ARQUITECTURA DE BUS DE MEMORIA CONTROLADA POR DEPENDENCIAS DE TAREAS

M. Nussbaum *

A. Cipriano **

RESUMEN

Se presenta un modelo MIMD basado en la idea de arquitecturas controladas por dependencias de datos. El modelo consta de una red que no requiere conmutación si no $O(n^2)$ memorias, donde n es el número de subsistemas que forman el sistema. Se caracteriza por permitir una ganancia de velocidad de procesamiento lineal con el número de procesadores, lo que es posible debido a que el sistema es controlado por un sistema operativo distribuido.

ABSTRACT

An MIMD architecture based on a dependence driven modell is presented. It consists on a network with no switching requirements but $O(n^2)$ memories, where n is the number of subsistems that conform the system. Its main caracteristic is a linear speed up of processing with the number of processors, which is possible through a distributed operating system, that controls the configuration.

* Ingeniero Civil Electricista (PUC 1980); M.Sc. Information and Computer Science (GA TECH, 1984). Profesor Escuela de Ingeniería, Departamento Ciencia de la Computación (143), P. Universidad Católica de Chile, Casilla 6177, Santiago, Chile.

** Ingeniero Civil Electricista (UCH 1973); Magister en Ingeniería Eléctrica (UCH 1974), Doktor Ingenieur (TU München, 1981); Profesor Escuela de Ingeniería, Departamento de Ingeniería Eléctrica (101), P. Universidad Católica de Chile, Casilla 6177, Santiago, Chile.

1 INTRODUCCION

La disponibilidad de microprocesadores de muy bajo costo y de muy alta capacidad de procesamiento hace posible hoy dia concebir supercomputadores basados en un gran número de estos dispositivos. Tales configuraciones, que conforman un sistema de multiprocesamiento del tipo MIMD (multiple instruction multiple data), hacen uso del paralelismo inherente a un programa, a objeto de conseguir que éste sea ejecutado en el menor tiempo posible.

Existen diferentes alternativas para implementar una máquina MIMD [1]. En este trabajo se presenta una arquitectura controlada por dependencias [2],[3],[4],[5] que incorpora nuevos elementos que le permiten un comportamiento lineal en su relación ganancia de velocidad versus número de procesadores.

En la sección 2 del trabajo se describe como presentar un programa como un grafo de tareas y la manera de procesarlo en forma paralela. En la sección 3 se describe el modelo de arquitectura propuesto y se finaliza con algunas conclusiones en la sección 4.

2 PROCESAMIENTO PARALELO DE UN PROGRAMA ESCRITO EN LENGUAJE DE ALTO NIVEL REPRESENTADO POR UN GRAFO DE DEPENDENCIAS

2.1 Descripción de un grafo

Un programa puede representarse como un grafo, cuyas unidades básicas se denominan nodos funcionales. Un nodo funcional es un segmento de programa que conforma una unidad que es difícil de paralelizar, como son relaciones recurrentes, condiciones indentadas, etc.

Para simplificar, supongamos que un nodo funcional es una subrutina de FORTRAN o un procedimiento o función de PASCAL. Un nodo tendrá así como entrada toda los datos que requiere para ser ejecutado, y su salida será el conjunto de datos que genere.

La figura 1 muestra un conjunto de procedimientos escritos en PASCAL, cuya representación como nodos funcionales se ilustra en la figura 2. El grafo de tareas de la figura 3, especifica el conjunto de todas las diferentes posibilidades de flujo entre los nodos funcionales del ejemplo de la figura 1.

Se observa que el flujo de ejecución puede ser alterado por toma de decisiones, como por ejemplo "repeat until" del nodo 0, el cual es una decisión que depende del booleano D. Cuando el resultado del loop es verdadero para D, la ejecución lleva a salir del loop; en caso contrario la ejecución permanece en el loop.

Se destaca también que el resultado de la ejecución de un nodo produce nuevos datos que pueden liberar más de un nodo, cuya ejecución puede así realizarse en paralelo. Por ejemplo, en el loop del nodo 0 es posible ejecutar dos ramas simultáneamente, las líneas (1) y (2) forman una de las ramas, mientras (3) y (4) forman la segunda. Además, la lectura de la variable F y la ejecución del nodo 2 con parámetro F pueden realizarse en paralelo con la ejecución del loop, dado que no existe interacción entre las variables.

2.2 Procesamiento paralelo

Un sistema de multiprocesamiento ideal es aquel en que la ganancia de velocidad crece linealmente con el número de procesadores conectados al sistema. Esto sólo es posible si la distribución de tareas ("schedulling") no se transforma en un cuello de botella al crecer el número de procesadores del sistema. Para esto es conveniente que el "scheduling" sea distribuido. Por otra parte, la ganancia de velocidad también se degrada por el costo computacional que corresponde a la comunicación requerida para el intercambio de datos.

En algunos casos es muy simple determinar la ganancia de velocidad que se obtiene en un sistema de multiproceso con "scheduling" distribuido, lo que permite comparar dos modelos de arquitectura. Supóngase que en un sistema de multiprocesamiento cada procesador ejecuta "scheduling", comunicación entre procesadores y ejecución de código, y que en un segundo sistema de multiprocesamiento existen dos procesadores independientes; uno para "scheduling" y comunicación y otro para ejecución de código. A fin de realizar la comparación entre ambas arquitecturas, empleando el programa ilustrado en figura 1, es necesario definir los siguientes términos:

Ts : Tiempo de ejecución serial total en un sistema de un sólo procesador.
Tpo : Tiempo de ejecución paralelo total, considerando que cada procesador ejecuta código, comunicación y "scheduling".
Tp : Tiempo de ejecución paralelo total, con procesadores independientes para ejecución de código y, "scheduling" y comunicación.
To : Tiempo de "scheduling" y comunicación para una tarea.
T1,T2 : Tiempo requerido para ejecutar los nodos 1 y 2 del ejemplo, respectivamente.
Tr : Tiempo necesario para leer datos de entrada.
n : Número de iteraciones en el loop de la figura.
S0 : Ganancia de velocidad que se obtiene con cada procesador realizando ejecución de código y, "scheduling" y comunicación.
Sp : Ganancia de velocidad que se obtiene con procesadores independientes para ejecución de código y, "scheduling" y comunicación.

Así se obtiene para el ejemplo anterior:

$$Ts = Tr + T1 + T2 + n*(Tr + 2*T1 + 2*T2)$$
$$Tpo = n*(3*To + Tr + T1 + T2)$$
$$Tp = n*(Tr + T1 + T2)$$

De aquí pueden determinarse las ganancias Spo y Sp:

$$Spo = Ts/Tpo = (Tr + T1 + T2 + n*(Tr + 2*T1 + 2*T2))/n*(3*To+Tr+T1+T2)$$

y

$$Sp = Ts/Tp = (Tr + T1 + T2 + n(Tr + 2* T1 + 2*T2))/(n*(Tr+T1+T2))$$

Puede observarse que para n grande que

Spo tiende a $(Tr + 2*T1 + 2*T2)/(3To + Tr + T1 + T2)$

y

Sp tiede a $(Tr + 2*T1 + 2*T2)/(Tr + T1 + T2)$

Finalmente, podríamos suponer para efectos de comparación que todos los tiempos T son iguales, obteniendose así :

$Spo = 5/6 < 1$
$Sp = 5/3 > 1$

Del análisis anterior se puede observar que al tener procesadores que ejecutan las tareas de ejecución de código, "scheduling" y comunicación puede obtenerse un comportamiento inferior con el sistema de multiproceso que con un sólo procesador serial, debido al "overhead" introducido. Por otro lado, al independizar la ejecución de código por un lado y "scheduling" y comunicación por otro, se observa que se aprovechan las características paralelas inherentes al algoritmo, obteniéndose en el ejemplo un mejoramiento de un 67%.

3 DESCRIPCIÓN DE LA ARQUITECTURA Y SU OPERACIÓN

3.1 Comunicación entre procesadores

Para solucionar el problema de canalizar la información entre procesadores se propone una red que ocupa $O(N^2)$ memorias, sin requerimientos de conmutación, donde N es el número de subsistemas, con cada subsistema constituido por dos procesadores, cuyas funciones se describen posteriormnete; dado que el costo de la memoria es bajo y en el futuro tenderá a bajar más aún, sistemas con Gigabytes de memoria comienzan a ser una realidad y los requerimientos anteriores no son excesivos.

A cada subsistema se asocian N buses de datos. De ellos, uno es bus de salida y los otros N-1 son buses de entrada correspondientes a los buses de salida de los N-1 subsistemas restantes. Además, cada subsistema tiene una memoria conectada a cada bus de entrada, la que cumple el papel de buffer. De esta manera cada subsistema tiene acceso directo a toda la información del sistema, eliminando los conflictos potenciales que pueden ocurrir al accesar los datos que requiere. Para obtener tal condición, cada subsistema envía a través de su propio bus las variables que ha procesado a todos los otros subsistemas, sin interrumpir la actividad de ninguno de ellos. Dado que todos los subsistemas tienen una unidad de memoria conectada a los N-1 buses de entrada (memoria de transferencia), los subsistemas obtienen la información que requieren directamente desde ésta sin alterar la ejecución normal de los otros subsistemas y sin necesidad de conmutación. Así se eliminan los conflictos de comunicación entre subsistemas, pues sólo un subsistema ocupa cada bus. Se observa que se ha obtenido un modelo de arquitectura en el cual la distribución de información se realiza mediante el proceso de direccionamiento, y no por una compleja red de conmutación. Esto es posible predefiniendo areas de memoria para las N memorias de transferencia. La única restricción es que los subsistemas requieren de un gran espacio de direccionamiento, lo que es en todo caso alcanzable con la tecnología actual de microprocesadores.

En el capítulo anterior se dedujo que el tiempo necesario para realizar el "scheduling" puede originar que un sistema de multiprocesamiento sea menos efectivo que un monoprocesador. En cambio, un modelo de arquitectura en que procesadores independientes realizan el trabajo del sistema operativo ("scheduling" y comunicación entre procesadores) y el trabajo de ejecución de código, permite obtener siempre tiempos de ejecución menores o iguales que con un sistema monoprocesador, pues los procesadores de código se encuentran activos el 100% del tiempo.

Para obtener la condición anterior, cada subsistema requiere de una memoria local accesable por los dos procesadores: el procesador de "scheduling" y comunicación (procesador S), y el procesador de código (procesador P). Se necesita además dividir la memoria local en dos bancos de memoria independientes; uno sobre el cual trabaje el procesador S, y otro sobre el cual trabaje el procesador P. Una vez que P finaliza su labor de procesar un cierto nodo, lo comunica a S, cambia de banco de memoria y comienza a ejecutar la tarea que S le había preparado con anterioridad. S vuelve a realizar su labor (a describirse mas abajo), pero empleando ahora el otro banco de memoria.

3.2 Distribución de tareas

El problema básico del "scheduling" o distribución de tareas, consiste esencialmente en la determinación de los nodos que están disponibles para ser ejcutados y la asignación de éstos a los procesadores que se encuentren sin trabajo. Esta labor es posible de realizar en forma distribuida, según el algoritmo cíclico que se describe a continuación.

Para poder realizar la asignación de tareas, se enumeran los procesadores y a los nodos disponibles se les asigna un criterio de prioridad, que puede basarse en el tiempo de espera, el número de nodos que libera una vez ejecutado o bién en otros criterios. La determinación de cuál de los nodos es conveniente de ser ejecutado por un subsistema dado no es sin embargo trivial, por el hecho que simultáneamente puede haber un grán número de procesadores S, buscando que nodo procesar próximamente. Este punto es sin duda uno de los mas críticos originados por la determinación distribuida de las tareas que se asignan.

Ya que gran parte de esta labor distribuida se duplicaría, es ventajoso contemplar un subsistema dedicado exclusivamente a esta labor común. Se le denominará subsistema de coordinación C y consistirá en cuatro procesadores C0, C1, C2 y C3.

El subsistema de coordinación C inicia una fase de "scheduling" con la lectura, desde su memoria de transferencia, de que procesadores están libres. El subsistema ordena y envía de vuelta a los procesadores S, mandando la lista sobre el bus dedicado para este fin, ocupando para esto el procesador dedicado C1 (ver figura 5).

También es necesario determinar qué nodos están disponibles para ser procesados, tarea que lleva a cabo el procesador dedicado C2. En figura 4, se observa que un nodo que fue liberado por un nodo predecesor no necesariamente es un nodo disponible por el hecho que un nodo puede tener muchos predecesores. Empleando la información sobre que nodos ya fueron procesados, obtenida por el procesador C2 de las memorias de transferencia de los procesadores, y el grafo del programa que se encuentra residente en la memoria local, se determinan los nodos que estan disponibles para su procesamiento. Estos nodos son ingresados a una lista que es ordenada por C3. La obtención de la lista ordenada de nodos se realiza así a base de una configuración "pipeline".

El procesador dedicado C0, realiza faenas de sincronización, tanto del pipeline (procesadores C2 y C3), como de los procesadores C3 y C1, sincronizando así la transferencia de amabas listas ordenadas. C0 tiene también la labor de ir eliminando de la lista de nodos disponibles de C2, aquellos nodos que ya fueron asignados en el ciclo respectivo.

Cabe destacar que la frecuencia con que este proceso de asignación se puede repetir esta dado por el máximo de tiempo requerido por los procesadores C1, C2 y C3, con lo que se obtiene una arquitectura síncrona de período variable (figura 6). Este intervalo de

tiempo dependerá del número de nodos y el número de subsistemas. El Número de nodos es un parametro dependiente del programa en ejecución, mientras que el número de subsistemas dependerá de la configuración. El tiempo máximo debido al número de subsistemas, dependerá del tiempo necesario en ordenar la lista de subsistemas que se encuentran buscando un nodo. Este tiempo es O(nlogn). Dado que n < 10000 se tiene que O(nlogn) = O(c*N) y O(c*N) = O(n) con lo que es posible obtener una ganacia lineal en la velocidad de procesamiento con el número de procesadores.

Así la labor de los procesadores S, a partir del momento en que algún S es informado que el procesador P asociado ha finalizado el procesamiento de la labor previamente asignada, queda reducido a:

1. Enviar la solicitud de requerimiento de nodo para procesar e indicar que el nodo correspondiente ha sido procesado.
2. Enviar los resultados obtenidos por el procesador P asociado.
3. Cargar en su memoria local las listas ordenadas de nodos y procesadores enviadas por C1 y C3 respectivamente.
4. Asociar ambas listas y elegir el nodo que le corresponde.
5. Cargar el código y los datos del nodo a procesar por su procesador P asociado en su memoria local compartida.

Cabe destacar que en el esquema anterior no es posible que ocurran "deadlocks", dado que ningún procesador P puede estar esperando por los datos de un nodo aún no finalizado de ejecutar, puesto que nodos que están esperando por datos no son considerados disponibles.

4 CONCLUSIONES

Se ha propuesto un modelo de arquitectura que aprovecha el paralelismo inherente a un programa, maximizando la concurrencia presente en él. Dado que el "scheduling" se lleva a cabo en forma distribuida, la configuración tiene la característica de poder crecer tanto como sea necesario. Se obtiene así un aumento lineal de la ganancia de velocidad respecto al número de procesadores presentes, restringido a la capacidad del algoritmo para ser procesado en forma paralela. Es importante destacar que este proceso distribuido no introduce "overhead", pues cada subsistema consta de un procesador dedicado a labores de sistema operativo y un segundo procesador para la ejecución de código.

Esta arquitectura fue concebida para alcanzar máxima velocidad, en tanto que la eficiencia no fue considerada del todo. Analizando este factor, se observa que el procesamiento de código representa a lo más el 50% de la capacidad total de procesamiento del sistema, dado que uno de los dos procesadores de cada subsistema se dedica exclusivamente al "scheduling" y comunicación. Sin embargo, considerando el estado actual de la tecnología, este punto no es de gran importancia, en la medida que lo fue en decadas pasadas cuando el sistema operavo tenía la mision de optimizar los costosos recursos del hardware.

Empleando procesadores convencionales de 16 bits (por ejemplo el Motorola 68000, el cual puede considerarse como una máquina de un MIPS), en una configuración de 100 subsistemas se podrían alcanzar velocidades de 100 MIPS y así capacidades de procesamiento similares a las de un supercomputador a un costo mucho menor. Además, hay que considerar que la velocidad indicada corresponde a procesamiento de código puro, puesto que el sistema operativo constituye un proceso concurrente y no implica en consecuencia "overhead".

Un inconveniente en la implementación de la arquitectura es la gran cantidad de buses requeridos. Cada procesador posee su propio bus, el que puede tener 40 o más líneas. Si se considera una configuración como la descrita en el párrafo anterior, se tendrían mas de 4000 líneas. Este punto puede solucionarse utilizando comunicación serial, con velocidades de transferencia de uno o dos órdenes de magnitud superior, lo que es alcanzable con la tecnología actual. Así se decrementa el número de líneas requeridas en la misma proporción.

La arquitectura propuesta constituye una arquitectura dinamicamente adaptable [5], [6]. Este tipo de arquitectura presenta las siguientes características:

1. Los recursos pueder reconfigurados a su mínima expresión.Esto significa que el sistema de mutiproceso con N procesadores, se puede reconfigurar por software para que hayan M (1 <= M <= N) grupos independientes trabajando. En este caso el número de programas concurrentes procesados simultáneamente es maximizado por el sistema, lo que permite la transformación de la máquina, de una máquina que maximiza "performance" a una gran máquina de "time-sharing".

2. Los recursos pueden ser conmutados entre diferentes tipos de procesadores (procesadores de I/O, máquinas pipeline, etc.), de tal manera que a cada procesador se asigne la tarea para la cual está mejor preparado. Así, en el ejemplo anterior se podría introducir un procesador de I/O que realice tareas de lectura.

REFERENCIAS

1. M.J. Flynn, "Some Computer Organizations and their Effectivness", IEEE Transactions on Computers, vol C-21, 1972, pp 948-960.

2. D.D. Gajski, D.J. Kuck and D.A. Padua, "Dependence Driven Computation", Proc. COMPCON, Spring 1981, pp 168-172.

3. R.G. Babb, "Parallel Processing with Large Grain Data Flow Techniques", Computer, Vol 17, Nº7, July 1984, pp 55-61.

4. J.L.A. Hughes, "Implementing Control-Flow Structures in Data Flow Programs", Proc. COMPCON, Spring 1982, pp 87-90.

5. S.R. Vegdahl, "A Survey of Prposed Architectures for the Execution of Functional Languages", IEEE Transactions on Computers, vol C-33, No.12, December 1984, pp 1050-1071.

6. C.R. Vick, S.P. Kartashev, and Steven I. Karatashev, "Adaptable Architectures for Supersystems", Computer, Vol 13 Nº11, November 1980, pp 17-35.

7. S.I.Kartashev and S.P. Karatashev, "Problems of Designing Supersystems with Dynamic Architectures", IEEE Transactions on Computers, Vol C-29, No. 12, December 1980, pp 1114-1132.

```
PROCEDURE Node_0 (var A: ARRAY [ 1..10] of integer;
                  var E:  boolean);
var B,C,F: Array [ 1..10] of integer;
      D: Boolean;
begin
   repeat read(A,B,C);
            Node_1(A,B,D);          (1)
            Node_2(A);              (2)
            Node_2(B);              (3)
            Node_1(B,B,E)           (4)
   until D;
   read(F);
   Node_2(F);
   Node_1(A,F,D);
end;

PROCEDURE Node_1(var A: ARRAY[ 1..10] OF integer;
                     B: ARRAY[ 1..10] OF integer;
                 var C: boolean);
var I: integer;
begin
    C:= FALSE;
    FOR I:= 2 TO 10
      do begin
           A[I]:= A[I-1] + B[I];
           IF A[I] < 100 then C:= true
         end;
end;

PROCEDURE Node_2(var A: Integer);
var I: integer;
begin
    FOR I:= 2 to 10
      DO A[I]:= (A[I] + A[I - 1]) / 2;
end;
```

FIGURA 2 Tres procedimientos escritos en Pascal, utilizados como ejemplos de nodos funcionales.

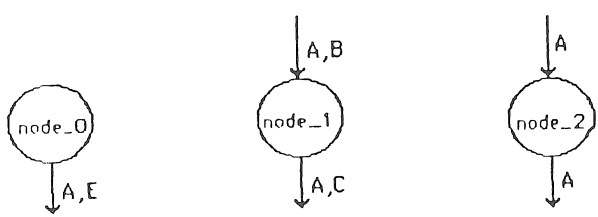

Figura 2 Nodos funcionales de los procedimientos de figura 1.

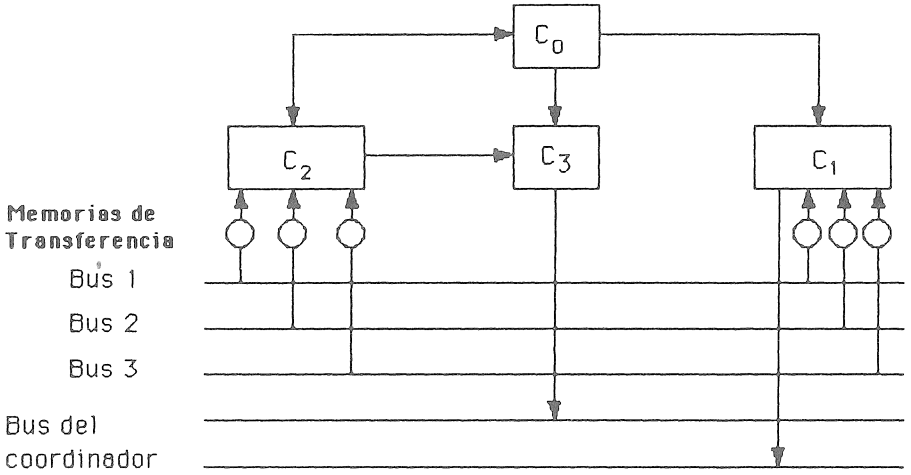

Fig. 5 Subsistema Coordinador.

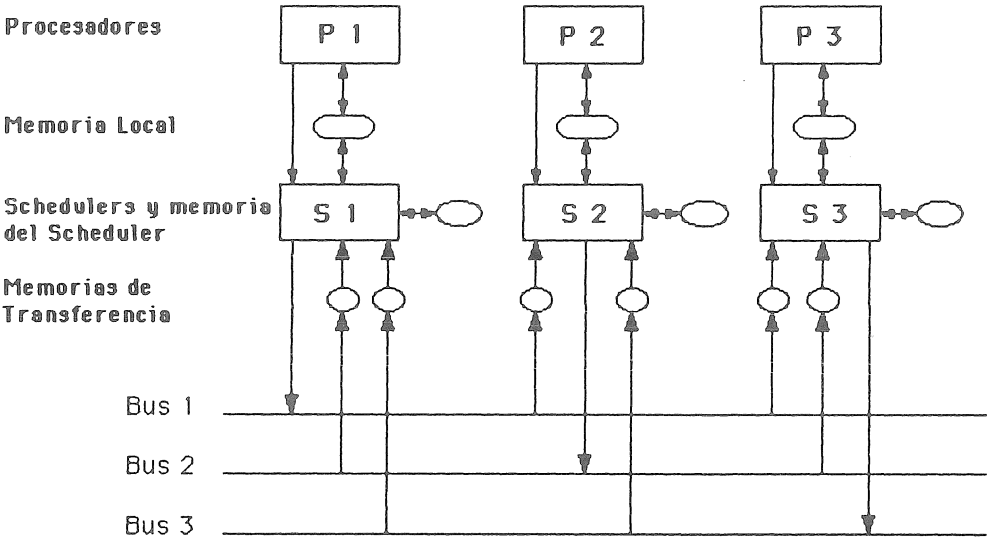

Fig. 6 Sistema de multiprocesamiento con tres subsistemas de dos procesadores cada uno.

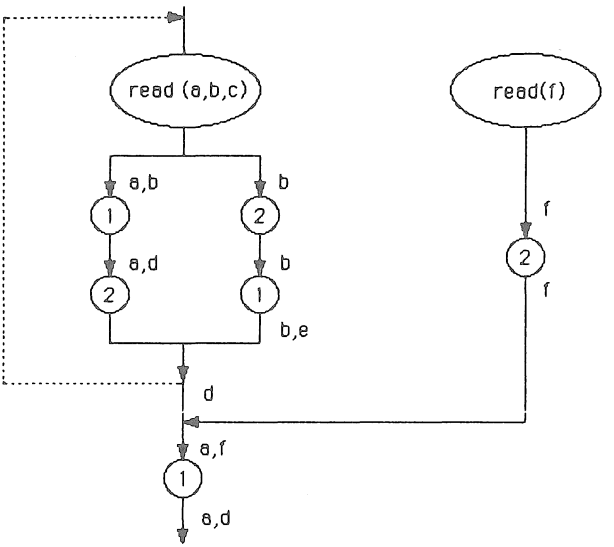

Fig. 3 Grafo de dependencias de tareas, para el ejemplo de la figura 1.

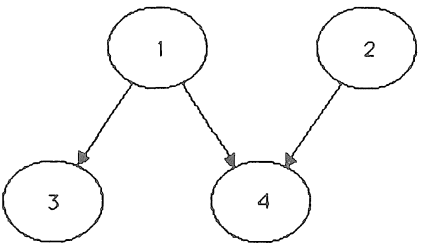

Fig. 4 Dependencia entre nodos.