctrl. La tabla local únicamente por funciones llamadas por el slv_ctrl.

La solución directa en la asignación de una entrada de la tabla local a un proceso es utilizar su número de identificación como índice dentro de la tabla. Un proceso con identificador pid utilizará la entrada pid en la tabla local, la entrada pid en los restantes procesadores no será utilizada. Para n procesos el desperdicio es de $n(2^d - 1)$ entradas.

Cada entrada en la tabla local de procesos es una estructura amplia que abarca contadores, apuntadores, canales naturales, etc., lo cual hace a la solución anterior es costosa.

Para resolver este problema se uso una tabla previa, llamada entry, en la cual el identificador del proceso es utilizado como índice. El único contenido de esa tabla es un índice de entrada de la tabla local de procesos. Como se ve en la figura.

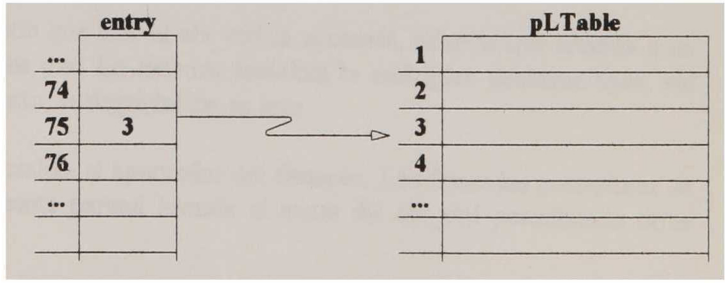


figura V.7

Siguen existiendo entradas no utilizadas, pero de una tabla bastante simple.

Para evitar que los identificadores que un proceso solicitó y no asignó puedan no ser liberados ss_ctrl cuando es avisado de que un proceso ha terminado determina que identificadores solicitó y los libera.

V.4.1. La construcción PAR

parbegin y *parend* son macros que hacen aparecer a las funciones *parBegin()* y *parEnd()* como palabras reservadas.

La secuencia de control para la construcción par, conspar, es la siguiente:

a) Cada paritem almacena identificador, código y parámetros del hijo que representa.
b) parend manda la información de los hijos y del patrón de comunicaciones al ss_ctrl.
c) parend desencalendariza al proceso padre.
d) ss_ctrl manda instrucciones a los slv_ctrl para crear a los hijos.
e) ss_ctrl lleva la cuenta de los hijos que han terminado.
f) ss_ctrl determina cuando todos los hijos han terminado.
g) ss_ctrl avisa al slv_ctrl correspondiente que recalendarize al proceso padre.
b) seq manda la información de los hijos y el patrón de comunicaciones al ss_ctrl
Sin embargo seq no desencalendariza al proceso padre.

A continuación se comentan algunos aspectos de la programación de esta construcción.

V.4.1.1 Información de los hijos y Serialización

El uso de los canales virtuales implica una sobrecarga por cada instancia in-out, independientemente del tamaño del mensaje. Por lo cual aunque era posible que cada paritem mandara su información se eligio formar un almacén.

Originalmente se consideró que el tiempo que se consume en declarar una conspar era mínimo.

Se optó, entonces, por un solo almacén en cada procesador. Se tenía la opción de que fuera manejado por las funciones de la construcción, esto implica la posibilidad de que se mezclará información múltiples declaraciones simultáneas.

Para proteger esta región crítica se permitio que sólo el slv_ctrl la accesará, además que atienda a un sólo proceso en caso de múltiples declaraciones par. Lo anterior **serializa la múltiples declaraciones**, sin embargo si el tiempo de cada una de ellas es corto, la degradación es leve.

La función parbegin ancla al slv_ctrl e inicializa el apuntador del almacén. Los llamados posteriores de paritem le indican al slv_ctrl llenarlo. Finalmente parend levanta el ancla del slv_ctrl permitiendo otras declaraciones.

parend desencalendariza al proceso padre y espera un mensaje de slv_ctrl, que llegará cuando todos los procesos hijos terminen.

El control de un proceso padre se da mediante una barrera, así cuando ss_ctrl recibe la información de una conspar incrementa la barrera del proceso padre con el número de hijos de la construcción.

Cada ocasión que un proceso termina avisa al ss_ctrl, quien disminuye en uno la barrera al padre y revisa si el valor es cero, en este caso manda un mensaje al slv_ctrl para recalendarizar a este proceso padre.

Invocar a la función parend es solicitar la transferencia de información de los hijos al ss_ctrl y desencalendarizar al proceso padre. Referirse al final de la construcción es indicar que todos los procesos hijos han terminado de ejecutarse.

El supuesto de que la declaración de la construcción es breve no siempre se cumple y anclar el slv_ctrl adolece de algunos defectos:

- El ancla evita que el ss_ctrl atienda a otros procesos, incluso aún cuando no estén declarando una conspar sino solicitando otro servicio, como se ve en la figura.

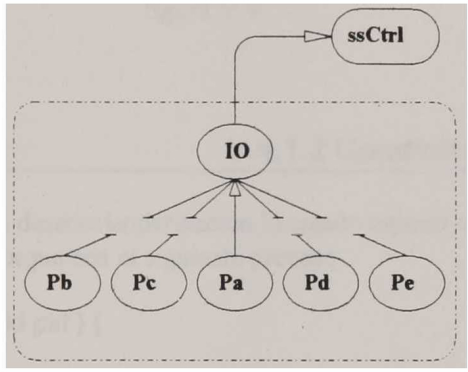


figura V.8

- Esta serialización se acentúa cuando se utiliza la construcción seq, mediante la cual los hijos corren simultáneamente con el padre.

- Además puede ocurrir un deadlock. Considerese el caso en que un hijo es colocado en donde su padre. El ss_ctrl nunca se enterará de su final puesto que el slv_ctrl no va atender su aviso. Esto si acaso el hijo no tenía parámetros, ni necesitó solicitar servicios de la capa.

  Para evitar este problema se deben implantar mecanismos de control más complejos. Sin embargo no es meritorio trabajar en este sentido, puesto que sería aceptar como necesaria la serialización de los procesos.

Una solución adecuada para impedir la serialización es crear una tabla individual por cada proceso. Como se puede observar en la siguiente figura. Con lo cual se evita que se mezcle información de diferentes construcciones y se evita serializar la actividad de los procesos.

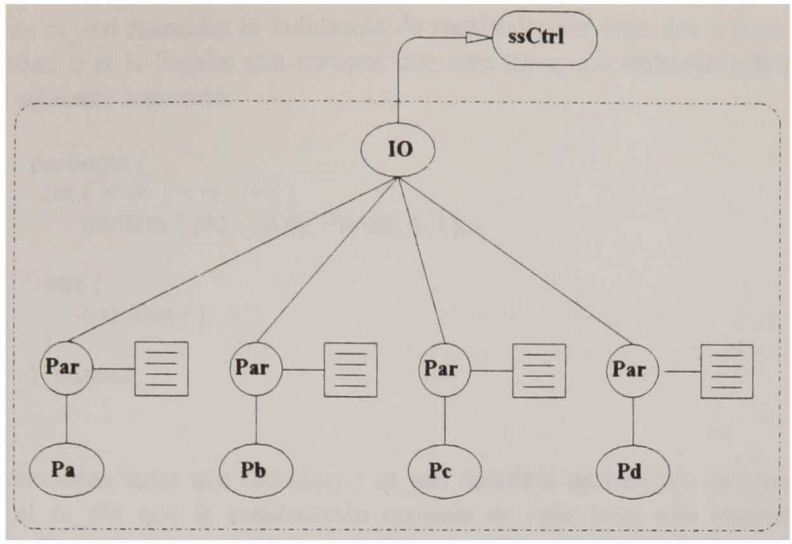


figura V.9

V.4.1.2 Condiciones de Recalendarización

Originalmente se implantó la desencalendarización haciendo esperar un mensaje a parend, sin embargo esto es muy "rígido", se verá el porque con el siguiente proceso.

```
proc procA ( Pid pid ) {

 parbegin {
  paritem ( pid, procB, Code, 0 );
  paritem ( pid, procC, Code, 0 );
  paritem ( pid, procD, Code, 0 );
  seq {
   calculosXY ( );
  }
 } parend;
     . . .
}
```

Si los hijos terminan antes que *calculosXY( )*, el slv_ctrl recibirá un mensaje que le indica que debe recalendarizar a procA. Quien se debe encargar de recibir este mensaje es *parend*, sin embargo esta será invocada sólo cuando *calculosXY( )* termine, entre tanto slv_ctrl será desencalendarizado.

Para evitar deadlocks se utilizaron semáforos para la calendarización como una técnica más "flexible" Así slv_ctrl para recalendarizar a un proceso hará un signal sobre su semáforo, pudiendo continuar su trabajo sin deber esperar por el parend.

Otro cambio en la implantación fue también con la recalendarización del padre.

Originalmente ss_ctrl mandaba la indicación de recalendarizar bajo dos circunstancias: cuando todos los hijos terminaban o si le llegaba una conspar con cero hijos, sin embargo esto provocaba problemas. Consideremos el siguiente segmento.

```
parbegin {
 for ( i= 0;  i < N;  i++ )
    paritem ( pld, _id [i], Partial, 1, i );

 seq {
    calculos ( );
 }
} parend;
```

Si los hijos terminan antes que *cálculos( )* ss_ctrl mandará un mensaje de restablecimiento y cuando parend informe al ss_ctrl que la construcción consiste de cero hijos este mandará otra indicación de recalendarización. Es decir se generarán dos signal por un sólo wait causando futuros errores.

La solución es evitar que el parend avise a ss_ctrl y no desencalendarize al padre cuando se trate de una conspar de cero hijos, sin embargo esto causa problemas con seq, para evitarlos la tabla local lleva el número total de hijos. Así cuando parend manda un mensaje al ss_ctrl la construcción al menos tiene un hijo. Asimismo parend desencalendariza al padre solo cuando el contador total es diferente de cero.

Por lo cual las condiciones para recalendarizar se reducen a una sola: la terminación de los hijos.

V.4.2. La construcción ALT

altbegin y *altend* son macros que hacen aparecer a las funciones *altBegin( )* y *altEnd ( )* como palabras reservadas.

Básicamente una construcción alt es una conspar con guardias para cada paritem:

- La función *altitem* lo primero que revisa es la condición, si es falsa termina sin ningún efecto, en caso contrario levanta una bandera, de manera que los siguientes altitem no tendrán efecto alguno.

- La función *altend* se comporta como *parend*.

La función *altresult* es simple de implantar ya que sólo hay que añadir un campo en la tabla local de proceso. Se limpia al inicio de la construcción y es marcado por el *altitem* cuyo guardia haya resultado cierto. Lo único que hace *altresult* es regresar este a valor a quien la llamó.

V.4.3. Paso de parámetros

Generalmente el hijo no es colocado en el procesador donde se declararon sus parámetros actuales por lo cual estos se tienen que almacenar, cuando los solicita a través de *params* slv_ctrl se los manda.

Con respecto al almacenamiento de los argumentos se tenian dos posibilidades: almacenarlos en el procesador del padre o en el procesador raíz, almacén padre y almacén raíz respectivamente. La segunda opción es la más adecuada, para determinar esto considerese los siguentes argumentos.

- *La sobrecarga del almacén padre es mayor que el del almacén raíz.*
  Por lo general el tamaño ocupado por los parámetros es corto, por lo que la sobrecarga de comunicaciones generalmente es mayor que el correspondiente a los parámetros.

- *Complejidad del almacén padre.*
  Para mejorar el almacén padre se deben generar canales virtuales entre todos los slv_ctrl. Sin embargo crearlos complica la programación de esta capa, incrementando su sobrecarga.

En la siguiente figura se puede observar esta complejidad revisando los canales virtuales que requiere únicamente un slv_ctrl, el número 0, en hipercubos d-2 y d-3. Se les encuentra dibujados con líneas sólidas.

Además la complejidad se incrementa al tener que manejar las condiciones de abrazos mortales en esta red con conectividad total.

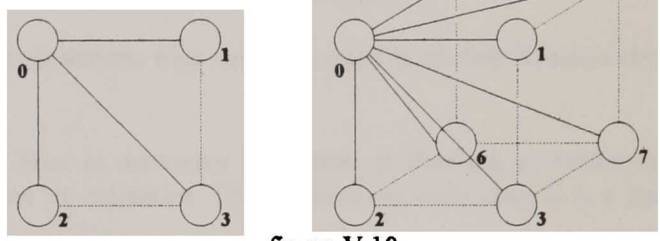


figura V.10

V.4.4. Mapeo

V.4.4.1. Mapeo según el modelo I

Al mapear a los hijos ss_ctrl busca equilibrar la carga de trabajo en base al número de procesos por procesador manteniéndolos lo más cercanos entre sí.

KHiper lleva a cabo el mapeo de acuerdo a:

- Si se proporciona el patrón de comunicaciones ss_ctrl busca una colocación óptima tomando en cuenta:

  1. Coloca al primer hijo en el procesador con menor carga de trabajo
  2. De entre los procesadores con carga mínima elige aquel que tenga la mínima distancia con el hijo ya colocado.
  3. La colocación de los restantes hijos se determina calculando la distancia promedio de comunicaciones con los procesos ya colocados. Considerando en cada caso al o a los procesadores con carga mínima.
  4. La regla de desempate cuando existen varios procesadores que ofrecen la misma distancia promedio de comunicaciones es elegir aquel que tenga el número de identificación más pequeño.

- En caso contrario el ss_ctrl asigna a cada hijo el procesador con menor carga.

Algoritmo recursivo

Se describe el procedimiento recursivo de colocación como

- Un proceso mapea a sus vecinos en el orden en como está especificado el patrón.
- Se inicia colocando al primer proceso dentro de la conspar.

Para evitar ciclos infinitos el KHiper sigue las dos siguientes reglas:

- Se verifica que un vecino no esté ya colocado antes de intentar colocarlo.
- Un proceso no intenta colocar a quien lo ha colocado.

Para ilustrar este algoritmo considerese el patrón de la siguiente figura

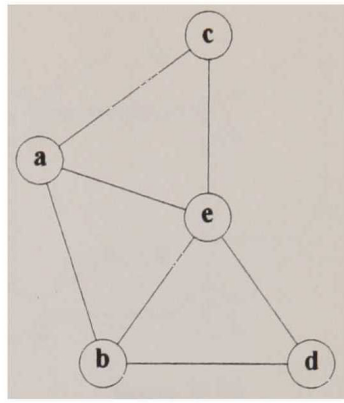


figura V.11

Sus funciones vecindad son:

ξ (a) $= \{$ b, c, e $\}$
ξ (b) $= \{$ a, d, e $\}$
ξ (c) $= \{$ a, e $\}$
ξ (d) $= \{$ b, e $\}$
ξ (e) $= \{$ a, b, c, d $\}$

El árbol generado para este mapeo es el siguiente.

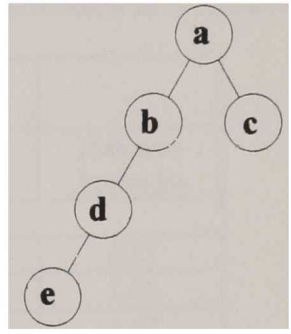


figura V.12

A continuación se puede observar como se desarrolla el algoritmo. Para simplificar considerese que al inicio todos los procesadores tienen la misma carga de trabajo: cero.

El primer proceso a ser mapeado es 'a', se elige al procesador número 0. Para colocar a 'b' se considera únicamente al canal que va al proceso 'a'. Solo falta elegir entre los procesadores 1, 2 y 4, se opta por el 1.

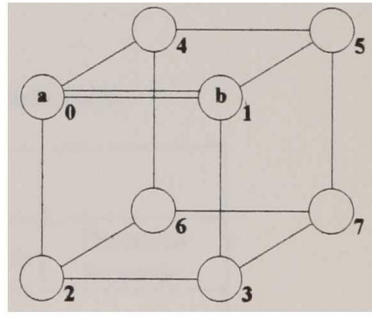


figura V.13

Para 'd' se conside sólo al canal que lo comunica con 'b'. Su distancia promedio es fácil de determinar. Se elige aquellos procesadores que se encuentran a un enlace de distancia: 3 o 5.

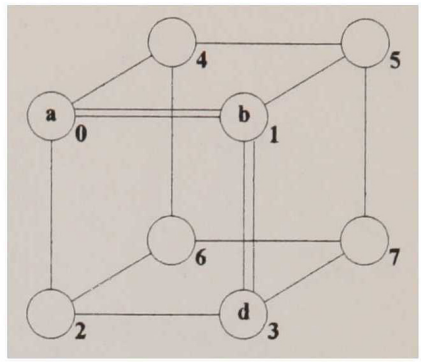


figura V.14

El siguiente es 'e'. La distancia promedio de este se basa en varios canales, no sólo en uno. Se muestran las distancias de sus canales junto con la distancia promedio para cada procesador en la siguiente tabla.

		Distancia de los canales				
Procesador Propuesto		ea	eb	ed		Distancia Promedio
2		1	2	1		1.33
4		1	2	3		2.00
5		2	1	2		1.67
6		2	3	2		2.33
7		3	2	1		2.00

KHiper coloca a 'e' en 2, continua con 'c', basándose en la siguiente tabla.

		Distancia de los canales			
Procesador Propuesto		ca	ce		Distancia Promedio
4		1	2		1.50
5		2	3		2.50
6		2	1		1.50
7		3	2		2.50

KHiper coloca a 'c' en 4.

En la figura se observa como quedan mapeados los procesos.

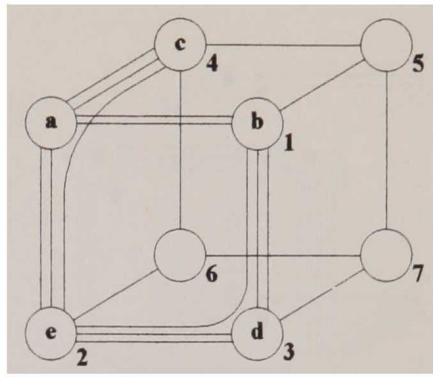


figura V.15

Elección del siguiente

La solución mas simple para obtener el siguiente proceso a colocar es tomar al que continua en la lista de conspar. Esta opción es fundamentalmente aleatoria, por lo cual no da buenos resultados.

Se puede mejorar la elección del siguiente si se toma en cuenta la estructura de la topología hipercubo.

- Se puede encontrar a un nodo que se encuentre a dos enlaces de distancia por dos caminos diferentes.

- Sean A y B dos procesadores separados por dos enlaces. Dentro de una de las intersecciones de conjuntos de procesadores adyacentes a A se encuentra a B.

  Por lo cual se debe tomar en cuenta las intersecciones de las vecindades para elegir al siguiente. De esta manera se va siguiendo el relativo ísomorfismo del patrón con el hipercubo.

Para ejemplificar la anterior afirmación considerese el siguiente patrón.

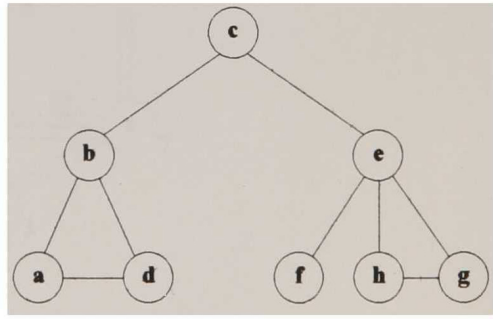
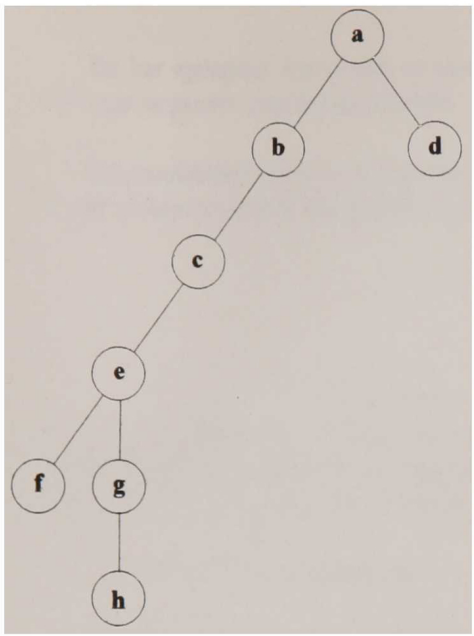


figura V.16

Con vecindades

$\xi(a) = \{ b, d \}$
$\xi(b) = \{ a, c, d \}$
$\xi(c) = \{ b, e \}$
$\xi(d) = \{ a, b \}$
$\xi(e) = \{ c, f, g, h \}$
$\xi(f) = \{ e \}$
$\xi(g) = \{ e, h \}$
$\xi(h) = \{ e, g \}$

Se ve en la siguiente figura el árbol recursivo y el mapeo logrado.



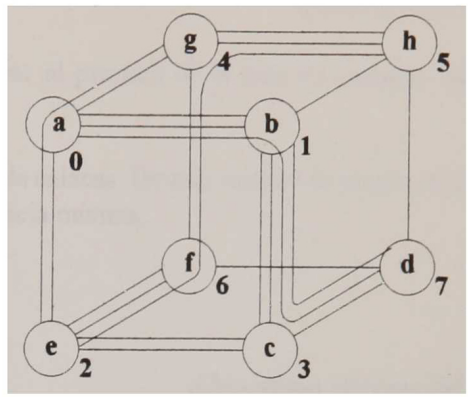
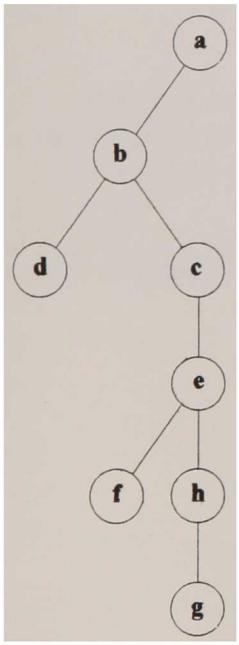


figura V.17

La distancia promedio para este mapeo tiene un valor de 1.67.

Se ve que la posición de 'd' es mala, sus dos canales son largos, se puede mejorar el mapeo, por ejemplo intentando acortar los canales de 3 enlaces de distancia.

Considerando las intersecciones, se obtiene el árbol recursivo y el mapeo siguientes.



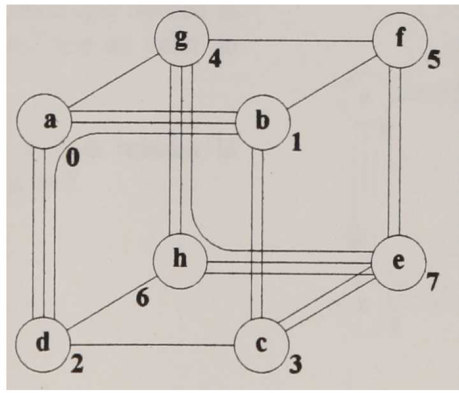


figura V.18

La distancia promedio es de 1.22. El mapeo mejoró en un 27%.

Quien primero.

En los ejemplos anteriores se tomaba como primero a mapear al primero de la lista de conspar, sin embargo se puede mejorar la elección.

Es conveniente colocar al proceso que tiene el mayor número de enlaces. De esta manera la mayor parte de sus vecinos cubrirán los procesadores disponibles con una distancia mínima.

Algoritmo secuencial.

Con este algoritmo después de colocar al primero se colocan todos sus vecinos después a los de la intersección de las vecindades, finalmente a los que aún no estén colocados. Se muestra como funciona el algoritmo sobre el patrón que se mapeo al principio.

El primero es 'e', se coloca en 0

A continuación el primero de sus vecinos 'a'. Se elige de entre los procesadores 1, 2 y 4, es colocado en 1.

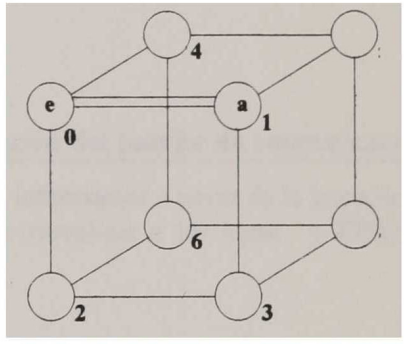


figura V.19

Para 'b' se tienen cuatro procesadores que ofrecen la misma distancia mínima, el 2, 3, 4 y 5 con un valor de 1.5. Se coloca en 2.

Para 'c', los procesadores 3, 4 y 5 le brindan la misma distancia mínima:1.5. Se coloca en 3.

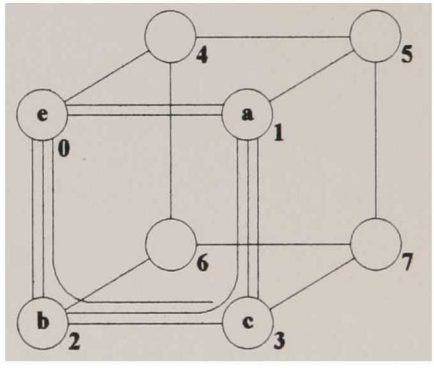


figura V.20

Para el último vecino, 'd', el 4 y el 6 muestran la misma distancia mínima, se queda en el procesador 4.

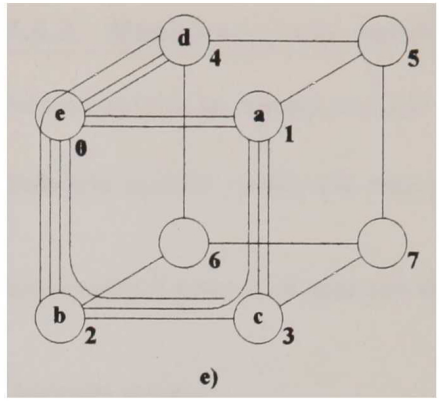


figura V.21

Adquisición del patrón de comunicaciones

Se consideró como alternativa para formar el patrón obtener su información a partir de la inicialización de los canales virtuales. Se desechó la opción dado que implica relocalizar a los hijos, y KHiper no contempla la migración de procesos.

V.4.4.2. Mapeo según el modelo II

Este algoritmo considera el tráfico de la red para calcular el tiempo que tarda un mensaje en llegar a su destino, para lo cual considera lo siguiente:

- Calcula el tiempo de entrega de un canal considerando sólo aquellos canales que comunican con procesos ya mapeados.
- Al buscar cual es el mejor procesador coloca hipotéticamente al proceso en cada uno de los disponibles.
- Utiliza el método del modelo anterior para elegir el siguiente proceso.

Se explica el algoritmo mapeando el mismo patrón que se utilizó anteriormente.

Este sin embargo tiene como diferencia que a los canales se les indica su dirección como se ve en la figura.

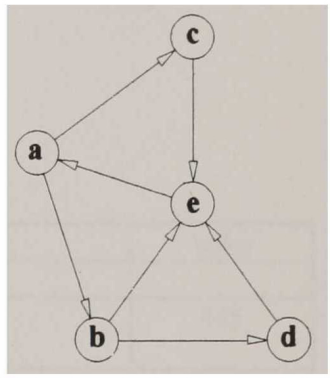


figura V.22

Sus cargas son.

l (ce) = 50 l (ac) = 50 l (ea) = 30
l (ab) = 50 l (be) = 70 l (bd) = 20
l (de) = 50

El primero a ser mapeado, por tener el mayor número de canales, es 'e'. Asignado a 0.

El siguiente a ser mapeado es 'b', el primer vecino de 'e'. Se calcula el tiempo de entrega colocándolo en cada uno de los disponibles con menor carga de trabajo: El 2, 3, ..., 7, de entre ellos los que muestran el valor más bajo son 1, 2 y 4. Se Elige a 1.

En la figura se ven colocados a 'e' y a 'b'.

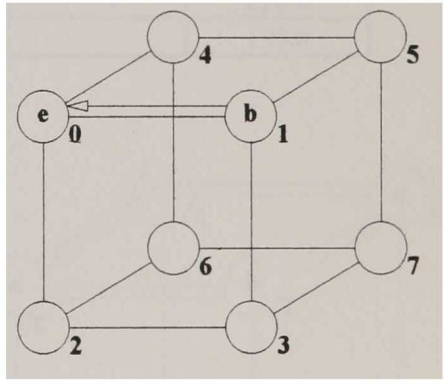


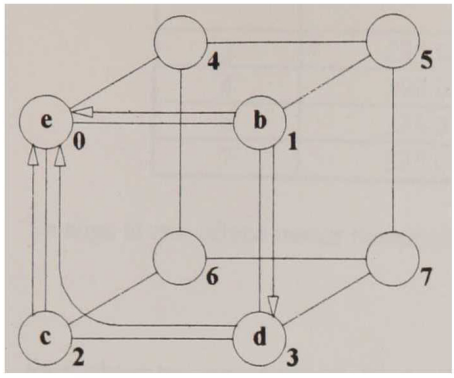
figura V.23

El siguiente a ser mapeado es 'c', el segundo vecino de e.

Los procesadores a considerar son 3, 4, ..., 7. El valor más bajo de tiempo de entrega es para 2 y 4. Se elige a 2.

En los dos casos anteriores se puede calcular el tiempo de entrega fácilmente puesto que los canales no tienen tráfico. Sin embargo con 'd' los cálculos ya no son tan simples.

En caso de ser colocado en 3 el canal 'de' se superpone a 'ce', colocado en 4 el canal 'bd' se superpone a 'be' y a 'de'. Como se ve en la figura.



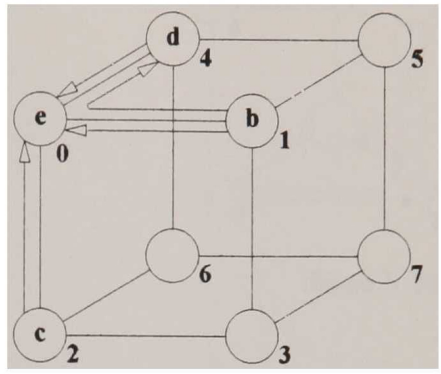


figura V.24

El tiempo propuesto de 'e'

Procesador	Expresión	Valor
3	(2 l(de) + l(ce) / 2) + l(bd) = (2 • 50 + 50 / 2) + 20	145
4	(l(de) + l(bd) / 2) + (2 l(bd) + l(be) / 2 + l(de) / 2) = (50 + 20 / 2) + (2 • 20 + 70 / 2 + 50 / 2)	160
5		120.0
6		220.0
7		190.0

Se elige al procesador 5.

En la figura se le ve junto con sus canales.

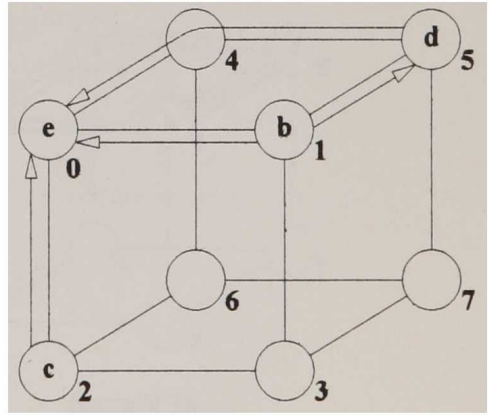


figura V.25

El último a ser mapeado es 'a', basándose en

Procesador	Tiempo de entrega
3	235.0
4	290.0
6	335.0
7	335.0

Se elige al que ofrece menor tiempo de entrega: el 3.

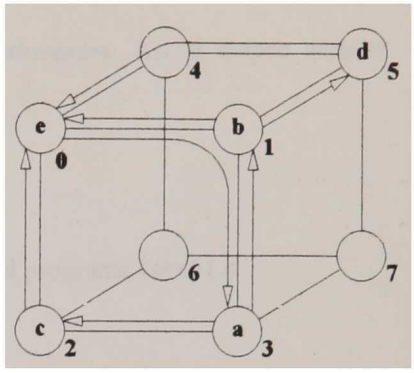


figura V.26

Para observar como para un mismo patrón ***con diferentes cargas se adquieren mapeos diferentes*** se verá el anterior patrón con dos juegos de valores diferentes.

Caso I

Si se cambia el valor del enlace 'be' a 185 se fuerza a que la penalización del canal 'ea' se incremente, debiendo ser colocado en 4.

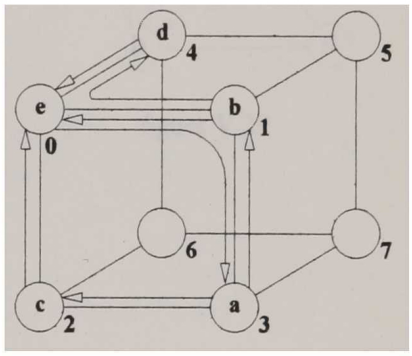


figura V.27

Caso II

Con el conjunto de valores iniciales se observa que se puede forzar a que el proceso d sea colocado en 4. Incrementando el caudal de 'de', decrementando el de 'bd' y el de 'be'.

Quedando el siguiente conjunto de valores

l (ce) = 50	l (ac) = 50
l (ea) = 30	l (ab) = 50
l (be) = 30	l (bd) = 10
l (de) = 60	

El mapeo se ve en la figura.

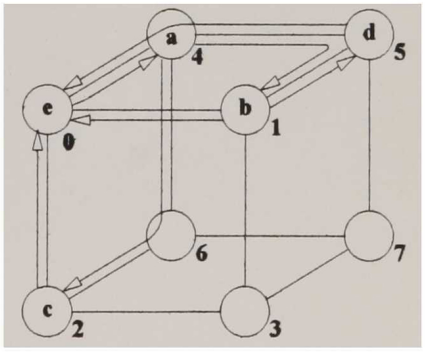


figura V.28

El modelo considera relaciones de caudales, más que valores absolutos. Así el mapeo anterior también se logra con el siguiente conjunto de valores

l (ce) = 10 l (ac) = 10
l (ea) = 6 l (ab) = 10
l (be) = 6 l (bd) = 2 l (de) = 12

Estos juegos de valores para el patrón mencionado se encuentran en el programa kerx51.c.

Considerese el patrón original con una modificación: un par de procesos comunicados por dos canales en contrasentido.

Para el modelo I el patrón se dibuja con una sola línea entre ellos, para el modelo II se deben dibujar ambos canales. Como se ve en la figura.

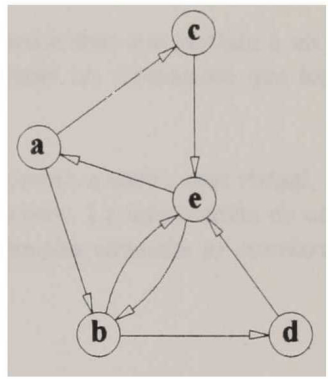


figura V.29

V.5. Control de los Canales Virtuales

Inicialización

La asignación de los canales extremos no se realiza en el padre, por lo cual él no conoce el número de canal extremo que van a usar sus hijos, tampoco conoce su colocación. Esta es la razón por la cual no les puede dar la información para llenar los campos destino de los canales extremos que van a usar.

De esta falta de información surge la necesidad de un mecanismo que permita a un proceso obtenerla. Es decir un par de procesos que se comuniquen deben de tener un mecanismo que los identifique como extremos de un canal virtual.

El mecanismo que se desarrolló consiste en asignar un número a cada canal virtual, así es como de ser un concepto pasan a ser un mecanismo de relación entre procesos. La información de cada uno de ellos la mantiene KHiper en una tabla, dado que esta es finita, ***los canales virtuales se convierten en un recurso del sistema.***

Se explica a continuación la inicialización de los canales.

La función *initchan* manda un mensaje a ss_ctrl, quien almacena el número de procesador donde se encuentra y el canal extremo que va a utilizar. A continuación espera que ss_ctrl le mande la información que necesita.

Por su parte ss_ctrl almacena la información del primer proceso que le llega, cuando le llega la información de la segunda la manda al primero y al segundo le manda la del primero.

Habilitación

La necesidad de la habilitación de un canal extremo surge de su inicialización. Observense los procesos de la figura, los cuales surgen del siguiente fragmento de programa.

```
1   proc Main (Pld pld)
2   {
3         . . .
4    getchan (& c0);
5    getchan (& c1);
6
7    parbegin {
8     paritem (pld, a, CodePa, 1, c1, c0);
9     paritem (pld, b, CodePb, 1, c0, c1);
10   } parend;
11        . .
12  }
13
14  proc Pa (Pld pld, Vchan c1, Vchan c0)
15  {
16  Echan  in, out;
17
18   . . .
19   chaninit (& in, c1);
20   chaninit (& out, c0);
21   . . .
22  }
23
24  proc Pb (Pld pld, Vchan c0, Vchan c1)
25  {
26  Echan  in, out;
27
28   . . .
29   chaninit (& out, c1);
30   chaninit (& in, c0);
31   . . .
32  }
```

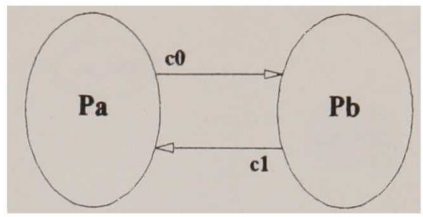
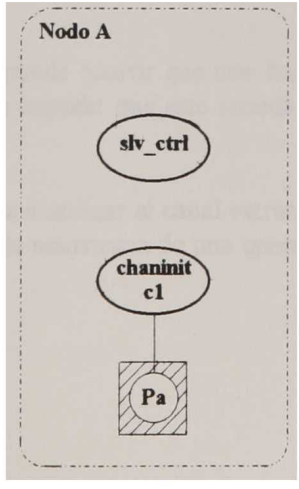


figura V.30

Para entender las dificultades que implica considerese una hipotética secuencia de pasos para lograrlo.

Después de que Pa llama a chaninit, ss_ctrl llena los campos de canal extremo y procesador destino de canal virtual c1.

Pa es desencalendarizado vía chaninit, como se ve en la figura.



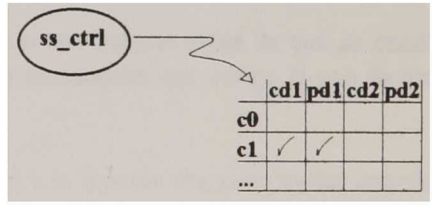
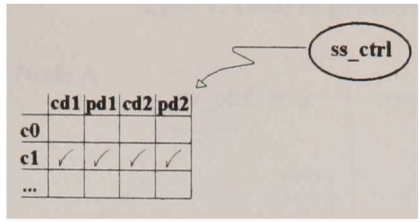


figura V.31

Posteriormente cuando Pb llama a chaninit, para inicializar a out, ss_ctrl termina de llenar los campos de c1 y le manda.

Pb es desencalendarizado vía chaninit, como se ve en la figura.



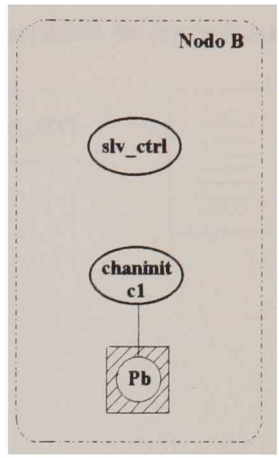


figura V.32

ss_ctrl entrega su información a Pa y posteriormente a Pb. En ese momento el proceso de inicialización de c1 queda completo.

De acuerdo a esto la secuencia de actividades presentada es la adecuada para inicializar un canal virtual, sin embargo **si en el programa anterior intercambio las líneas** 29 y 30, el conjunto de actividades para inicializar el canal virtual falla y **el sistema entra en deadlock**. Se verá porque.

Cuando Pa llama a chaninit ss_ctrl llena los primeros campos de c0 y los de c1 cuando Pb haga lo mismo. Después de esto Pa y Pb son desencalendarizados vía chaninit.

Y aqui es donde realmente se encuentra el problema ya que para que chaninit de la línea 19 termine ss_ctrl debe de terminar de llenar la entrada de c1, pero ello requiere de chaninit de la línea 30, quien a su vez requiere la terminación de la función de la línea 29. Lo mismo pasa con la chaninit de la línea 29. Y el sistema entra en deadlock.

De donde ***chaninit debe mandar la información de su canal extremo sin esperar por su información destino.***

Sin embargo, con esta solución puede ocurrir que una función in o out ocurran antes de que su canal extremo esté inicializado. Se necesita impedir que esto suceda con un mecanismo que inhiba el uso de los canales extremos.

Después de que el proceso que va a utilizar el canal extremo llamó a la función chaninit pueden ocurrir dos situaciones de acuerdo al orden de ocurrencia de una operación de comunicación y la inicialización del canal.

Caso I: Ocurre primero la operación de comunicación.

Cuando esta operación determina que su canal no está habilitado pasa a esperar un mensaje de permiso.

Posteriormente cuando se inicialize el canal, slv_ctrl creará a postenable. Quien mandará el mensaje de habilitación del canal, después de marcar el campo de habilitación. Como se ve en la figura.

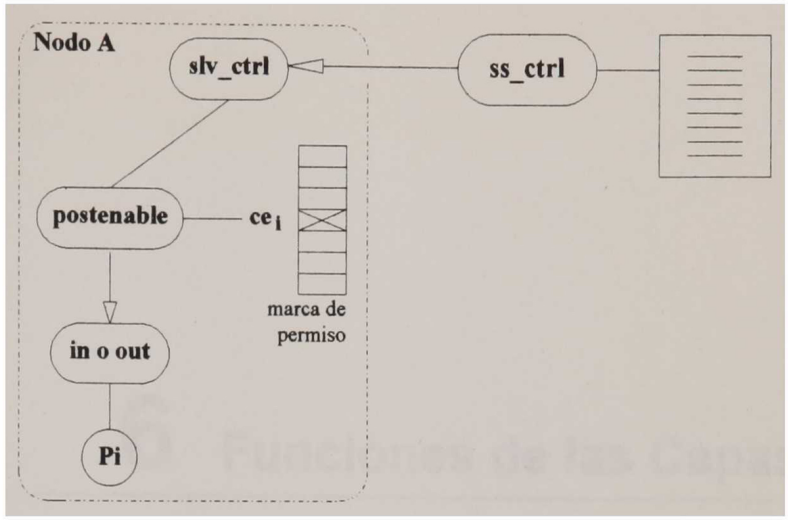


figura V.33

Caso II: Ocurre primero la inicialización.

slv_ctrl crea a postenable quien marca el campo de habilitación, después espera para mandar su mensaje de habilitación. Posteriormente la operación de comunicaciones aceptará el mensaje.

La creación de postenable como proceso es necesaria para impedir que slv_ctrl sea quien tenga que esperar para mandar el mensaje de habilitación, lo que podría llevar a un deadlock.

V.6. Mandar y recibir información a pantalla y teclado.

El compilador ofrece los procesos mux [9] cuando se desea que un procesador adicional al raíz acceda a las facilidades de la computadora anfitrión, sin embargo dado que cada mux exige el uso exclusivo de un enlace del procesador la dimensión del hipercubo se ve limitada a 2.

KHiper es monousuario, esto implica que por lo general, los datos son distribuidos por la raíz. Es raro encontrar un programa en el que varios procesos soliciten información, sin embargo dado que puede suceder KHiper-control ofrece una serie de funciones de adquisición de datos por teclado y funciones para mandar información a pantalla.

6 Funciones de las Capas

VI.1. Descripción de las funciones de la capa de comunicaciones

Familia altwait

Use esta familia para encontrar, si existe, cual de un conjunto de canales extremos está listo para comunicarse. Sí más de un canal está simultáneamente listo para comunicarse, será elegido el primero en la lista.

Parámetros a usar:

EChan echan0, EChan echan1, ..., son los canales a probar.
vector es un arreglo canales extremos a ser probados.
hMany es el número de canales a probar.

Diferencias:

1 Solo regresa cuando al menos uno de los canales esté listo para comunicarse.
2 Regresa un valor negativo cuando ningún proceso está intentando mandar.
3 Regresa el número de canal (echan0, echan1, ...) que está listo para comunicarse.
4 Regresa un valor entre 0 y hMany-1, indicando quien en la lista está listo.

		1	2	3	4
EChan altnowait	(int hMany, EChan echan0, EChan echan1, ...)		√	√	
EChan altnowaitI	(int hMany, EChan echan0, EChan echan1, ...)		√		√
int altnowaitVec	(int hMany, EChan *vector)		√	√	
int altnowaitVecN	(int hMany, EChan *vector)		√		√
EChan altwait	(int hMany, EChan echan0, EChan echan1, ...)	√			
int altwaitI	(int hMany, EChan echan0, EChan echan1, ...)	√			√
int altwaitVec	(int hMany, EChan *vector)	√			
int altwaitVecN	(int hMany, EChan *vector)	√			√

Familia broadcast

Manda un mensaje sobre un conjunto de canales. Es más poderosa que un conjunto de outmess, que se ejecutan secuencialmente, ya que funciona concurrentemente.

Parámetros a usar:

hMany indica el número de canales a los cuales se les manda el mensaje.
EChan echan0, EChan echan1, ..., son los canales por los que se va a mandar.
vector es un arreglo canales por los que se va a mandar.
size es número de bytes del mensaje

Diferencias:

1 Espera a que todos los canales hayan establecido su comunicación.
2 Manda un mensaje del tamaño de un word

		1	2
void broadcast	(int size, char* data, int hMany, EChan echan0, ...)		
void broadcastvec	(int size, char* data, int hMany, EChan *vec)		
void broadcastvecwait	(int size, char* data, int hMany, EChan *vec)	√	
void broadcastwait	(int size, char* data, int hMany, EChan echan0, ...)		
void broadcastword	(int data, int hMany, EChan echan0, ...)		√
void broadcastwordVec	(int data, int hMany, EChan *vec)		√
void broadcastwordVec	(int data, int hMany, EChan *vec)	√	√
void broadcastwordVecwait	(int data, int hMany, EChan *vec)	√	√

Familia guardin

Lee un mensaje sólo en caso de que el out correspondiente este listo, en tal caso regresa 1. En caso contrario regresa un 0.

int guardinmess (EChan exch, char* data) Leé un mensaje de un word de tamaño.
int guardinword (EChan exch, int * data) Leé un mensaje de cualquier tamaño.

Parámetros a usar:
exch canal extremo por el que se leé el mensaje.
data variable en donde se leé la información.

inhibit

Revoca la habilitación al uso del canal extremo exch.

void inhibit (EChan exch)

Familia in

Lee un mensaje por el canal extremo indicado. Desencalendariza al proceso que la llamo hasta que la comunicación esté lista.

int inmess (EChan exch, char* data) Regresa la longitud del mensaje.
int inword (EChan exch, int *info) Regresa el valor leído.

Familia out

Manda un mensaje por el canal extremo especificado. Desencalendariza al proceso que la llamo hasta que la comunicación esté lista.

void outmess (EChan exch, int size, char* data)
void outword (EChan exch, int data)

Familia set

Establece los campos destino y habilita su uso.

Parámetros a usar:
exch Canal extremo local.
dnode Procesador destino.
dChan Canal extremo destino para el canal extremo exch.

Diferencias:
1 Establece los campos destino
2 Habilita el uso del canal.

	1	2
void setAndEnable (EChan exch, int dnode, int dChan)	√	√
void preEnable (EChan exch)		√
void setEnd (EChan exch, int dnode, int dChan)	√	

VI.2. Descripción de las funciones de la capa de control

chaninit

Inicializa y habilita un canal extremo para formar parte de uno virtual.

Parámetros a usar:
pexchan Canal extremo a inicializar.
vchan Canal virtual que sirve para inicializar.

void chaninit (EChan *pexchan, VChan vchan)

Ejemplo en: kerx18.c

Familia getchan

Solicita canales virtuales.

Parámetros a usar:
pchan Variable del tipo de un canal virtual.
hmany Número de canales solicitados.
pvchan0 Apuntador del canal donde será almacenados el valor regresado.

void getchan (VChan *pchan)	Consigue un sólo canal.
void getchanarray (int hmany, VChan *pchan)	Consigue un arrgelo de canales.
void getchans (PId pId, int hmany, VChan * pvchan0, ...)	Consigue una lista de canales.

Ejemplos en: kerx18.c, kerx22.c, kerx21.c

Familia get

Lee de la entrada estandard.

Las funciones con terminación str imprimen una leyenda en pantalla solicitando la información. La cadena a imprimir se encuentra almacenada en la variable __Screen__, para usarla se debe declarar previamente la variable screen del tipo tscr.

pbyte Apuntador a char.
pdouble Apuntador a doble.
getfloat Apuntador a flotante.
pstring Apuntador a char.
pint Apuntador a word.
size Indica el tamaño del mensaje.

Función		Ejemplo en
getbyte	(char *pbyte)	*kerx12.c*
getbytestr	(char *pbyte, int size)	*kerx38.c*
getdouble	(double *pdouble)	*kerx38.c*
getdoublestr	(double *pdouble, int size)	*kerx38.c*
getfloat	(float *pfloat)	*kerx38.c*
getfloatstr	(float *pfloat, int size)	*kerx38.c*
getstring	(char *pstring)	*kerx38.c*
getstringstr	(char *pstring, int size)	*kerx38.c*
getword	(int *pint)	*kerx38.c*
getwordstr	(int *pint, int size)	*kerx38.c*

Familia newpid

Solicita un o un grupo de identificadores de proceso.

hmany	es el número de identificadores solicitados.
pointerpid	Apuntador a identificador de proceso.
pointerpids	Apuntador al arreglo a identificadores de proceso.
ppid0, ppid1	Apuntadores a identificador de proceso.

void newpid	(PId *pointerpid)	Solicita un identificador.
void newpidarray	(int hmany, PId *pointerpids)	Solicita un arreglo.
void newpids	(PId pId, int hmany, PId * ppid0, . . .)	Solicita una lista.

hmanybytes

Informa de la cantidad de información efectuada por un enlace.

int hmanybytes (int nodo, int enlace, int option)

Los primeros dos parámetros indican de quien se requiere la información.

enlace toma el valor 0 para X, 1 para Y y 2 para Z.

option 0: información de la capa de comunicaciones
1: información de la capa de la capa de control.
2: suma de ambas.

Ejemplo en: kerx37.c

toScreen

Manda una leyenda a la pantalla, de size bytes, almacenado en la cadena apuntada por str.

toScreen (PId ppid, int size, char *str)

Este macro manda una leyenda a la pantalla.

toscreen (SCR, (exp))

exp es la expresión que se le pasa a una función printf. SCR es el nombre de un macro.
Se debe declarar previamente la variable screen del tipo tscr.

Ejemplo en: kerx01.c

VI.3. Macros usados en los programas del usuario.

El uso de macros permite a KHiper ocultar algunos detalles de implementación, acerca la escritura de un programa a la sintaxis propuesta, mejora la estética y libera el trabajo del usuario de algunos detalles innecesarios.

Para aquellos interesados en consultar el código de KHiper se muestran los macros usados.

```
#define proc                        void

#define mynode()                    node
#define mypid()                     pId

#define params(a)                   loadArgs   (pId, (char*) a)
#define end()                       _finish          (pId)
#define parbegin                    parBegin   (pId);
#define parend                      parEnd     (pId);
#define paritem                     parElement
#define seq                         sequen           (pId);
#define placeat(proc,slave)         placeAt    (pId, proc, slave)

#define altbegin                    altBegin   (pId);
#define altend                      altEnd     (pId);
#define altitem                     altElement
#define altplace(place)             altPlace   (pId,place)
#define altresult                   altHowMany (pId)

#define getpid(dato)                getPid     (pId, dato)
#define getpids                     getPIds
#define getpidarray(num,array)      getPIdArray    (pId, num, array)

#define getchan(a)                  getChan    (pId, a)
#define getchans                    getChans
#define getchanarray(num,array)     getChanArray(pId, num, array);
#define chaninit(a,b)               chaninit   (pId, a,b)

#define neigh                       eVec
#define distprom()                  (float)dpc (pId)/(float)DpcFactor
#define bestmap()                   bestMap    (pId)

#define altwait                     altwait
#define altnowait                   altnowait
#define altwaiti                    altwaitI
#define altnowaiti                  altnowaitI
#define altwaitvec                  altwaitVec
#define altnowaitvec                altnowaitVec
```

```
#define altwaitvecn          altwaitVecN
#define altnowaitvecn        altnowaitVecN

#define outWord              outword
#define inWord               inword
#define outMess(n,s,d)       outmess(n,s,(char*)d)
#define inMess(n,d)          inmess(n,(char*)d)

#define broadcastword        broadcastWord
#define broadcastwordVec     broadcastWordv
#define broadcastmess        broadcast
#define broadcastmessv       broadcastVec

#define hmanybytes           hMBytes
#define toscreen(size)       toScreen (pId, sprintf size, __Screen__)
#define tscr                 char
#define screen               __Screen__[SIZESCREEN]
#define SCR                  __Screen__

#define getbyte(_byte)       getByte    (pId, _byte)
#define getword(_int)        getword    (pId, _int)
#define getfloat(_float)     getFloat   (pId, _float)
#define getdouble(_double)   getDouble  (pId, _double)
#define getstring(string)    getString  (pId, string)

#define getbytestr(_byte, size)      getByteS   (pId, _byte,   size, __Screen__)
#define getwordstr(_int,  size)      getwordS   (pId, _int,    size, __Screen__)
#define getfloatstr(_float, size)    getFloatS  (pId, _float,  size, __Screen__)
#define getdoublestr(_double,size)   getDoubleS (pId, _double, size, __Screen__)
#define getstringstr(string, size)   getStringS (pId, string, size, __Screen__)

#define Vchans(name,num)     Chan name[num]; getchanarray (num,name)
#define Pids(name,num)       PId  name[num]; getpidarray  (num,name)
```

La inclusión del identificador del proceso se eliminó con los macros, en la mayor parte de funciones, sin embargo dado que en C no existen macros con número variable de parámetros fue necesario transferir esta responsabilidad al usuario.

7 Conclusiones y Resultados

VII.1 Resultados

Archivos donde se encuentran las capas de KHiper:

- La de comunicaciones en el archivo ipc.c.
- La de control:
  a) Los procesos que corren en todos los procesadores en el archivo kern.c
  b) Los que se ejecutan únicamente en la raíz en kerr.c

Programas de prueba.

De la capa de comunicaciones:	37 con nombres de archivo x???.c
De la capa de control:	51 con nombres de archivo kerx??.c

Archivos de ejecución por lotes.

Estos se proporcionan para hacer transparente el sistema al usuario.
Se diseñaron versiones para trabajar

- Con hipercubos de orden 1, 2 y 3.
- Para trabajar con hipercubos corriendo en un solo procesador.

VII.2.1 Graficación de autómatas celulares

Los autómatas celulares cada vez encuentran más aplicaciones, por lo cual es conveniente acelerar el tiempo de ejecución de los programas que se usan para graficarlos con el uso del transputer y aún más con una red de ellos.

Cuando se tienen pocos hipercubos y se desea generar las gráficas para toda la familia de un autómata celular no es práctico acaparar uno de ellos para observar una por una, lo más conveniente es generar todas, almacenarlas y posteriormente revisarlas en otra computadora.

Como una aplicación del KHiper implanté:

- Un programa que genera las gráficas de una familia de autómatas celulares y las almacena en archivos.
- El software encargado de mostrarlas en la PC.

El programa generador de gráficas utiliza los canales virtuales y las construcciones par y seq de KHiper-Control, además muestra como separarlo en dos archivos, requisito cuando se quieren utilizar facilidades de la PC no apoyadas por KHiper, como el acceso a archivos.

El proceso principal se encarga de generar las máscaras que determinan a cada miembro de la familia y comunicarlas a cuatro procesos, cada uno de los cuales va a calcular una cuarta parte de la gráfica. Cada proceso se encuentra en un procesador diferente.

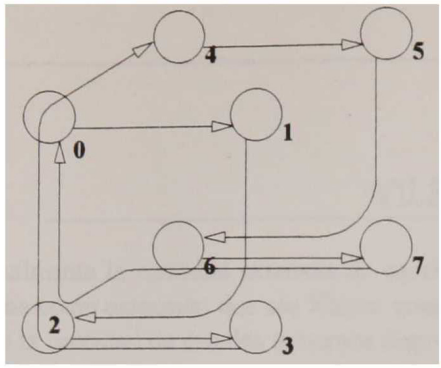
Este programa es una simple introducción al uso del hipercubo para graficación de autómatas celulares. Por lo cual adolece de restricciones, como el tipo de autómata, la dimensión del hipercubo y el tipo de autómata, ya que está diseñado para evitar pasar información de intersecciones.

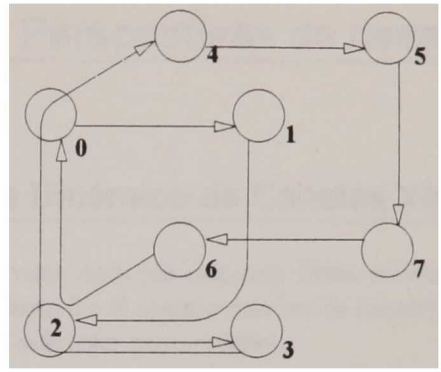
El programa se encuentra en los archivos *kerx36r.c y kerx36n.c.*

VII.2.1 40,320 anillos en un hipercubo d-3

Este programa se encarga de recorrer el hipercubo en forma de anillo visitando cada procesador una sola vez. Calcula cada uno de los posibles anillos, recorriendo uno diferente en cada ocasión. Cada anillo es una permutación de la secuencia 01234567, por lo cual recorre los $(2^{dim})!$ anillos posibles.

En cada procesador se crea un proceso, para formar parte del anillo. En esta figura se observan las dos primeras rutas a recorrer en el hipercubo d-3.





En ciertos casos el uso de funciones recursivas en programas concurrentes facilita el trabajo, por ejemplo para calcular permutaciones.

En este programa se van a ejecutar simultáneamente los procesos *hoja_* y *startEnd*. El primero se encarga de crear la ruta siguiente y el segundo de pasarlo a los procesos que forman el anillo, quienes entonces conocen el proceso de quien van a recibir el mensaje y a quien mandarlo.

*hoja_* llama recursivamente a *hoja*, quien comunica la ruta que ha calculado.

Cada vez que *startEnd* necesita la ruta siguiente la espera de *hoja*.

El trabajo de *startEnd* y de *hoja* está sincronizado por el paso de mensajes, de manera que *hoja* calcula una nueva ruta hasta que *startEnd* acepte la actual.

Este programa se encuentra en *x11x.c*. En *x10x.c* se muestra la solución secuencial.

VII.3 Perspectivas de desarrollo

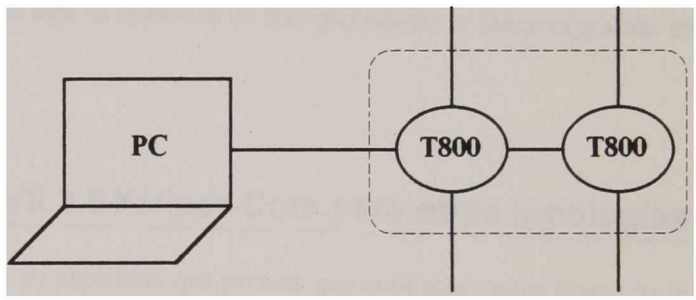
VII.3.1 Límite Dinámico de Canales Virtuales

Actualmente la cantidad existente de canales virtuales viene dada por una cota fijada arbitrariamente. Se propone como extensión que sea Khiper quien dinámicamente fije el número máximo de canales virtuales revisando la cantidad de canales extremos disponibles en los diferentes procesadores.

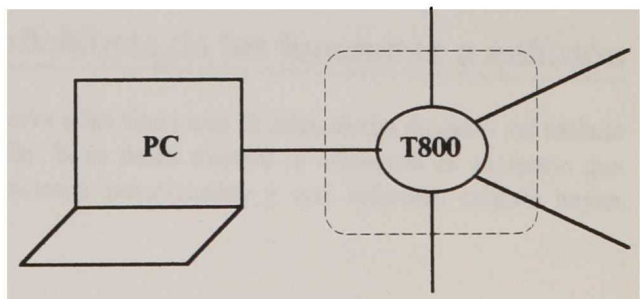
VII.3.2 Aumentar la dimensión de KHiper a orden 4

El transputer conectado a la computadora anfitrión utiliza uno de sus enlaces para este fin de manera que para comunicarse con los demás transputer sólo cuenta con 3 enlaces. Lo anterior limita la dimensión del hipercubo a una dimensión máxima de 3 ya que el orden o dimensión del hipercubo es el número de enlaces del que deben disponer todos y cada uno de los procesadores de la topología.

La solución que se propone es substituir el transputer raíz por un par de transputers. Este par substituto realiza las funciones del raíz original siendo su único objetivo el de incrementar el número de enlaces del que se puede disponer en la raíz.



Las modificaciones que se deberán realizar sobre la capa de comunicaciones consiste en pasar la información entre los dos procesadores simulando uno sólo, con lo cual los procesadores adyacentes no verán sino un sólo procesador. Y el par substituto se comportará como un procesador de cinco enlaces.



La capa de control no se modificará, y KHiper podrá funcionar, entonces, con una dimensión de 4, es decir con 16 procesadores.

VII.3.3 Implementar KHiper sobre un hipercubo de 80x86

KHiper funciona utilizando las funciones que proporciona el compilador 3L, quien a su vez utiliza las características del kernel del transputer. Si se crea para un procesador de la familia 80x86 la capa que proporcione las mismas funciones de concurrencia y comunicaciones que las proporcionadas por el compilador 3L se puede utilizar el código de KHiper para implantar un hipercubo formado por este tipo de procesadores.

VII.3.4 Migración de procesos

Un proceso recalendarizado por KHiper continua su ejecución en el procesador donde fue desencalendarizado, sin considerar la carga de cada procesador con lo cual perdemos la oportunidad de mejorar la eficiencia.

En los modelos que se desarrollaron para el mapeo no se considero la posibilidad de que la cantidad de comunicaciones entre dos procesos cambiara con el tiempo, por lo cual en ocasiones es conveniente relocalizar a un proceso a un procesador que favorezca las comunicaciones

Al hecho de cambiar a un proceso para que siga su ejecución en otro procesador se llama migración de procesos.

VII.3.5 KHiper-Com para otras topologías

KHiper para mandar un mensaje recurre a un algoritmo que permite que cada procesador dentro de la ruta determine a quien le va mandar la información. Esta decisión las toma en cuenta cada procesador.

Se puede hacer funcionar KHiper sobre otras topologías si se crean los algoritmos de ruteo correspondientes, esto en especial es simple para topologías hipercúbicas.

VII.3.6 Incrementar la eficiencia de las funciones a anfitrión

En este momento el ss_ctrl les de atención exclusiva a las funciones de adquisición de datos del teclado y a las utilizadas para mandar información a pantalla. Si se desea mejorar la eficiencia es necesario que KHiper genere procesos que lleven a cabo las funciones mencionadas y que informen cuando hayan terminado.

VII.3.7 Acceder archivos desde cualquier procesador

En este momento solo los procesos corriendo en el procesador raíz son capaces de acceder a los servicios de la PC. Otra extensión que se propone es aumentar las funciones de KHiper-Control de manera que cada slv_ctrl funcione como un proceso mux proporcionado por el compilador.

VII.3.8 Convertir KHiper multiusuario

Considerando las ventajas que ofrece se propone hacer a KHiper multiusuario.

VII.3.9 Mejorar las funciones vecindad

En la manera en que se especifican las vecindades existe información redundante con el consecuente incremento de trabajo para el programador. Es necesario plantear formas alternativas de describir los patrones de comunicación ó modificar las funciones existentes para ser más eficiente el trabajo del usuario.

VII.3.10 Implementar métodos para simular memoria compartida

KHiper tiene la limitación de no proporcionar mecanismos para utilizar memoria compartida, ya sea EREW o CREW. La cual es conveniente implementar sobre KHiper.

VII.3.11 Medir el overhead causado por las capas de KHiper

Para analizar el funcionamiento de la capa de comunicaciones KHiper ofrece varias funciones, sin embargo no las proporciona para la capa de control. Como extensión se propone que se añadan estas funciones.

VII.3.12 Mapeo estocástico

Una alternativa que se está investigando intensamente es el mapeo estocástico. KHiper coloca los procesos de manera determinística con las desventajas consecuentes.

Una extensión importante es desarrollar un algoritmo de mapeo estocástico para KHiper.

VII.3. Contexto de programación

A continuación se mencionan algunas características del compilador, del configurador y del encadenador que crean el contexto sobre el cual se desarrolla este trabajo.

Una aplicación es una colección de uno o más tareas ejecutándose concurrentemente. Cada tarea esta formada por una región propia de código y datos así como un vector de puertos de entrada y uno de salida. Las tareas solo se comunican mediante paso de mensajes.

Después de escribir el código los pasos para lograr una aplicación son compilarlo, encadenarlo y configurar la red, para cada uno de estos pasos se utiliza la respectiva utileria.

Una aplicación requiere un archivo de configuración. En el se indica el hardware sobre el cual se está trabajando, las tareas que la conforman, los procesadores sobre los cuales van a ser ejecutadas y la forma en como están conectadas.

El archivo de configuración es quien especifica la topología sobre la que se ejecuta un patrón de comunicaciones.

Se puede lograr que una aplicación con un patrón específico corra en un solo procesador o en una red, cambiando únicamente su archivo de configuración. Sin necesidad de compilar y encadenar nuevamente cada una de las tareas.

Por lo cual se puede trabajar una topología aun sin tener todos los procesadores necesarios, cuando se tengan todos, simplemente se cambia el archivo de configuración.

Además de la tarea existe otro tipo de proceso concurrente, el thread (hilo), que algunos textos mencionan como proceso ligero (ligthweigth proces).

Las tareas pueden crear hilos. Los hilos creados por una tarea comparten datos, código y memoria y pueden comunicarse vía memoria compartida o por paso de mensajes.

Se pueden colocar una o varias tareas por procesador. Los hilos sólo donde se encuentra la tarea que los creó.

Se desarrolló KHiper considerando cada nodo del hipercubo como una tarea. ***Los procesos que forman a KHiper junto con los procesos del usuario son hilos (threads)***. Basándose en:

- Evitarle al usuario tener que generar un archivo de configuración para cada aplicación.
- Que la topología y KHiper sean transparentes.
- Una actividad de KHiper es el mapeo de los procesos del usuario, para desarrollar un mapeo dinámico se requiere de información a tiempo de ejecución. Por lo cual los procesos del usuario no deben ser tareas.

- Para lograr el control de los procesos de acuerdo a las construcciones del modelo c.s.p. los procesos del usuario deben ser hilos.

Una tarea es una función principal, main en el lenguaje C. El código de un hilo se especifica como el de una función que regresa nada. El código de la tarea y de los hilos que va a crear se proporcionan en un mismo archivo D.O.S. Para n de tareas idénticas solo se necesita un archivo ejecutable.

Para lograr mapeo dinámico KHiper replica el código de los procesos del usuario en todos los procesadores, lo que se llama código simétrico.

El compilador proporciona funciones para la creación de hilos y para el control de ellos las funciones para desencalendarisarlos, para recalendarisarlos y para finalisarlos.

El uso de los semáforos está restringido a una sola tarea, esto significa que el ámbito de los hilos creados por una tarea no se extiende más alla de un procesador.

La manera más cómoda para realizar el control de un hilo es utilizando semáforos, sin embargo, cuando están alojados en tareas diferentes sólo se puede realizar mediante paso de mensajes, esto implica que KHiper debe llevar el control mediante esta segunda técnica.

VII.4 Transparencia del sistema

Para lograr una aplicación se deben crear los archivos de cada tarea diferente dentro del sistema, el archivo de configuración y es conveniente un archivo bat donde se especifique los diferentes archivos fuente a ser compilados y los diferentes modos de encadenamiento.

KHiper evita al usuario el preocuparse por desarrollar todos esos archivos.

VII.5 Ventajas y desventajas

Las ventajas que KHiper ofrece son:

- Extensiones de concurrencia al lenguaje C
- Uso del modelo c.s.p.
- Disminuye drásticamente el tiempo de programación del hipercubo
- Permite el desarrollo de código portable
- Elimina dependencia de la arquitectura
- Lo anterior da abstracción al programador
- Genera la posibilidad de usarlo en:
  - Otras topologías
  - Sobre un hipercubo de procesadores 80x86
- Mejora la eficiencia de algunos programas

Las desventajas que KHiper presenta son:

- Una sintaxis que presente una mejor aproximación a c.s.p. requiere de un precompilador, el cual no está contemplado dentro de mi trabajo.
- KHiper-Control tiene la desventaja de presentar un "cuello de botella" provocado por el hecho de funcionar como una capa con control centralizado.
- Requiere de una forma incómoda el patrón de comunicaciones
- Los procesos de la construcción par solo se ejecutan hasta el fin de la declaración de la misma
- Para usar un canal se requiere declararlo y conseguirlo, sería más cómodo que un proceso se comunicara con otro simplemente nombrándolo.
- No tiene manera de saber cuando la comunicación por un canal se intensifica, con esa información se pueden relocalizar los procesos para incrementar la eficiencia.

VII.6 Conclusiones

Después de programar utilizando KHiper se pueden apreciar sus ventajas y se observa como las perspectivas de uso del paralelismo se ven incrementadas con una herramienta como esta, sin embargo también resalta la necesidad de desarrollar un precompilador, mejorar la manera de proporcionar el patrón de comunicaciones y disminuir el código referente a los canales.

Dentro de los alcances de este trabajo no están las mediciones del rendimiento de KHiper, lo cual requiere probarlo con una amplia gama de programas, de manera que solo su amplio uso puede indicar su eficiencia.

Algo que destaca notablemente es el hecho de que los horizontes de desarrollo de esta herramienta son amplios, dentro de este panorama esta herramienta se sitúa como una aportación a la solución de los problemas implícitos dentro de este paradigma de programación.

[1] C.A.R. Hoare.
Communicating Sequential Processes.
Communications of the ACM, August 1978. Volume 21. Number 8.

[2] Angel L.DeCegama
The Technology of Parallel Processing
Prentice Hall.

[3] Ronald S. Cok
Parallel Programs for the Transputer
Prentice Hall.

[4] F.Thomson Leighton
Introduction to Parallel Algorithms and Architectures
Morgan Kaufmann Publishers

[5] Hwang and Briggs.
Computer Architecture and Parallel Processing.
McGraw Hill.

[6] Ian Grahan and Tim King
The Transputer Handbook
Prentice Hall.

[7] Franco P. Preparata and Jean Vuillemin
The Cube-Connected Cycles: A Versatile Network for Parallel Computation.
Communications of the ACM, May 1981. Volume 24. Number 5.

[8] Yaron Wolfstahl
Mapping Parallel Programs to Multiprocessors: A dynamic approch.

[8] M.M.Flynn
"Some Computer Organizations and their Effectiveness"
IEEE Transactions on Computers, C-21, No. 9 (Sept. 1972) pp. 948-60

[9] 3L Ltd.
Parallel C User Guide, Version 2.2.2.

[10] Erol Gelenbe.
Multiprocesor Performance.
John Wiley and Sons

[11] Logical Systems
Manual del compilador.

[12] Andrew S.Tanenbaum
Sistemas operativos Modernos.
Prentice Hall.

[13] Derek L. Eager, Edward D. Lazowska, and John Zahorjan.
Adaptive Load Sharing in Homogeneus Distributed Systems
IEEE Transactions on software Engineering, Vol. SE-12, No. 5, May 1986.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará el **Ing. Gustavo Sergio Téllez Rangel**, declaramos que hemos revisado la tesis titulada:

"Kernel escalable para una red de transputer en topología hipercubo"

y consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente

M. en C. José Oscar Olmedo Aguírre

Dr. Sergio V. Chapa Vergara

Dr. Guillermo Benito Morales Luna

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
Sofia	3/3/97	7/3/97
Fotocopiado - Rep Tec.	2 may 97	6 may 97