Architettura e programmazione dei sistemi multiprocessore

di Giuseppe Cardinale Ciccotti

quarta parte

Abbiamo già avuto modo di verificare come la possibilità di estrarre parti di codice da eseguire in parallelo sia possibile a più livelli. Per schematizzare consideriamone quattro: intraistruzione, interistruzione, di procedura e di programma. Il primo è un livello per così dire «fisico» nel senso che il parallelismo può essere ottenuto i molti modi, addirittura senza parellelismo ottenuto strutturando il microprogramma della macchina, se ovviamente è microprogrammata, oppure, al limite, progettando circuiti ad hoc, in modo che le operazioni relative ad ogni istruzione elementare vengano eseguite in parallelo. Il quarto, all'opposto, prevede l'esecuzione di più programmi in concorrenza ed è un livello molto «astratto» in quanto il parallelismo può essere ottenuto in molti modi, addirittura senza parallelismo reale, con la tecnica del time-sharing. I due livelli intermedi rappresentano invece il ponte fra il parallelismo hardware e quello software; infatti l'esecuzione di istruzioni o frammenti di codice in parallelo, influisce sul progetto dell'architettura del sistema sia sui supporti programmativi

Parallelismo tra processi

Per un approccio concettuale più generale e completo in un ambiente così vario come quello delle macchine MIMD, consideriamo una nuova struttura logica che chiameremo «processo».

Un processo non è altro che un insieme di istruzioni eseguite in modo sequenziale. Si comprende perciò come il parallelismo relativo al secondo e terzo livello possa essere ricondotto al parallelismo tra processi; nel caso in cui più processi paralleli siano costituiti da una sola istruzione avremo infatti, un parallelismo interistruzione. In realtà il concetto di processo è così generale che un intero programma potrebbe essere visto come un processo e se possibile costituito a sua volta di processi. Se è possibile allora separare un programma in processi indipendenti, appare chiaro che questi potranno essere eseguiti in parallelo. Un programma parallelo perciò potrà essere strutturato in una gerarchia di processi a loro volta suddivisi in processi finché non sia più possibile parallelizzare il frammento di codice o non il processo stesso sia costituito da una sola istruzione. Da un punto di vista pratico i processi potranno essere sincroni o asincroni e quindi sarà necessario disporre di costrutti e istruzioni come quelli che abbiamo visto nei precedenti articoli di questa serie ed in particolare nel numero 98 di MC. La partizione di un algoritmo in processi indipendenti può essere fatto in due maniere: dal programmatore stesso che nell'analisi del programma individua i compiti parallelizzabili e struttura il codice di conseguenza o da un programma appositamente studiato che analizza il codice di un programma seriale e ne produce del codice equivalente parallelo.

Naturalmente il primo metodo è molto più efficiente per la stessa ragione per cui un compilatore ad alto livello produce un codice macchina sicuramente più lungo e ridondante di quello scritto da un abile programmatore Assembler (Turbo Pascal 5.0 permettendo!). Di fondo qualunque programma per quanto sofisticato non potrà mai andare al di là del codice in ingresso, nel senso che se il programmatore ha, ad esempio, strutturato i dati in maniera errata, il programma non riuscirà a produrre un codice efficiente. Questo significa, lo ripetiamo, che è necessaria una «forma mentis» diversa quando si passa a strutturare algoritmi che debbano essere eseguiti in parallelo.

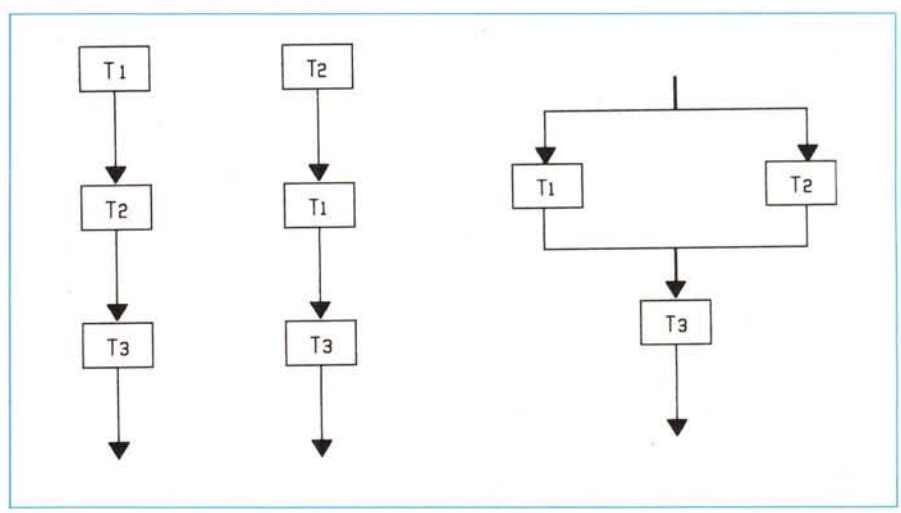

Figura 1 - Esecuzione seriale e parallela di processi.

Criteri di estrazione del parallelismo

Vediamo ora quali sono i criteri su cui basare l'analisi per l'estrazione del parallelismo interprocesso. La «dipendenza dei dati» è sicuramente il fattore principale che determina l'indipendenza delle istruzioni; un'istruzione o se volete un frammento di codice o ancora meglio un processo, è indipendente da un'altra se i suoi dati di input non dipendono dai dati di output dell'altra. Una condizione necessaria è la «commutatività», in figura 1a abbiamo tre diverse istruzioni sequenziali, se T3 è indipendente dall'ordine in cui T1 e T2 sono eseguite, possiamo dire che esiste del parallelismo tra T1 e T2. Tuttavia questa condizione non è sufficiente in quanto non è detto che due processi che possano essere eseguiti in qualsiasi ordine fra loro siano in realtà paralleli. Consideriamo quest'esempio, un filtraggio di un'immagine raster che calcola il valor medio di ciascun pixel rispetto agli adiacenti e poi ne inverte tale valore, possiamo eseguirlo perlomeno in due modi:

1. Primo modo

Per ciascun pixel
a. Valore del pixel = Media dei 9 pixel adiacenti
b. Valore del pixel = Valore max. - valore del pixel

2. Secondo modo

Per ciascun pixel
a. Valore del pixel = Valore max. - valore del pixel
b. Valore del pixel = Media dei 9 pixel adiacenti

Il risultato sarà il medesimo ma è chiaro che questi due processi non potranno essere eseguiti in parallelo. Esistono quindi delle condizioni più stringenti che devono essere soddisfatte affinché processi sequenziali possano essere eseguiti in parallelo. Sono le importanti «condizioni di Bernstein». Esse si basano su due insiemi distinti di variabili per ciascun processo Ti:

1. L'insieme di input Ii che racchiude tutte le locazioni di memoria per le quali la prima operazione di Ti che le interessa è una lettura.
2. L'insieme di output Oi che comprende tutte le locazioni di memoria che vengono scritte durante Ti. Le condizioni sotto le quali due processi T1 e T2 possono essere eseguiti come due processi indipendenti e concorrenti sono:

1. Le locazioni da cui T1 legge non devono essere modificate da operazioni di scrittura di T2. Ciò significa che gli insiemi I1 e O2 devono essere mutuamente esclusivi, cioè

$I1 \cap O2 = 0$

2. Per simmetria deve valere contemporanemante

$I2 \cap O1 = 0$

Inoltre è necessario che lo stato del sistema (il contenuto di tutte le locazioni di memoria) all'esecuzione di T3 sia indipendente dall'ordine di esecuzione di T1 e T2, quindi I3 deve essere in muta esclusione con le locazioni coinvolte da operazioni di scrittura in entrambi i processi T1 e T2, cioè:
$(O1 \cap O2) \cap I3 = 0$

Se si considera Ti come un'istruzione di un linguaggio ad alto livello, allora Ii e Oi rappresentano, rispettivamente i dati di input (le variabili che compaiono alla destra delle istruzioni di assegnazione) e quelli di output di Ti. In figura 2 mostriamo un frammento di codice in pseudolinguaggio per il calcolo di un'espressione aritmetica e i rispettivi insiemi I e O. Dato che $I1 \cap O2 = 0$, $I2 \cap O1 = 0$, e $O1 \cap O2 = 0$, i processi T1 e T2 possono essere eseguiti in parallelo. Il processo T3 non può invece, essere eseguito in parallelo né con T1 né con T2, in quanto $I3 \cap O2 = 0$ e $I3 \cap O1 = 0$. Quindi possiamo scrivere un programma equivalente per eseguire concorrentemente i processi T1, T2, T3 come in figura 3. Le

Figura 2 - Calcolo di un'espressione aritmetica, e i rispettivi insiemi di input e di output necessari alla verifica delle condizioni di Bernstein.

```
T1; X=(A+B)*(A-B)
T2; Y=(C-D)*1/(C+D)
T3; Z=X+C

I1={A, B}   I2={C, D}   O1={X}   O2={Y}
```

```
begin
 cobegin
  X=(A+B)*(A-B)
  Y=(C-D)*1/(C-D)
 coend
 Z=X+Y
end
```

Figura 3 - Verificate le condizioni di Bernstein, possiamo scrivere un programma parallelo che calcola l'espressione aritmetica.

```
for i=1 to n do
 begin
  A(i)=B(i)
  C(i)=D(i)
 end
```

Figura 4 - Questo loop di n iterazioni può essere parallelizzato perché tutte le iterazioni rispettano a due a due le condizioni di mutua esclusione.

condizioni che abbiamo posto possono essere usate come base di un metodo per la estrazione automatica del parallelismo da programmi sorgenti scritti in linguaggi ad alto livello. Tuttavia l'efficienza che si riesce ad ottenere con questo metodo è in genere bassa. Una delle principali lacune di tale approccio è la parallelizzazione dei loop; come potete immaginare con tale sistema si riesce a parallelizzare l'esecuzione all'interno di ogni singola iterazione del loop ma tutte le iterazioni sono poi eseguite sequenzialmente. Tuttavia è possibile adottare un criterio abbastanza semplice per scoprire se tutte le iterazioni di un loop possono essere eseguite in parallelo oppure debbono essere computate serialmente. Il «test delle replicazioni totali» può essere affrontato a vari livelli di sofisticazione, uno dei più semplici è il seguente: consideriamo le istruzioni S1,..., Sn eseguite in ciascuna delle n iterazioni del loop, avremo perciò gli insiemi di input e di output Ij e Oj relativi ad ogni iterazione j. Se per tutti gli i diversi da j $Ii \cap Oj = 0$, allora tutte le iterazioni del loop possono essere replicate cioè eseguite in parallelo, in figura 4 trovate un codice che verifica tale condizione. Tuttavia esistono dei casi che pur non verificando questo vincolo possono essere parallelizzati con qualche semplice accorgimento; si può notare che nel loop in figura 5 l'istruzione T = g(A(i)) viola le condizioni di mutua esclusione.

Basta però che l'istruzione T = g(A(i)) venga riscritta come T(i)=g(A(i)) e che T(i) compaia anche nella successiva istruzione, affinché le singole iterazioni del loop siano parallelizzabili.

```
for i=1 to n do
 begin
  A(i)=f(A(i))
  T=g(A(i))
  B(i)=h(T)
 end
```

Figura 5 - Questo codice viola le condizioni di Bernstein per i loop, pertanto l'esecuzione delle iterazioni non può essere parallelizzata; tuttavia basta riscrivere opportunamente il codice affinché lo diventi.

Strutturazione degli algoritmi

La capacità di separare un programma in processi indipendenti dipende perciò da una serie di criteri abbastanza definiti, quindi è possibile creare una metodologia, anche se non sistematica, al fine di affrontare tale problema. Tuttavia un parallelismo maggiore può essere ottenuto sfruttando l'altra caratteristica propria dei sistemi multiprocessore, le comunicazioni. Algoritmi che non verificano le condizioni di Bernstein, possono comunque essere parallelizzati sincronizzandoli secondo uno dei meccanismi che abbiamo visto negli appuntamenti precedenti. Resta tuttavia da valutare l'efficienza di tale approccio, sappiamo infatti che le comunicazioni introducono un overhead che, al limite, potrebbe vanificare i vantaggi ottenuti con l'esecuzione parallela. L'efficienza dipende inoltre dalla architettura stessa dell'hardware, allocando, per esempio, i dati in modo opportuno, è possibile risparmiare preziosi overhead di comunicazione.

Questo implica la conoscenza delle caratteristiche principali, non sempre note, della macchina per cui si struttura l'algoritmo. È naturale quindi, come d'altronde nella progettazione degli algoritmi seriali, affrontare la programmazione con un livello di astrazione maggiore; dal punto di vista della gestione delle comunicazioni ciò significa considerare dei meccanismi di comunicazione asincroni. Come è prevedibile, il vantaggio che otteniamo considerando soltanto le caratteristiche logiche e non quelle temporali comporta d'altra parte lo svantaggio di introdurre delle potenziali fonti di inefficienza. Rendendo asincrone le comunicazioni, diventano imprevedibili alcuni fattori temporali legati per esempio ai conflitti per l'accesso alla memoria.

L'effettivo utilizzo di molti sistemi multiprocessore è limitato dalla mancanza di metodi pratici per il progetto di algoritmi efficienti per tali computer.

Esistono tuttavia alcune linee di guida che aiutano la stesura di algoritmi efficienti. Una delle tecniche principali, che racchiude tutto quello che abbiamo considerato finora, consiste nel «Partizionamento» dell'algoritmo in processi e dell'«Assegnazione» di questi ai processori.

Quest'ultimo punto è la naturale estensione del problema dello scheduling di ambienti multitasking, applicato ad ambienti multiprocessore e pertanto può essere affrontato almeno ad un livello di astrazione con le stesse metodologie. La differenza principale risiede

```
var pixel[0..m-1,0..n-1]: byte;
var histog[0..255]: integer;
init histog[0..255]=0; /* inizializzazione dei contatori
                          delle frequenze */
for i=0 to m-1 do
 for j=0 to n-1 do
  histog[pixel[i,j]]=histog[pixel[i,j]]+1;
```

Figura 6 - Algoritmo seriale per il calcolo dell'istogramma di un'immagine raster a 255 livelli di grigio.

Figura 7 - Versione parallela dell'algoritmo per l'istogramma di una immagine. La parola chiave parfor attiva p processi paralleli che devono accedere alla variabile histog [] in mutua esclusione affinché sia mantenuto il vincolo di correttezza. È necessario perciò dichiarare una sezione critica.

```
var histog[0..255]: integer: shared; /* variabile condivisa */
init histog[0..255]=0; /* inizializzazione contatori delle frequenze */
 s=m/p;
 parfor i=1 to p do /* attivo p processi paralleli */
  begin
   var pixel[(i-1)*s..  s*i-1,0..n-1]: byte;  /* p  array di  s righe di
                                                 pixel */
   for k=(i-1)*s to s*i-1 do
    for j=0 to n-1 do
     csect histog[pixel[k,j]] do
           histog[pixel[k,j]]=histog[pixel[k,j]]+1;
```

```
var histog[0..255]: integer;
init histog[0..255]=0; /* inizializzazione contatori delle frequenze */
 s=m/p;
 parfor i=1 to p do /* attivo p processi paralleli */
  begin
   var pixel[(i-1)*s..s*i-1,0..n-1]: byte; /* p array di s righe di pixel */
   var lhistog[i,0..255]: integer; /* p array per l'istogramma locale */
    for k=(i-1)*s to s*i-1 do
     for j=0 to n-1 do
      lhistog[i,pixel[k,j]]=histog[i,pixel[k,j]]+1;
   end
  for j=0 to 255 do /* somma degli istogrammi locali */
   for i=1 to p do
    histo[j]=histog[j]+lhistog[i,j];
```

Figura 8 - Parallelizzazione dell'algoritmo per il calcolo dell'istogramma, in modo tale da evitare gli overhead dovuti ai conflitti di accesso alla memoria. Notate comunque, l'overhead finale per la somma totale degli istogrammi locali e la necessità di un maggior numero di locazioni di memoria.

nella considerazione che, rispetto al caso uniprocessore, si pone la scelta di adoperare strategie di allocazione statiche o dinamiche dei processi e delle risorse ad essi necessarie. È chiaro che nel primo caso l'efficienza sarà maggiore non essendo previsti overhead dovuti ad un sistema operativo più evoluto e potente necessario per la gestione dinamica dei processi, naturalmente quest'ultima sarà assai più flessibile e semplice da usare dal lato dei programmi utente. Per illustrare invece i problemi legati al «Partizionamento» vediamo ora un esempio pratico di programmazione.

Problemi di partizionamento

Consideriamo un algoritmo che esegua un istogramma di una immagine raster che è rappresentata da un array rettangolare di pixel. Ognuno di questi è un numero intero a 8 bit che rappresenta, per esempio, un livello di grigio in una scala tra il nero, 0, e il bianco, 255.

Fare un istogramma non significa altro che tenere traccia della frequenza di occorrenza di ciascun livello di grigio nella immagine. L'idea alla base dell'algoritmo è di predisporre un array che chiameremo histog [0...255], che contenga in ogni locazione il numero di pixel di ciascun grigio contati nella immagine.

La nostra immagine è rappresentata da un array di m righe ed n colonne che chiameremo pixel. Sia pixel (i,j) quindi il livello di grigio relativo alla locazione (i,j) dell'immagine; il programma seriale per computare l'istogramma è dato in figura 6. Si vede facilmente che la sua complessità è O(mn). Notiamo subito che tutto l'algoritmo è dominato dall'esecuzione dei loop che scandiscono righe e colonne. Disponendo di un sistema con p processori si può pensare di partizionare l'immagine in m/p=s righe e di attivare p processi ognuno con il compito di analizzare s righe contigue dell'immagine. Naturalmente bisogna soddisfare il vincolo di condivisione dell'array histog [0-255], è necessario infatti che ogni processo acceda a tale variabile. In figura 7 potete trovare la versione parallela di quest'algoritmo. Tuttavia il numero di conflitti è proporzionale al numero di processi e quindi non è detto che aumentando il numero di processi si ottengano risultati migliori, anzi ci sarà sicuramente un valore di p tale per cui l'efficienza decresce. Se non ci fossero tali conflitti il tempo di esecuzione dell'algoritmo sarebbe proporzionale a O(sn) con uno speed-up potenziale pari a p. Consideriamo quindi il caso in cui il codice relativo al processo e il segmento dell'immagine da analizzare siano contenuti nella memoria di ciascun processore e supponiamo che ciascun processo abbia un proprio array histog [0-255] locale. In figura 8 trovate il frammento di codice che illustra tale soluzione; come potete notare è presente un ulteriore ciclo alla fine per sommare tutti gli istogrammi parziali. Questo naturalmente costituisce un overhead rispetto alle altre soluzioni, tuttavia in generale questo tempo può essere calcolato e ottimizzato con molta accuratezza. Un altro problema potrebbe risiedere nell'occupazione totale degli array locali, tuttavia le dimensioni della memoria occupata da questi è irrisoria rispetto a quella della immagine.

Conclusioni

Anche per questo tipo di architetture, i sistemi multiprocessore, dobbiamo purtroppo concludere che allo stato attuale delle ricerche non esiste una metodologia completa ed efficiente per strutturare algoritmi paralleli. Questi risultano sempre molto dipendenti dall'architettura per cui sono stati pensati e una loro generalizzazione porta sempre delle inefficienze. Addirittura può accadere che un programma efficiente per un certo tipo di architettura, risulti molto inefficiente per un altro. Vogliamo qui dare un ultimo criterio per valutare l'efficienza di un algoritmo: questa può essere misurata rispetto alla percentuale di utilizzazione dei processori o allo speed-up ottenuto, un concetto fondamentale risulta quindi quello del compromesso «computazione-comunicazione». Tale concetto è simile al compromesso «spazio-tempo» ormai d'uso nella programmazione seriale. Abbiamo visto come partizionando sia possibile ridurre la complessità delle comunicazioni, aumentando la complessità delle computazioni.

Questa considerazione ci porta ad un interrogativo assai stimolante: è preferibile un processore veloce ma poco dotato per le comunicazioni oppure un processore predisposto per questo tipo di attività tuttavia meno potente dal punto di vista della velocità di elaborazione e dell'estensione del set di istruzioni? Fino ad ora i principali produttori di microprocessori ci hanno «regalato» processori «tradizionali» molto potenti ma assai poco inclini al colloquio, tuttavia incominciano ad apparire in commercio nuovi processori assai più versati alle comunicazioni veloci e che, tra l'altro, offrono anch'essi performance nell'ordine delle decine di MIPS.

Nei prossimi appuntamenti parleremo anche di queste «meraviglie» della tecnologia e degli orizzonti applicativi che ci schiudono.

Bibliografia
H'ang K., Briggs F., «Computer architecture and parallel processing», Mc Graw-Hill, 1988.