di Andrea de Prisco

OCCAM: canali e messaggi

Dopo la doverosa introduzione del numero scorso, questo mese scenderemo un po' più nei dettagli riguardo le comunicazioni tra processi OCCAM. Vedremo poi come risolvere il problema della sincronia delle comunicazioni, mostrando un primo esempio di programma OCCAM. Naturalmente parallelo...

Array e matrici

OCCAM mette a disposizione, purtroppo, ben pochi tipi di dato. Oltre ai già citati INT e BYTE esiste il tipo di dato BOOL (che può assume solo uno dei due valori TRUE o FALSE) e, in alcune implementazioni, anche il tipo REAL32 e REAL64 che differiscono per il numero di bit delle rispettive rappresentazioni.

Tutti i tipi di dato (compresi i canali, come vedremo) possono essere «vettorizzati» ossia utilizzati come vettori o matrici. Ciò è possibile indicando nella dichiarazione il numero di elementi di cui è composta la nostra struttura accanto al tipo degli elementi stessi. Ad esempio la dichiarazione:

```
[100]INT pippo:
```

definisce un array di interi di nome «pippo» i cui elementi sono indirizzati con pippo[0], pippo[1], ..., pippo[99]. Analogamente possiamo definire array multidimensionali semplicemente indicando più dimensioni nella dichiarazione:

```
[50][100]BOOL pluto:
```

definisce una matrice 50×100 i cui elementi sono di tipo BOOL.

Sono possibili assegnamenti tra array senza ricorrere a loop, a condizione che gli array sorgente e destinazione siano dello stesso tipo (dimensione dell'array e tipo degli elementi). Se le dimensioni non coincidono (ma solo il tipo degli elementi) è possibile effettuare assegnamenti parziali. Parti di array più grandi in array più piccoli o, viceversa, array più piccoli in parti di array più grandi. Scrivendo ad esempio:

```
[pippo FROM 10 FOR 50]
```

indichiamo di fatto i 50 elementi dell'array pippo dalla posizione 10 fino alla posizione 59. Questa indicazione è valida tanto per la sorgente che la destinazione di un assegnamento o, più in generale, in qualsiasi valutazione di espressione. Facciamo un esempio. Immaginiamo di aver dichiarato anche l'array «piolo» nel seguente modo:

```
[50]INT piolo:
```

l'assegnamento:

```
piolo:= [pippo FROM 10 FOR 50]
```

è certamente lecito. Analogamente possiamo riferirci a sottoinsiemi sia come sorgente che come destinazione:

```
[piolo FROM 5 FOR 10]:=[pippo FROM 50 FOR 10]
```

in questo caso i dieci elementi di pippo dalla posizione 50 alla posizione 59 sono copiati negli elementi di piolo dalla posizione 5 alla 14. L'importante è che il sottoinsieme destinazione e il sottoinsieme sorgente abbiano pari dimensioni (ed elementi dello stesso tipo).

Per finire (i progettisti di OCCAM si sono proprio sbizzarriti con gli array) è possibile definire una abbreviazione per indicare un determinato sottoarray di un array di partenza. Ad esempio scrivendo:

```
pippino IS [pippo FROM 10 FOR 50]
```

possiamo alternativamente accedere ai 50 elementi di pippo attraverso il nome pippino. Così pippino[0] sarà pippo[10] e così via fino a pippino[49] che sarà pippo[59]. In tal modo pippino è a tutti gli effetti un array di 50 elementi di tipo INT e come tale dovrà essere trattato in ogni istruzione di assegnamento (sia come sorgente che come destinazione) o di valutazione di espressione.

Canali con tipo

Come precedemente anticipato, le comunicazioni tra processi OCCAM avvengono tramite scambio messaggi per mezzo di canali di comunicazione. Ricordiamo che il canale è un mezzo logico di comunicazione sin al momento del lancio dell'applicazione parallela (quindi rimane tale anche terminata la compila-

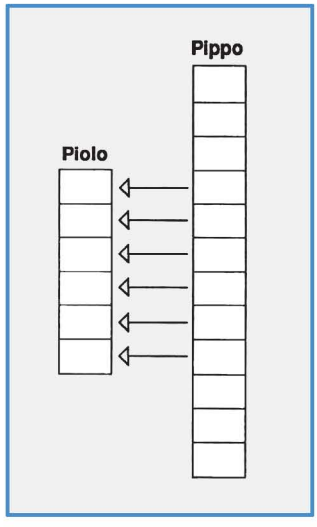


Figura 1 - In OCCAM è possibile con un semplice assegnamento trasferire parte di un array più grande in un array più piccolo e viceversa.

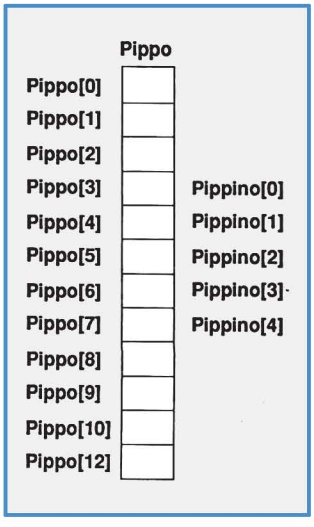


Figura 2 - Con il comando "IS" in OCCAM è possibile indirizzare porzioni di array predefiniti attraverso altri nomi (vedi testo).

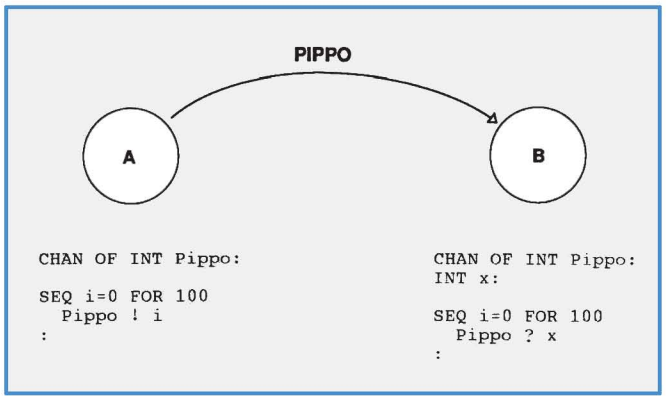


Figura 3 - In OCCAM le comunicazioni inter process sono sempre sincrone.

zione) in cui assume aspetto fisico in due distinte forme a seconda che i processi comunicanti siano in esecuzione sullo stesso o su differenti transputer. Nel primo caso, infatti, il canale viene mappato in memoria e corrisponde, in pratica, ad un buffer «a zero posizioni» in cui il processo mittente inserisce il messaggio inviato e contemporaneamente (nello stesso istante logico) il destinatario lo legge. Nel caso di processi in esecuzione su CPU diverse, il canale logico è mappato sul link fisico che collega i due transputer in questione.

Dal punto di vista della programmazione, prima di utilizzare un canale è necessario dichiarare il suo nome e il tipo di messaggio che dovrà trasferire (quest'ultimo tipo, nella definizione del canale, è detto «protocollo»). Ad esempio:

```
CHAN OF INT pippo:
```

definisce un canale di nome «pippo» sul quale «passano» solo ed esclusivamente interi (il protocollo è INT).

Volendo definire canali un po' più complessi, è sufficiente indicare il tipo del messaggio come nel caso appena visto per gli interi. Ad esempio se dobbiamo trasferire da un processo ad un altro array di 100 byte, scriveremo:

```
CHAN OF [100]BYTE NomeCanale:
```

Chiaramente esiste anche un meccanismo per definire canali con protocollo «array di lunghezza variabile», di cui conosciamo la lunghezza soltanto a tempo di esecuzione: nel momento in cui ci accingiamo ad effettuare la comunicazione da parte del mittente e che conosceremo appena completata l'operazione da parte del destinatario del messaggio. Scrivendo:

```
CHAN OF INT::[]BYTE NomeCanale:
```

dichiariamo che il canale è utilizzato per trasferire array di BYTE (potevano anche essere INT) di lunghezza variabile. Al momento della vera e propria send indicheremo la lunghezza assieme all'array da trasferire. Ad esempio:

```
NomeCanale ! 25::pippo
```

spedisce l'array pippo (precedentemente dichiarato come [25]BYTE pippo) lungo, per l'appunto, 25 byte. Nello stesso programma in un altro punto potremo utilizzare lo stesso canale per spedire un array di lunghezza diversa, ad esempio:

```
NomeCanale ! 100::pluto
```

in questo caso pluto è un array di 100 byte. Conseguentemente il processo destinatario effettuerà la sua receive indicando una variabile di tipo INT nella quale riceverà la lunghezza dell'array e un nome di un proprio array lungo almeno quanto l'array da ricevere:

```
NomeCanale ? Lunghezza::Array
```

Esistono poi delle abbreviazioni per definizioni di canali il cui tipo è abbastanza complesso. Si tratta di definire a parte il protocollo di comunicazione e poi inserire il nome di questo nelle varie definizioni di canale necessarie.

```
PROTOCOL sequenza IS INT;INT;INT;INT:
CHAN OF sequenza lista:
```

definisce un protocollo di nome sequenza corrispondente a quattro interi «sparati» l'uno dopo l'altro e subito dopo un canale di nome «lista» con protocollo «sequenza». Così il processo mittente potrà ad esempio eseguire:

```
lista ! 15; 35; 665; 455
```

e corrispondentemente il destinatario:

```
lista ? a; b; c; d
```

per ricevere nella sua variabile «a» il valore 15, nella variabile «b» il valore 35, in «c» il valore 665 e in «d» 455.

Naturalmente la sequenza di tipi indicata nella definizione di protocollo può anche essere disomogenea: l'importante che tanto il mittente quanto il destinatario si attengano strettamente alla definizione data prima di utilizzare il canale per le comunicazioni.

Protocolli variabili

Potrebbe essere utile dichiarare un canale con protocollo per così dire flessibile: più protocolli diversi da utilizzare di volta in volta a tempo di esecuzione a seconda dei casi. Questo è possibile in OCCAM grazie al cosiddetto protocollo variabile. Proviamo a definirne uno di nome «multi»:

```
PROTOCOL multi
 CASE
  a; BOOL;INT
  b; BYTE;BYTE
  c; BYTE;BOOL;INT
:
```

e poi definiamo il canale di nome «star» utilizzante il protocollo «multi» sopra definito:

```
CHAN OF multi star:
```

Il processo mittente, per effettuare la send di un messaggio sul canale star deve indicare quale dei tre protocolli possibili intende utilizzare per quella comunicazione e poi regolarsi di conseguenza.

Scrivendo ad esempio:

```
star ! a; TRUE; 250
```

utilizziamo il canale star impostando il protocollo «a» e, quindi, inviando al destinatario un valore BOOL seguito da un INT.

Il processo destinatario che utilizza il canale a protocollo variabile può comportarsi in due differenti modi: se conosce a priori quale dei tre tipi di messaggio arriverà indicherà semplicemente il caso «a» nella sua receive:

```
star ? CASE a; x; y
```

dove naturalmente «x» è una variabile BOOL e «y» una variabile INT. Diversamente, se non conosce a priori il tipo del messaggio in arrivo può indicare le varie alternative:

```
star ? CASE
  a; x; y
  b; z; t
  c; z; x; y
```

in questo modo se sul canale arriva un messaggio di protocollo «a» saranno assegnate le variabili «x» e «y», se è di tipo «b» saranno assegnate le variabili di tipo BOOL «z» e «t», se il protocollo è di tipo «c» saranno le variabili «z», «x» e «y».

Comunicazioni asincrone

In OCCAM tutte le comunicazioni inter process avvengono in maniera sincrona: l'attimo logico in cui un processo mittente effettua una send è lo stesso in cui il destinatario esegue la receive.

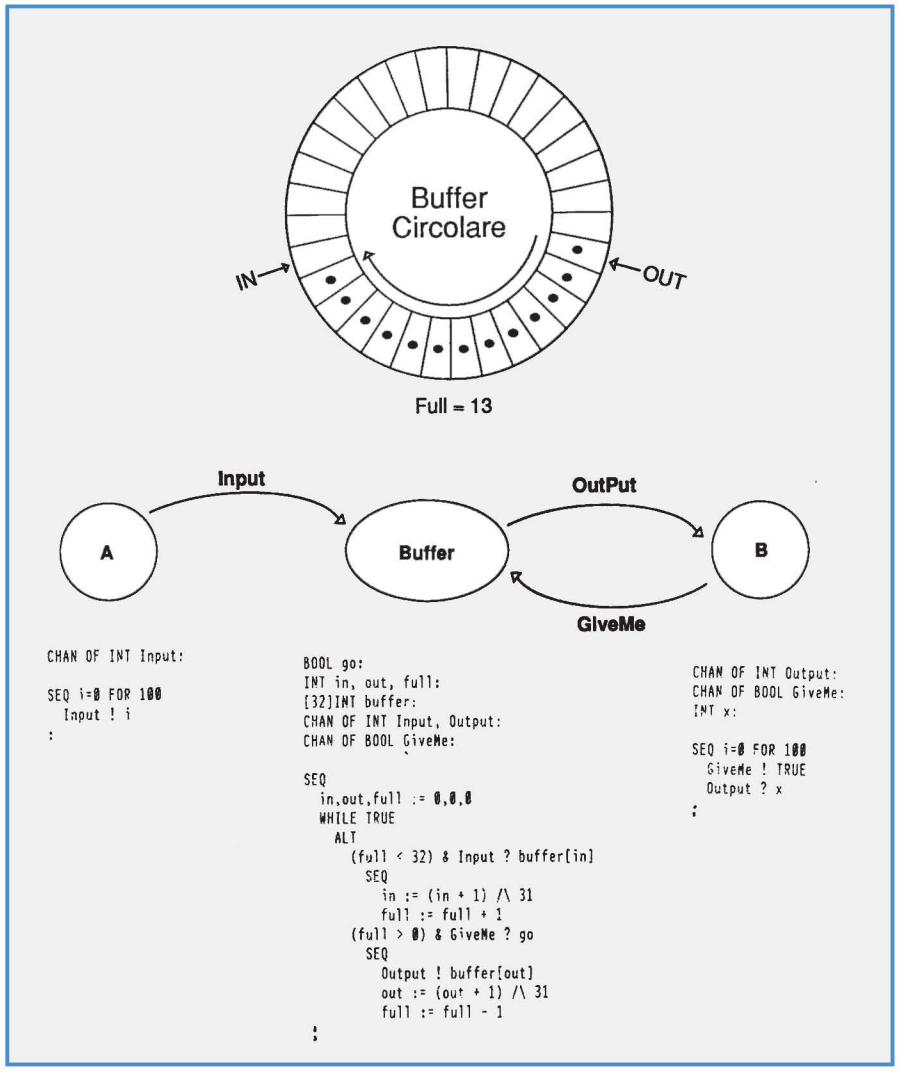
In altre parole se uno dei due processi arriva prima all'appuntamento attenderà il rispettivo partner prima di continuare per la propria strada.

Esistono però delle situazioni in cui è necessario svincolare il processo mittente dal processo destinatario permettendo al primo di lasciare il messaggio anche nel caso in cui il destinatario non sia ancora pronto a ricevere. Pensate ad esempio ad un postino che non trovando nessuno in casa lascia la posta nella cassetta delle lettere del destinatario. Quello che dobbiamo implementare è, per l'appunto, una sorta di cassetta postale in cui il mittente deposita i messaggi e dalla quale il destinatario li preleva.

Non essendo però possibile in OCCAM aggiungere direttamente buffer ai canali, l'unica soluzione possibile nella programmazione ad ambiente locale cui OCCAM si rifà è quella di aggiungere tra mittente e destinatario un processo buffer. In pratica un processo che comunica in maniera sincrona con mittente e destinatario ma disponendo all'interno di un buffer per i messaggi in transito in pratica svincola i processi comunicanti almeno fintantoché lo stesso buffer non si riempie.

Il funzionamento è schematizzato nelle figure 3 e 4. Nel primo caso il mittente e il destinatario sono direttamente comunicanti quindi la loro comunicazione è rigidamente sincrona, nel secondo caso interponendo un processo buffer la comunicazione è asincrona.

Da segnalare il fatto che il destinatario non effettua direttamente un'operazione di receive sul buffer ma in pratica



Figua 4 - Per effettuare comunicazioni asincrone è necessario aggiungere un processo buffer.

invia un messaggio di richiesta lettura ogni volta che intende ricevere un messaggio. Sarà poi il buffer ad inviarglielo una volta ricevuta la sua richiesa lettura ogni volta che intende ricevere un messaggio. Sarà poi il buffer non è vuoto.

In pratica il processo buffer riceve due tipi di messaggi ed invia un tipo di messaggio. Riceve dal processo mittente i messaggi da inserire al buffer e dal processo destinatario le richieste di lettura. A questo invierà. come detto, i messaggi via via bufferizzati.

Commentiamo brevemente il listato del processo Buffer. Le prime cinque linee sono le dichiarazioni delle variabili, array e canali che utilizzeremo nel processo. Il buffer è implementato da un array di interi da 32 posizioni e con i due puntatori «in», «out», lo utilizziamo come buffer circolare.

Ad ogni inserimento incrementiamo il puntatore «in», ad ogni estrazione il puntatore «out». Tutt'e due modulo 32 (la lunghezza del buffer) ovvero effettuando l'AND bit a bit con il valore 31 (in binario, cinque "uno"). La variabile «full» contiene costantemente il numero di messaggi bufferizzati: è utilizzata dal comando alternativo ALT nell'espressione logica delle due guardie d'ingresso.

In questo modo sono automaticamente ignorate le richieste di inserimento se il buffer è pieno («full» uguale a 32) e quelle di estrazione se il buffer è vuoto («full» uguale a zero). Per finire, attorno al comando ALT è posto un loop «infinito» formato semplicemente dal «WHILE TRUE».

Conclusioni

Questo non vuol essere un corso di OCCAM quindi non intediamo scendere più di tanto nei dettagli programmativi di tale linguaggio. Stiamo utilizzando OCCAM solo per mostrarvi qualche esempio di programmazione parallela nonché alcune soluzioni di problematiche tipiche di questo genere di programmazione. Diverso sarebbe il discorso se tutti i lettori di questa rubrica avessero a disposizione un sistema a transputer...

MC