di Andrea de Prisco

OCCAM: parallelismo puro

Può anche darsi che non abbiate mai sentito la parola OCCAM. Non potete però dire altrettanto della parola «transputer». OCCAM, per tagliare corto, è il linguaggio di programmazione dei transputer. E se quest'ultimo è il processore più «multitasking» che c'è, OCCAM è il linguaggio di programmazione più «parallelo» che c'è. Tanto parallelo che un programma scritto in OCCAM può essere formato da istruzioni da eseguire non «una dopo l'altra», ma addirittura contemporaneamente. Sì, un intero programma eseguito, dal punto di vista logico, in un solo passo: anche se le istruzioni sono centinaia o migliaia. O, addirittura, formato da pochissime istruzioni che però sottintendono un parallelismo a tempo di esecuzione pronto ad esplodere al fatidico «run».

Parallelismo «esplosivo»

Non è proprio l'argomento di questa puntata, ma tanto per farvi venire ancora di più la voglia di continuare a leggere (o, soprattutto, per non farvela passare...) anticipiamo questo mese qualche informazione in più riguardo l'ultima frase dell'introduzione di quest'articolo.

Chiunque programma i computer sa che i loop FOR-NEXT servono essenzialmente per non riscrivere tante volte la stessa porzione di codice. Se dobbiamo eseguire 5 volte una sezione di un programma, nessuno si sognerebbe mai di riscriverla 5 volte di seguito (a meno di non avere problemi di velocità), ma semplicemente la incorpora in un loop impostando questo all'esecuzione sequenziale per il numero di volte voluto.

OCCAM, da bravo linguaggio di programmazione parallela, permette di fare la stessa cosa nel «parallelo». Esistono costrutti OCCAM che permettono di lanciare in parallelo tanti processi quanti ne abbiamo impostato nel costrutto stesso: ognuno lavorerà su un certo insieme proprio di dati e il comando di ripetizione parallela terminerà quando saranno terminate tutte le istruzioni lanciate dal comando stesso e che, ripetiamo, saranno eseguite parallelamente.

Chiariamo con un esempio. Immaginate la sezione di programma Pascal-Like:

```
for i:=1 to 20
  a[i] := 0;
```

che non fa altro che annullare il vettore «a». Qualsiasi linguaggio di programmazione sequenziale azzererà i vari elementi l'uno dopo l'altro, partendo dal numero 1 fino al numero 20. In OCCAM, volendo, possiamo parallelizzare l'esecuzione scrivendo così:

```
PAR i=1 FOR 20
  a[i] := 0
```

che annullerà simultaneamente i venti elementi del vettore «a». Se non ci credete, in parte non avete tutti i torti. OCCAM non è stato ancora implementato nella sua pienezza e un'istruzione come quella appena vista, in esecuzione su un singolo transputer sarà eseguita sì in parallelo, ma in parallelismo simulato. In pratica il transputer è in grado di eseguire più processi contemporaneamente ma, ovviamente, grazie al proprio multitasking di macchina.

Quindi su un singolo transputer i venti assegnamenti saranno eseguiti l'uno dopo l'altro (anche se non necessariamente in ordine), mentre su una rete di chip di questo tipo sarà possibile l'esecuzione in parallelismo reale.

Come però vedremo meglio nei prossimi appuntamenti, il meccanismo della ripetizione parallela è tutt'altro che un'inutile esercitazione tecnologica in quanto ci permetterà, anche sul singolo chip, di implementare in maniera parallela algoritmi intrinsecamente paralleli senza la necessità di forzare un'implementazione sequenziale degli stessi...

Ne riparleremo più approfonditamente a partire al prossimo mese.

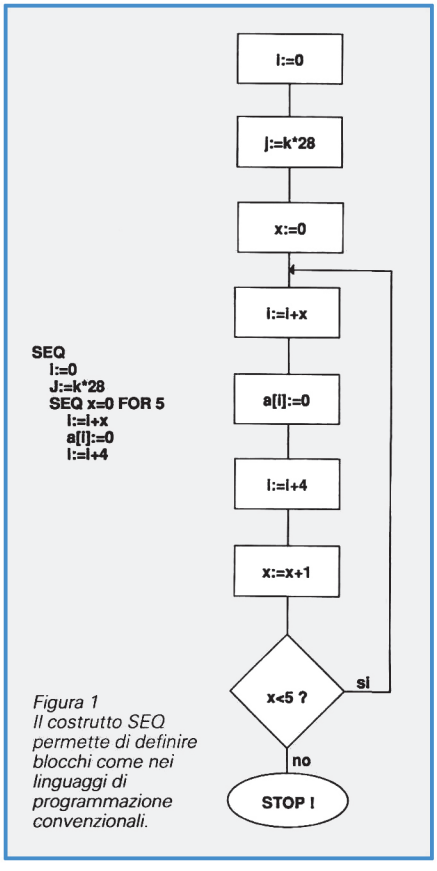

Figura 1
Il costrutto SEQ permette di definire blocchi come nei linguaggi di programmazione convenzionali.

Un programma OCCAM

La struttura di un programma OCCAM è la seguente:

```
dichiarazione:
processo
```

La dichiarazione è a sua volta... una dichiarazione oppure una dichiarazione seguita da un'altra dichiarazione. Con le dichiarazioni, come in altri linguaggi convenzionali, indichiamo nomi e tipi delle variabili, costanti, array o canali (vedi dopo) che utilizzeremo nel processo che segue.

Lo *scope* dei nomi dichiarati è sempre locale al processo che segue. Un processo può essere un processo primitivo oppure un costrutto (che combina più processi primitivi). Questi sono di solo cinque tipi: assegnamenti, processi d'ingresso, processi d'uscita, processi nulli (SKIP) e processi di terminazione (STOP).

Gli assegnamenti (anche qui nulla di nuovo) permettono di associare valori alle variabili precedentemente dichiarate. Come in Pascal il simbolo di assegnamento è := (duepunti uguale). I processi di ingresso e di uscita permettono la comunicazione.

Come nel modello CSP visto lo scorso mese, il comando di uscita è rappresentato da un punto esclamativo quello di ingresso dal punto interrogativo.

Alla sinistra di questi metteremo il nome del canale sul quale effettuare la comunicazione, alla destra l'oggetto da trasferire se si tratta di un comando d'uscita, una variabile targa per il comando d'ingresso. Ad esempio:

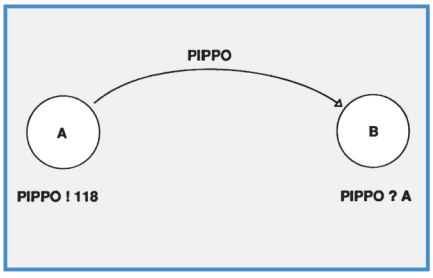

Figura 3
In OCCAM le comunicazioni sono sempre simmetriche e sincrone.
Il canale di comunicazione tra due processi è il «mezzo» di trasmissione. Se la comunicazione avviene tra processi in esecuzione sullo stesso transputer il canale è mappato in memoria, se la comunicazione avviene tra processi in esecuzione su transputer diversi il canale è implementato dal link fisico di comunicazione.

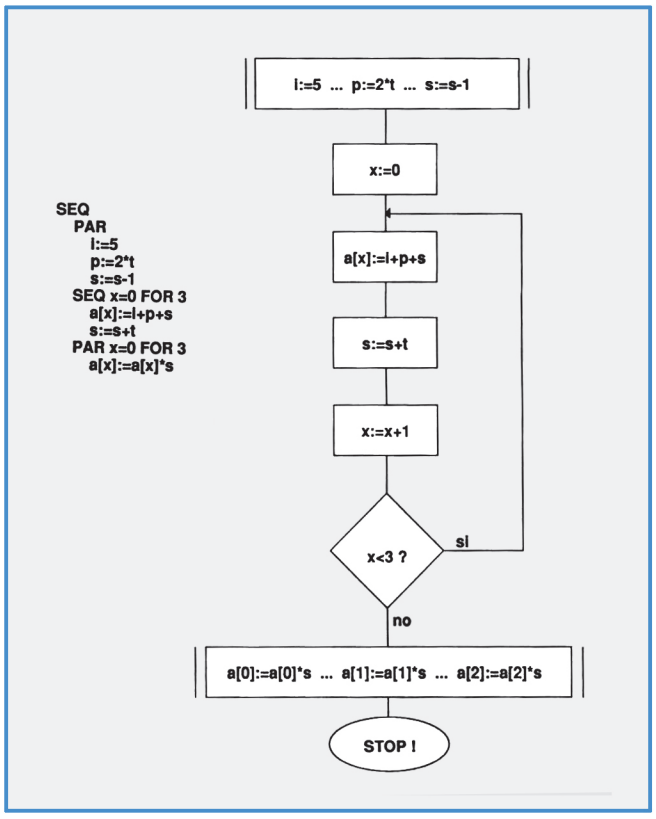

Figura 2
Il costrutto PAR permette l'esecuzione parallela di più istruzioni (in OCCAM, processi). Nel diagramma di flusso (a lato) l'esecuzione parallela è indicata dai moduli contenenti più istruzioni.

```
Pippo ! 118
```

invia sul canale «Pippo» l'intero 118, mentre:

```
Pippo ? A
```

riceve nella variabile A il valore depositato nel canale dal mittente. In OCCAM tutte le comunicazioni sono simmetriche e sincrone. Ogni canale è utilizzato da un solo mittente e un solo destinatario e la comunicazione avviene tanto per il primo quanto per il secondo nel medesimo istante logico. Se il mittente arriva all'appuntamento prima del destinatario aspetterà che questo legga il messaggio inviato prima di proseguire per la sua strada. Discorso analogo per il destinatario che aspetterà il mittente nel caso in cui dovesse eseguire la sua «receive» prima della corrispondente «send». Come per le variabili il canale, nome e tipo, deve essere dichiarato prima dell'utilizzo da parte del processo.

Il tipo del canale corrisponde al tipo del messaggio che dovrà trasportare. Occorre sottolineare che il canale, nella stesura di un programma OCCAM, è un oggetto logico che diventerà fisico solo nel momento in cui verrà lanciata l'applicazione. Se l'applicazione multitask che stiamo lanciando girerà su un singolo transputer il canale può esere inteso come una struttura implementata in memoria nella quale il processo mittente inserisce il messaggio da trasferire, il processo destinatario preleva il messaggio in arrivo. Se la stessa applicazione (senza bisogno di ricompilazione alcuna) viene invece lanciata su più transputer, processi in esecuzione su chip diversi accedendo al canale di comunicazione, di fatto utilizzeranno i link fisici di cui il transputer dispone e con i quali dialoga con gli altri transputer.

I principali costrutti (per combinare più processi primitivi) sono:

```
SEQ
IF
CASE
WHILE
PAR
ALT
```

Il primo, il più semplice, permette di eseguire più processi in sequenza. Ad esempio:

```
SEQ
  a:=1
  b:=2
  Canale1 ! a+b
  Canale2 ? R
```

i quattro processi indicati dopo SEQ (attenzione che in OCCAM quelli che sembrano normali statement sono processi essi stessi!) saranno eseguiti in sequenza, come in un normale linguaggio di programmazione. In pratica SEQ identifica un blocco ed è esso stesso (a sua volta) un processo. Quindi possiamo associare a questo alcune dichiarazioni locali che non saranno più valide una volta eseguita la sequenza.

Dove termina il costrutto SEQ? Ovvero: qual è l'ultima istruzione del SEQ in cui ci troviamo? In OCCAM i blocchi sono tutti delimitati dall'indentazione: due caratteri a destra per iniziare una sequenza, due caratteri a sinistra per terminarla. Così, il frammento di codice:

```
SEQ
  a:=1
  b:=2
  Canale1 ! a+b
  Canale2 ? R
c:=0
d:=a+R
```

è in pratica formato da tre processi. Un primo processo SEQ (a sua volta composto da quattro processi) e due processi di assegnamento. Chiaramente il tutto deve poi essere combinato per formare (in qualsiasi altro modo) un ulteriore unico processo combinato dai tre processi appena descritti. Se ad esempio questi tre processi devono essere eseguiti in sequenza basterà aggiungere un altro SEQ all'inizio scrivendo:

```
SEQ
  SEQ
    a:=1
    b:=2
    Canale1 ! a+b
    Canale2 ? R
  c:=0
  d:=a+R
```

Proviamo allora a completare questo codice per farlo diventare un programma OCCAM. Mancano innanzitutto le dichiarazioni, immaginiamo di voler limitare lo *scope* della variabile «b» al solo SEQ interno; tutte le altre dichiarazioni sono, per così dire, globali:

```
INT a, c, d, R:
CHAN OF INT Canale1,Canale2:
SEQ
  INT b:
  SEQ
    a:=1
    b:=2
    Canale1 ! a+b
    Canale2 ? R
  c:=0
  d:=a+R
```

La lettura del codice dovrebbe essere abbastanza semplice così come riconoscere in questo la struttura base fornita all'inizio. La dichiarazione iniziale è formata da due dichiarazioni, una per le variabili di tipo INT l'altra per i canali anch'essi di tipo INT. Segue l'unico processo SEQ a sua volta formato da altri processi: la dichiarazione della variabile «b» è locale al SEQ più interno quindi non esiste più una volta terminata l'esecuzione di questo.

Detto ciò non ci rimane che svelarvi il funzionamento degli altri costrutti.

Rapidamente:

```
IF
  {Espressione Booleana1}
    Processo1
  {Espressione Booleana2}
    Processo2
   :
   :
  {Espressione BooleanaN}
    ProcessoN
```

è il costrutto condizionale. In pratica viene eseguito il primo processo (che può essere a sua volta primitivo o combinato come ci pare) la cui espressione booleana ha valore TRUE. Da segnalare che se nessuna espressione booleana è verifica-

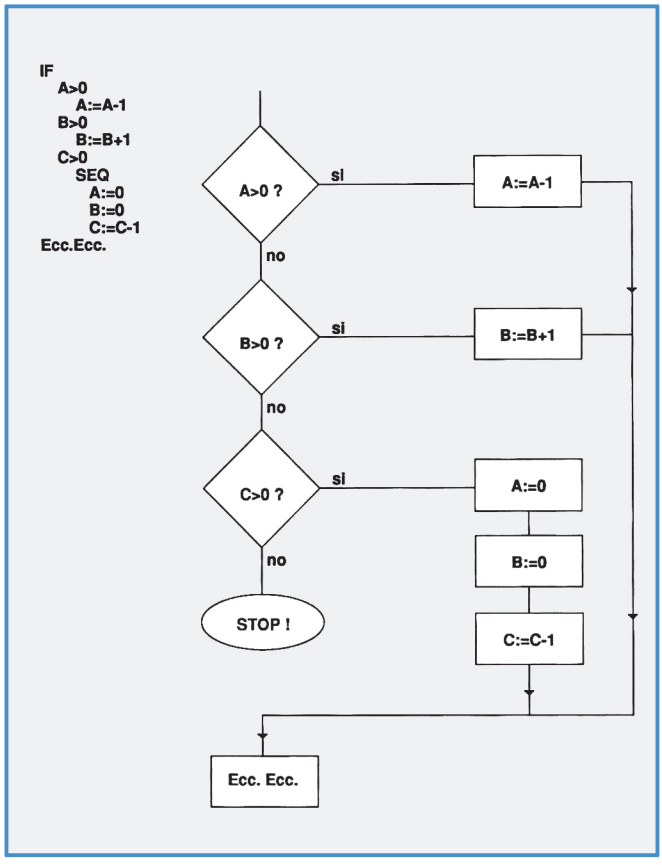

Figura 4 — Se nel costrutto IF tutte le espressioni booleane danno risultato FALSE il processo in esecuzione si ferma.

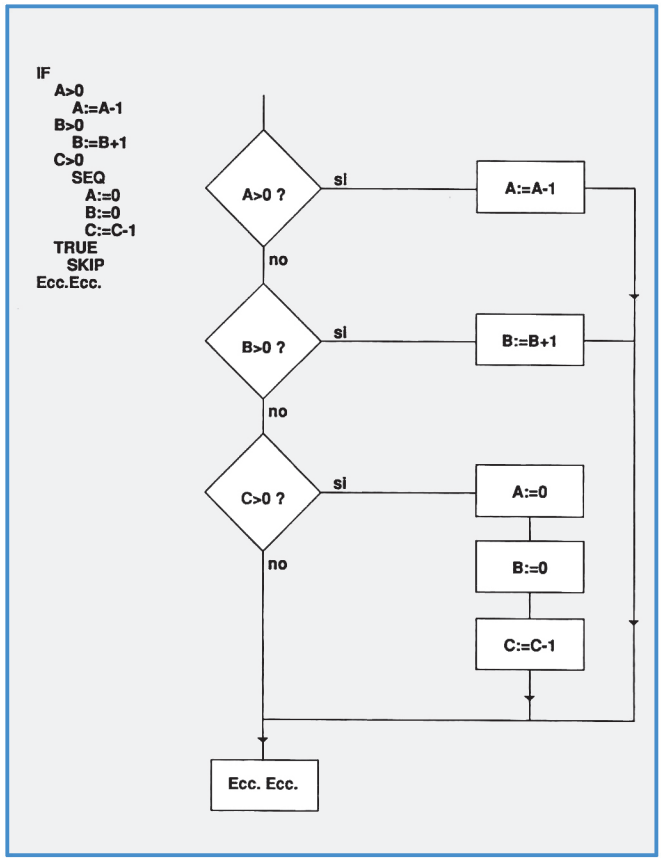

Figura 5 — Prevedendo una «uscita d'emergenza» (TRUE... SKIP) il costrutto IF ha un comportamento più normale (vedi anche figura 4).

ta l'intero programma si blocca terminando lì l'esecuzione (comportamento simile al processo primitivo STOP). Per questo è conveniente prendere gli opportuni provvedimenti prevedendo una uscita finale con espressione booleana costante TRUE magari con processo SKIP se non dobbiamo eseguire nulla in questo caso. Esempio:

```
IF
  a>0
    a:= a-1
  b>0
    b:=b+1
  c>0
    SEQ
      a:=0
      b:=0
      c:=c-1
  TRUE
    SKIP
```

Alla terza scelta è stato associato un processo combinato (un SEQ) quindi nel caso in cui le prime due espressioni danno esito falso e «c» è maggiore di zero saranno eseguite le tre linee indicate. Come caso finale abbiamo inserito un TRUE...SKIP che ci permette di continuare l'esecuzione nel caso in cui nessuna delle tre espressioni logiche dia esito TRUE.

Il costrutto CASE è molto simile (come al solito) all'IF: è eseguito il primo processo corrispondente all'espressione che ha lo stesso valore della variabile del CASE. Come nell'IF è necessario che almeno una alternativa sia verificata pena il blocco del programma. Per forzare una uscita d'emergenza si usa lo statement ELSE. Esempio:

```
CASE x
  1
    x:=0
  2
    x:=5
  ELSE
    x:=x-1
```

Se x è uguale ad 1 sarà eseguito x:=0, se x è uguale a 2 sarà eseguito x:=5, in tutti gli altri casi x sarà diminuito di 1.

Il costrutto WHILE permette di effettuare loop su condizione. Ad esempio:

```
WHILE a>0
  SEQ
    x:=x+2
    a:=a-x
```

esegue il processo SEQ (formato a sua volta da due processi di assegnamento) fintantoché «a» è maggiore di zero.

Il costrutto PAR (già precedentemente citato) permette di eseguire tutti i processi indicati in parallelo. Naturalmente devono esistere le condizioni affinché l'esecuzione in parallelo sia possibile. È sintatticamente simile al costrutto SEQ. Ad esempio:

```
PAR
  a:=a+1
  Canale ! b
  c:=2
```

esegue in parallelo i tre processi indicati. Si sarebbe avuto errore a tempo di compilazione se, ad esempio, avessimo scritto:

```
PAR
  a:=a+1
  Canale ! a
  c:=2
```

(notare la «a» al posto della «b» nel processo di uscita). In questo caso non è possibile l'esecuzione in parallelo dal momento che la variabile «a» è contemporaneamente modificata nel primo processo ed utilizzata nel secondo.

Per finire, il costrutto ALT permette di effettuare il controllo del non determinismo come già visto negli articoli che hanno preceduto questa puntata.

La forma più semplice del costrutto ALT non prevede, per la verità, alcun controllo ma permette di attendere simultaneamente messaggi da più canali. Non appena arriva un messaggio su uno dei canali indicati, sarà letto il messaggio ed, eventualmente, eseguito un apposito processo. Esempio:

```
ALT
  chan1 ? x
    {processo 1}
  chan2 ? y
    {processo 2}
  chan3 ? z
    {processo 3}
```

La versione «completa» del costrutto ALT permette di aggiungere una espressione logica ad ogni processo di ingresso in modo da ottenere una guardia d'ingresso. Ad esempio:

```
ALT
  (ax>0) & chan1 ? x
    {processo 1}
  (ay>0) & chan2 ? y
    {processo 2}
  (az>0) & chan3 ? z
    {processo 3}
```

In questo modo è possibile restringere il range di canali da testare in funzione del valore delle espressioni booleane che abbiamo inserito prima di ogni processo di ingresso: saranno testati solo i canali le cui espressioni booleane hanno dato esito positivo. Se tutte le espressioni booleane danno esito negativo il programma si blocca: anche in questo caso è necessario prevedere una via d'uscita tramite la solita costante TRUE.

Replicator

Non si tratta di un film di fantascienza anche se, per alcuni versi potremmo definirla tale. Ai costrutti SEQ, PAR, ALT e IF è possibile aggiungere un indice in modo da far «ciclare» il costrutto stesso. In questa sede analizzeremo solo SEQ e PAR che utilizzeremo più spesso in seguito, rimandando le altre spiegazioni al momento più opportuno. Il costrutto:

```
SEQ variabile = valore FOR volte
  {processi}
```

esegue i processi indicati tante volte quante indicate dopo la parola chiave FOR assegnando alla variabile indicata il valore iniziale di volta in volta incrementato di uno. Ad esempio:

```
SEQ i=0 FOR 5
  x = i*2
  canale ! x
```

equivale a scrivere:

```
SEQ
  x = 0
  canale ! x
  x = 2
  canale ! x
  x = 4
  canale ! x
  x = 6
  canale ! x
  x = 8
  canale ! x
```

Il costrutto:

```
PAR variabile = valore FOR volte
  {processi}
```

Esegue i processi indicati in parallelo tante volte quante indicate dopo la parola chiave FOR utilizzando in ogni processo lanciato un diverso indice della variabile indicata. Come nel normale PAR non deve verificarsi che una qualsiasi variabile modificata da un processo parallelo sia contemporaneamnte utilizzata da un altro processo parallelo.

Conclusioni

Non facciamo in tempo, questo mese, a mostrarvi un esempio di programma OCCAM. Tenete duro fino al prossimo MC: intanto, tra un bagno e una rilassante passeggiata in riva al mare, studiatevi un po' tutto quello che abbiamo mostrato in questa puntata.

Vi assicuro comunque che ne vedremo delle belle...

MC