coordinamento di Andrea de Prisco

Architettura e programmazione dei sistemi multiprocessore

parte terza

di Giuseppe Cardinale Ciccotti

Negli elaboratori paralleli di tipo MIMD, il Sistema Operativo ha un'importanza fondamentale. A differenza dei sistemi Pipeline o Array Processor che vengono visti come coprocessori e che quindi richiedono solo un piccolo insieme di routine per la loro gestione, nei sistemi multiprocessore è il kernel stesso che deve sfruttare efficientemente le capacità di parallelismo asincrono proprie dell'hardware. Dei requisiti di quest'ultimo abbiamo già parlato nelle precedenti sezioni dedicate ai sistemi multiprocessore, ci restano dunque da valutare le caratteristiche del software nel suo complesso; tratteremo perciò del sistema operativo e dei supporti programmativi

Classificazione dei sistemi operativi multiprocessore

Tra un sistema operativo, che nel seguito abbrevieremo per comodità con s.o., di un multiprocessore e quello di un sistema che utilizzi la multiprogrammazione vi è poca differenza da un punto di vista concettuale. C'è comunque un'addizionale complessità nel s.o. quando più processori devono lavorare simultaneamente. Questa complessità è dovuta al fatto che più task vengono eseguiti in maniera asincrona. Le capacità funzionali che sono spesso richieste in un sistema operativo multiprocessore includono l'allocazione delle risorse, la protezione della memoria, la prevenzione dei blocchi critici e la gestione di terminazioni inattese dei processi. In più il s.o. deve provvedere ad una corretta ed efficace utilizzazione delle risorse e perciò gestire opportunamente l'insieme dei dispositivi di input/output e soprattutto il carico di lavoro dei vari processori; quest'ultimo punto in particolare è una caratteristica unica e fondamentale dei s.o. multiprocessore e può richiedere la valutazione di diversi ed alternativi schemi di funzionamento. Una delle ragioni principali dell'uso dei sistemi multiprocessore è la possibilità di predisporre di un certo grado di tolleranza ai guasti, in modo che il guasto di un certo numero di processori non blocchi il funzionamento del sistema, ma ne abbassi in maniera progressiva l'efficienza. Il s.o. deve essere quindi in grado di riconoscere e gestire situazioni di fault hardware, riconfigurando il sistema in maniera opportuna. Queste specifiche comportano l'uso di un certo numero di tecniche per supportare la gestione del parallelismo anche mentre un programma viene eseguito. La presenza di più di un processore nel sistema comporta quindi un nuovo tipo di approccio al progetto del s.o. che effettivamente rimane ancora un problema di ricerca. Introdurremo qui soltanto le configurazioni base che sono state proposte nei sistemi multiprocessore prodotti.

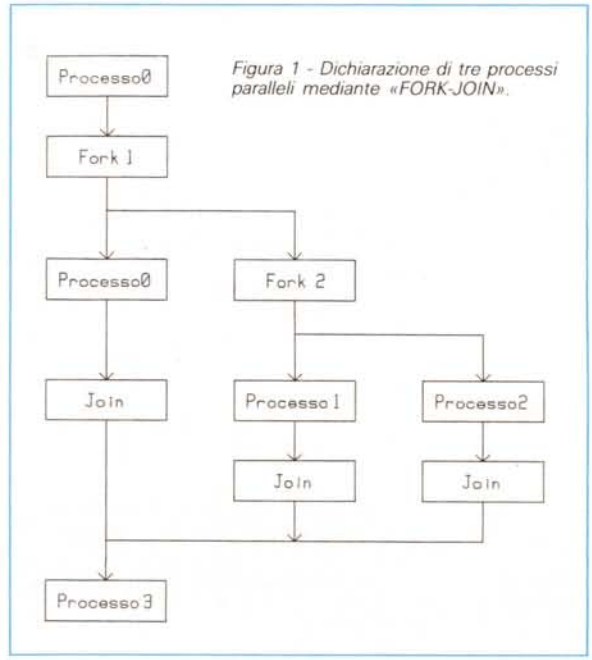

Figura 1 - Dichiarazione di tre processi paralleli mediante «FORK-JOIN».

Tre sono le organizzazioni fondamentali che sono state utilizzate nel progetto dei s.o. multiprocessore, e precisamente: la configurazione «master-slave», il «supervisore separato» per ciascun processore e il «controllo con supervisore flottante». Que-

sta piccola classificazione ricalca anche un certo ordine storico, in quanto la prima configurazione è stato il primo tipo di s.o. assunto per sistemi multiprocessore. Risulta d'altra parte, la meno efficiente delle tre anche se è decisamente la più semplice da implementare poiché può essere progettata come una estensione di un s.o. uniprocessore multitasking.

Nel modo master-slave, un solo processore, chiamato master, mantiene lo stato del sistema e distribuisce il lavoro agli altri processori. Un esempio di questo tipo si ha nel Cyber-170, dove le routine del sistema operativo sono eseguite soltanto da un processore periferico PO. Tutti gli altri processori sono trattati come slave di PO. Nel sistema DEC System 10 ci sono due processori identici; uno è designato master e l'altro slave, il kernel è eseguito solo sul master e lo slave è trattato come una risorsa disponibile. Questa configurazione comporta che tutte le richieste generate da interrupt, trap o chiamate dirette al supervisore, vengano trasmesse dal processore slave al processore master che, identificata la richiesta, la serve in maniera appropriata. Come si vede, da un punto di vista concettuale, non c'è nessuna differenza con un sistema uniprocessore. L'affidabilità di questo sistema non è migliore di quella di un sistema uniprocessore, in quanto un fault del master provoca il blocco del sistema che deve essere riavviato. Inoltre l'utilizzazione di processori slave può diventare estremamente bassa se il master non è in grado di distribuire il lavoro abbastanza in fretta da tenerli occupati. Il sistema master-slave è tuttavia abbastanza efficiente per quelle applicazioni dove il carico di lavoro è ben definito oppure per i sistemi asimmetrici nei quali gli slave sono meno potenti del processore master.

In un sistema con un supervisore separato per ciascun processore, il s.o. assume delle caratteristiche molto differenti dalla configurazione master-slave. Concettualmente l'approccio al problema è lo stesso adottato per le reti di computer, dove ogni processore mantiene una copia del kernel. La condivisione delle risorse avviene ad alto livello, per esempio tramite scambio di file. Anche se ogni processore serve le proprie richieste e un proprio insieme di I/O, vi è comunque un certo grado di interazione tra i processori ed è quindi necessario condividere alcune sezioni del codice del s.o. oppure replicarlo per ciascun processore. Scegliendo la prima ipotesi, è indispensabile che il codice sia, come si dice «rientrante», nel senso che può essere prelevato, ma non spostato e l'accesso ad esso impegnato soltanto per il tempo necessario alla lettura del codice stesso. La seconda soluzione ha il pregio di non creare competizione per l'accesso a questa risorsa, tuttavia comporta l'occupazione di una zona di memoria, anche molto ampia, che potrebbe risultare sottoutilizzata. Una soluzione intermedia può rivelarsi l'uso di memorie cache, di cui abbiamo trattato nella seconda parte, che mantengano i segmenti di codice di più frequente accesso. Tuttavia la determinazione sulla base delle frequenze di accesso, di quale porzione del sistema operativo si debba conservare è un compito difficile e fortemente dipendente dalla applicazione.

```
begin
 S0;
cobegin
 S1;
 S2;
 .
 .
 .
 Sn;
coend
 Sn+1;
end
```

Figura 2 - Dichiarazione di n processi paralleli tramite il costrutto «COBEGIN-COEND».

Lo schema a controllo con supervisore flottante considera tutti i processori come un insieme di risorse senza alcuna caratteristica distintiva fra di essi. La maggiore difficoltà di questo modo di operazione consente tuttavia la più alta flessibilità del sistema. Le routine del supervisore «viaggiano» da un processore all'altro secondo necessità e quindi molti processori possono eseguire routine di servizio in modo supervisore contemporaneamente. In questo modo il carico di lavoro può essere bilanciato istante per istante secondo necessità, in quanto ogni singolo processore è in grado di redistribuire il lavoro. Considerando poi che l'insieme dei dispositivi di I/O è comune a tutti i processori, si può comprendere come anche da questo punto di vista, tale configurazione raggiunga un'efficienza maggiore degli schemi precedenti. La condivisione della totalità delle risorse del sistema comporta un'attenta progettazione dei meccanismi di protezione e di integrità delle risorse, sia dal punto di vista funzionale che da quello dell'efficienza, in quanto la frequenza dei conflitti di accesso può essere molto elevata. Per questo motivo è necessario prevedere meccanismi di priorità che non penalizzino la flessibilità del sistema con livelli di accesso rigidi e quindi predisporre algoritmi per la determinazione statistica delle priorità. Esempi di s.o. basati su questo schema di funzionamento sono l'MVS e il VM dei sistemi IBM 3081. La maggior parte dei s.o. sono comunque basati su schemi misti, ma la tendenza attuale e quella di progettare s.o. sempre più distribuiti.

Il software per i sistemi multiprocessore

Abbiamo già visto nei precedenti articoli, su questa serie sul calcolo parallelo,

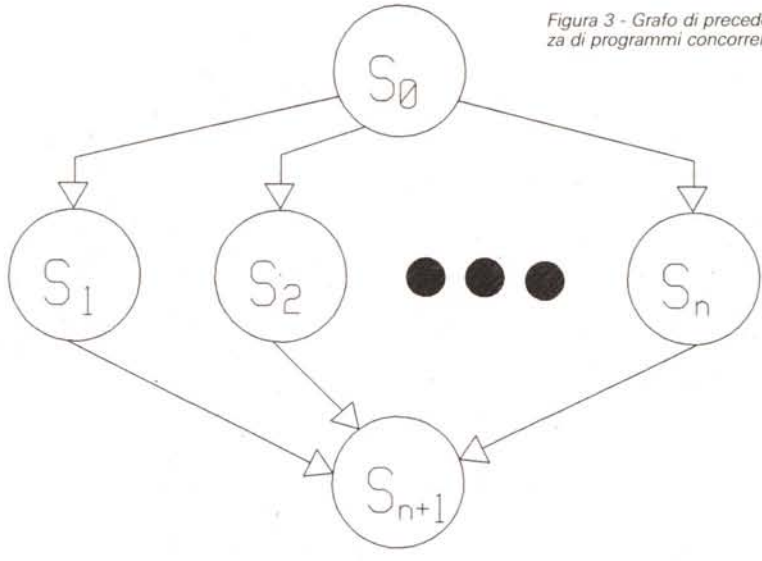

Figura 3 - Grafo di precedenza di programmi concorrenti.

```
var v: shared;
var w: shared;
cobegin
 csect v do P;
 csect w do Q;
coend
```

Figura 4 - Un semplice esempio di concorrenza tramite sezioni critiche. Più sezioni critiche possono essere eseguite in parallelo. Lo pseudocodice mette in evidenza che le variabili relative alle sezioni critiche debbono essere dichiarate «shared» cioè condivise.

come ogni struttura abbia necessità di un proprio stile di programmazione e di particolari strutture e istruzioni per supportare tutto il parallellismo di cui è capace l'hardware. Tuttavia abbiamo considerato come le strutture Pipeline ed Array Processor, richiedano un'attenta progettazione dell'algoritmo per evitare dannosi overhead dovuti alle comunicazioni. Nelle macchine tipo MIMD, invece, si vorrebbe raggiungere l'obiettivo di rendere l'algoritmo indipendente dalle risorse hardware a disposizione, per esempio dall'allocazione della memoria, premesso che ancora non esiste una metodologia per rendere l'algoritmo indipendente dalla struttura del sistema. Un'importante caratteristica hardware che influenza la programmazione in un sistema multiprocessore è la non omogeneità del sistema. È ovvio in linea di principio, che, se ci sono differenze funzionali nei processori, essi debbano essere trattati in maniera differente. Per esempio, se un processore possiede capacità di emulazione, che gli altri non hanno è chiaro che quei programmi che richiedono tale emulazione, dovranno essere eseguiti su quel processore. Abbiamo valutato come negli Array Processor un simile requisito dovesse essere previsto e soddisfatto dal programmatore applicativo; nei sistemi multiprocessore invece, è il s.o. che si occupa di gestire l'allocazione delle risorse correttamente ed efficientemente. Naturalmente questi requisiti comportano una maggior complessità del software di sistema rispetto al caso uniprocessore.

Un altro aspetto in cui la programmazione per i sistemi multiprocessore differisce da quella dei sistemi uniprocessore è nello stile di programmazione che deve essere coerente alla costruzione di una applicazione parallela. Bisogna considerare che un programma è diviso in processi indipendenti ognuno dei quali è allocato ad un processore ed ad un insieme di risorse hardware e software. Un processo può essere eseguito concorrentemente ad altri oppure ritardato quando ha necessità di interagire con altri processi. Quindi un programma parallelo sarà composto da almeno due processi interagenti. In un sistema multiprocessore, la sincronizzazione assume una grande importanza poiché penalizza il sistema in maniera non prevedibile. A differenza di un Array Processor dove le sincronizzazioni sono predisposte esplicitamente dal programmatore, in un sistema multiprocessore, queste sono gestite dal s.o. che è necessariamente basato su criteri più generali. Tuttavia la necessità di assicurare in ogni situazione la correttezza dei programmi, comporta dei meccanismi di sincronizzazione non molto efficienti per una certa ridondanza nella programmazione e nella frequenza di esecuzione. È questo il motivo perché vengono previste comunque delle strutture di controllo che il programmatore può adoperare nelle proprie applicazioni. Dato che queste strutture identificano segmenti di codice indipendenti, il programmatore è portato a progettare e costruire l'applicazione in modo modulare e più facilmente controllabile.

Il primo esempio di struttura di controllo è il «messaggio» che è di gran lunga la più usata, per il fatto che c'è una corrispondenza diretta con la struttura logica di sincronizzazione prevista nei sistemi uniprocessore multitasking. Come in questi sistemi i processi vengono eseguiti concorrentemente e si sincronizzano attraverso lo scambio di messaggi. Naturalmente i messaggi possono essere scambiati a vari livelli, nel senso che ci può essere un controllo a basso livello sincronizzando i processori sulla singola istruzione, in questo caso il messaggio sarà generato e gestito in hardware, oppure il controllo può essere a livello più alto sincronizzando i processi per esempio soltanto alla fine di ogni sezione indipendente.

Una seconda struttura di controllo è il «chore», che è una piccola unità di codice in cui è spezzato il programma. In questo approccio il processo coincide con il «chore». La sua caratteristica principale è che la sua esecuzione non può venire interrotta, fino al completamento. Quindi il «chore» deve necessariamente essere piccolo per evitare lunghe attese ed usare poche risorse in modo da non bloccare l'esecuzione di altri «chore». In generale l'uscita di un «chore» implica l'esecuzione di altri «chore» fino al completamento del programma. Consideriamo la sezione del s.o. che controlla la lettura da disco di un file di overlay: ci possono essere due «chore», uno che ordina la richiesta di trasferimento dal disco, e un altro che restituisce l'acknowledge alla fine del trasferimento e inizia l'azione seguente.

La terza struttura di controllo non sequenziale è quella basata sul «sistema di produzioni» spesso usata nei sistemi dedicati ad applicazioni di intelligenza artificiale. Le produzioni sono espressioni del tipo <antecedente, conseguente>. Se la valutazione dell'antecedente restituisce il valore booleano «vero» allora viene eseguito il conseguente. In un sistema dove ci sono in generale molteplici produzioni attive sono spesso richieste quattro fasi di scheduling (a) per la selezione degli antecedenti da valutare, (b) per ordinare, se necessario, l'esecuzione degli antecedenti, (c) per selezionare i conseguenti che devono essere eseguiti e (d) per ordinare, se necessario, l'esecuzione dei conseguenti.

```
cobegin
 P1:csect v do P;
 P2:csect w do Q;
coend
```

Figura 5 - Questo codice può produrre un «deadlock» in quanto nessuno dei processi P1 e P2 può rilasciare le risorse necessarie all'altro.

L'ultimo punto che distingue il s.o. di un sistema multiprocessore da quello di uno uniprocessore è la sezione relativa alla gestione degli errori. Mentre con un solo processore è sempre possibile assicurare un ambiente «corretto» disabilitando interrupt, e in casi limite bloccando l'attività dell'I/O, valutando perciò la correttezza di un programma solo sulla sequenza di istruzioni da cui è costituito, in un sistema a più processori ciò non previene situazioni di stallo e di blocco che quindi devono essere prevenute predisponendo sezioni hardware e software dedicate a questo scopo. Tuttavia per la mancanza di teorie che assicurino le condizioni di correttezza dei program-

BIBLIOGRAFIA

Hwang K., Briggs F., «***Computer Architecture and Parallel Processing***», McGraw-Hill 1988.

Tisato F., Zicari R., «***Sistemi Operativi, architettura e progetto***», CLUP 1985.

mi paralleli, il software che è deputato al controllo della correttezza aggiunge spesso ridondanza e inefficienza.

Requisiti dei linguaggi di programmazione paralleli

Vediamo ora quali sono i costrutti e le strutture che un linguaggio di programmazione deve avere per permettere l'esecuzione di processi concorrenti. Precisiamo subito cosa intendiamo per processi concorrenti: essi lo sono se le rispettive esecuzioni si sovrappongono. Due processi non sono concorrenti se l'esecuzione della prima istruzione del secondo parte dopo la terminazione dell'ultima istruzione del primo.

Una delle prime notazioni usate per esprimere la concorrenza è stata definita nel 1966 con l'introduzione del costrutto FORK, JOIN. Il FORK genera un nuovo processo specificando il nome del programma che deve essere mandato in esecuzione esattamente come se si trattasse di una subroutine, tuttavia a differenza di questo caso, il programma che esegue la FORK continua la sua esecuzione contemporaneamente al programma chiamato. Per ritornare ad unico programma da due o più programmi indipendenti si usa l'istruzione JOIN; in figura 1 vi è il diagramma che illustra il funzionamento del FORK e del JOIN.

L'uso del FORK è abbastanza intuitivo e, se usato dopo aver attentamente pianificato le possibilità di parallelismo, è un modo efficace di generare dinamicamente i processi; tra l'altro un s.o. popolare come Unix utilizza un metodo simile a questo per allocare i suoi processi. Un'estensione del costrutto FORK-JOIN ai linguaggi strutturati è stato proposto da Dijkstra; in questo caso un insieme di processi possono essere eseguiti in parallelo dichiarandoli con il costrutto COBEGIN-COEND come in figura 2. COBEGIN consente di dichiarare esplicitamente le parti di programma che possono essere eseguite in parallelo permettendo inoltre di distinguere le variabili condivise da quelle locali. In figura 3 potete vedere il grafo che illustra la precedenza tra i processi del listato in figura 2; tenete presente che il processo Sn+1 è eseguito soltanto dopo che tutti i processi S1... Sn sono terminati. Il fatto che i processi abbiano un solo ingresso e una sola uscita permette inoltre di seguire i canoni di una programmazione strutturata facilitando i compiti del compilatore in fase di controllo del programma.

I processi definiti con COBEGIN sono completamente disgiunti e ciò significa che il processo S2 non può accedere ad una variabile del processo S1. Tuttavia questa restrizione benché assicuri la correttezza del costrutto COBEGIN-COEND, non è molto efficiente, infatti due processi che hanno in comune una sola variabile devono essere eseguiti serialmente mentre basterebbe sincronizzare l'accesso a tale variabile. Potremmo dire che il COBEGIN nasconde al programmatore il problema dello scambio dei dati fra i processi, tuttavia deve necessariamente risolverlo a qualche altro livello.

```
var e: shared event;
cobegin
. .
/* processo mittente */
  begin
    .
   signal(e);
    .
  end;
  .
/* processo ricevente */
  begin
    .
   wait(e);
    .
  end;
  .
coend
```

Figura 6 - Esempio di sincronizzazione fra processi. Il processo mittente e quello ricevente sono paralleli perché dichiarati nel corpo dello stesso «COBEGIN».

Sezioni critiche e mutua esclusione

Il problema della condivisione delle risorse può essere risolto almeno dal punto di vista della correttezza, fornendo dei costrutti che esplicitino i segmenti di codice che utilizzano variabili condivise. Queste parti di programma sono dette «sezioni critiche» e devono essere trattate in maniera differente dal resto del programma.

L'accesso a tali sezioni è infatti soggetto alla «mutua esclusione»: soltanto un processo alla volta può eseguire quel segmento di codice. Naturalmente saranno necessari dei meccanismi tali da consentire agli altri processi di verificare se la sezione critica è già allocata, bisogna però assicurare che questa fase di test duri un tempo finito come pure l'esecuzione della sezione critica stessa. Questi due requisiti aggiuntivi fanno sì che la correttezza venga assicurata. Tuttavia queste condizioni sono difficili da testare per un compilatore a meno di restrizioni programmative, così è cura del programmatore assicurare che le sezioni critiche rispondano a tali requisiti.

I costrutti MUTEXBEGIN-MUTEXEND oppure CSET sono i più usati per identificare le sezioni critiche. Entrambi queste notazioni permettono la dichiarazione di variabili che possono essere locali o condivise; in questo secondo caso, il programmatore comunica al compilatore che quella variabile è sì condivisa, ma solo un processo alla volta può modificarla.

In figura 4 vedete un frammento di un programma che illustra questa situazione. Comunque anche la dichiarazione esplicita della sezione critica non assicura l'immunità al blocco critico, infatti una situazione come quella in figura 5 potrebbe causarlo. Il P2 accede alla sua sezione critica prima che P1 possa farlo o viceversa nessuno dei due processi potrà terminare, in quanto nessuno può rilasciare le risorse che servono all'altro. Senza entrare in dettagli che ci porterebbero lontano (sulla prevenzione e recupero dei blocchi critici esiste una vasta letteratura), consideriamo un modo diverso di gestire le allocazioni delle risorse, vale a dire la «sincronizzazione». Se fino ad ora ci eravamo posti come fine la totale indipendenza dei processi, ammettiamo ora che la correttezza possa essere assicurata sincronizzando esplicitamente i processi. Una maniera semplice di farlo è quella di utilizzare degli interrupt per segnalare la richiesta e la terminazione di processi asincroni tra processori.

Un modo più generale è quello di programmare degli «eventi». Quando un processo decide di attendere un evento ferma la sua esecuzione finché un altro processo «segnala» il verificarsi di tale evento. In figura 6 trovate un breve frammento di codice che mostra l'utilizzo dell'evento. Da un punto di vista formale l'evento non è niente altro che una variabile di tipo «evento», naturalmente dovrà essere condivisa.

Conclusioni

In questa terza parte sulle macchine MIMD, abbiamo iniziato a valutare «vizi e virtù» del software a corredo dei sistemi multiprocessore. Nella quarta ed ultima parte su tale argomento, vedremo qualche regola fondamentale per estrarre il parallelismo dai nostri problemi, come organizzare un programma parallelo e le principali strategie di scheduling. MC