di Andrea de Prisco

OCCAM: parallelismo «esplosivo»

Il nostro viaggio all'interno di OCCAM continua questo mese con un argomento che lascerà, i più, sconcertati. Parleremo del cosiddetto «parallelismo esplosivo», eufemismo non troppo scientifico che utilizziamo per indicare quella capacità di OCCAM di generare processi paralleli a tempo di esecuzione ed in numero noto solo in quel momento. Vedremo dunque come con poche righe di codice si riescono ad implementare meccanismi ad elevato parallelismo e dalla potenza a dir poco impressionante

L'augurio

OCCAM è, fortunatamente per certi versi, purtroppo per altri, più potente dei transputer sui quali gira. In hardware è stato sì possibile implementare tutte le funzionalità richieste, ma alcune di queste, per ovvie ragioni, con qualche limitazione.

Ci riferiamo essenzialmente al parallelismo dei programmi OCCAM che, lanciati sul singolo transputer sono eseguiti «solo» in parallelismo simulato, mentre per un'esecuzione su rete di transputer è necessario allocare staticamente, sui più chip, i vari processi.

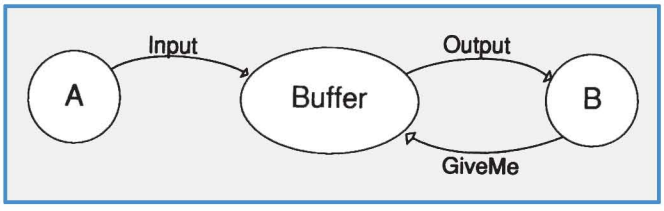
I link fisici, infatti, sono utilizzati essenzialmente per scambiare messaggi tra processi in esecuzione su nodi di elaborazione diversi, ma non per far migrare a tempo di esecuzione processi da eseguire parallelamente. Questo sicuramente a causa del fatto che, sempre attualmente, l'overhead dovuto a questa sorta di migrazione di processi sarebbe troppo oneroso e magari (anzi sicuramente) annullerebbe i vantaggi della parallelizzazione reale.

Ma... c'è un «ma». Anche se pieno zeppo di fantaSCIENZA. Quando i progettisti della INMOS hanno dato un nome alla loro creatura hanno scelto la parola transputer. Fusione di altri due termini molto noti che sono «transistor» e «computer». Ciò ad indicare che il «chippetto» è un computer (e anche buono) ma nasce per essere utilizzato come una specie di transistor ovvero come mattoncino «materia prima» di qualcosa di ben più potente del transputer stesso: la rete di transputer. Questa, ha dalla sua il fatto di poter essere estesa infinitamente e con essa la potenza di calcolo che può offrire. L'analogismo è praticamente perfetto: come con i transistor possiamo ipotizzare (e realizzare...) circuiti elettronici complicatissimi formati da decine, migliaia, milioni di semiconduttori, con i transputer è possibile costruire sistemi di calcolo a parallelismo massiccio della potenza desiderata, senza limiti né teorici né tecnici.

E fin qui non stiamo ancora sognando. Ma proviamo a spingere la nostra fantasia oltre. Qual è stato il cosiddetto «passo successivo» per il mondo dei transistor? La risposta è semplice: il circuito integrato. Un «pezzetto di silicio» sul quale vengono «intarsiati» migliaia, oggi anche milioni, di transistor per concentrare in pochissimo spazio circuiti elettronici dalle dimensioni e costi svariati ordini di grandezza superiori.

E chi ci vieta di augurare al transputer la stessa sorte? O forse non è facile ipotizzare intere reti di transputer sul singolo chip? Magari prima 8, 16, 32 unità sullo stesso silicio ma più in là (quando probabilmente non si tratterà più di silicio...) migliaia o anche milioni di unità di calcolo pronte ad assecondare, e quindi a soddisfare, le già citate «esplosioni» di parallelismo possibili in OCCAM.

Figura 1



```
[100]CHAN OF INT coda: -- si dichiara un array di canali di interi
                       -- ogni processo lanciato utilizzera' un
                       -- canale di ingresso e un canale d'uscita
                       -- di quest'array implementando una pipeline
PAR
  PAR i=0 FOR 100        -- lancio 100 volte il seguente processo
    WHILE TRUE
      INT x:
      SEQ
        coda[i  ] ? x          -- leggo il valore da bufferizzare
        coda[i+1] ! x          -- spedisco il valore da bufferizzare

  INT x:                       -- primo processo del buffer che
  SEQ                          -- interagisce col processo mittente
    input ? x
    coda[0] ! x

  INT x:                       -- ultimo processo del buffer che
  SEQ                          -- interagisce col processo destinatario
    coda[99] ? x
    output ! x

:
```

Figura 2

Un buffer parallelo

Sul numero scorso di MC, quale primo esempio di programmazione OCCAM abbiamo mostrato come risolvere il problema della sincronicità delle comunicazioni interponendo tra processo mittente e processo destinatario un processo Buffer. All'interno di questo (figura 1) i messaggi inviati dal processo mittente venivano parcheggiati in un array di messaggi (utilizzato circolarmente) in attesa che questi venissero richiesti dal processo destinatario. Per utilizzare, infatti, un comando alternativo con guardie d'ingresso era necessario che il processo destinatario inviasse una richiesta al buffer per ricevere un nuovo messaggio in arrivo. Quindi la soluzione, per quanto valida dal punto di vista OCCAM, aveva il piccolo difetto di rendere necessaria anche una lieve modifica al processo destinatario prima di utilizzare il buffer vero e proprio.

La soluzione che mostreremo questo mese sposa le caratteristiche di OCCAM in maniera più spinta utilizzando il già citato «parallelismo esplosivo» ed avendo anche il vantaggio di non richiedere alcuna modifica al processo destinatario che vedrà arrivare i suoi messaggi esattamente come se non esistesse alcun buffer tra esso e il processo mittente.

In figura 2 è mostrato il listato del nuovo programma buffer, in figura 3 sono rappresentati i processi coinvolti una volta lanciato il processo buffer. Vediamo come funziona. Quello che succede è molto semplice: invece di utilizzare un array di messaggi in cui depositare gli arrivi e prelevare le partenze utilizzando un unico canale di ingresso ed un altrettanto unico canale di uscita, si utilizza direttamente un array di diciamo 100 canali utilizzati da altrettanti processi creati dal processo buffer. Ognuno di questi processi figli non farà altro che ricevere un messaggio dal canale (i)esimo e rispedire immediatamente il messaggio ricevuto sul canale (i+1)esimo. Pensate, ad esempio, ad un certo numero di operai che scaricano, manualmente, un camion di mattoni passandoseli di mano in mano dal cassone sino al punto dove è necessario stiparli. L'operaio «mittente» sarà quello posizionato sul camion, l'operaio «destinatario» quello che posiziona uno dopo l'altro tutti i mattoni ricevuti. Tutti gli operai «intermedi» bufferizzano l'operazione così il mittente può iniziare a «spedire» anche se il destinatario non è pronto essendo costretto ad arrestarsi solo nel caso in cui il destinatario non riceve e tutti gli operai sono rimasti con un mattone in mano.

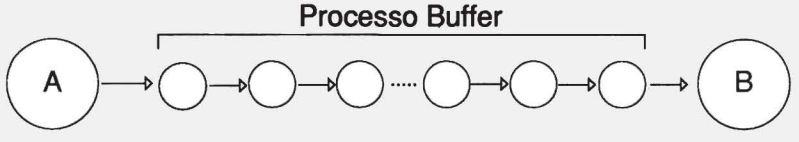


Figura 3

Nel listato di figura 2 troviamo innanzitutto la dichiarazione dell'array di canali necessari, come detto, alla bufferizzazione. Segue un primo PAR che lancia in parallelo i tre processi in esso contenuti: il primo è in buffer vero e proprio che commenteremo tra breve, il secondo e il terzo l'interfaccia di ingresso e d'uscita del buffer che dialogano con i veri processi «A» e «B» che devono comunicare. Corrispondono, in pratica, ai due operai citati precedentemente: il primo sul camion il secondo nel luogo dove occorre stipare i mattoni in arrivo.

Per quanto riguarda il buffer vero e proprio, notiamo subito che esso è formato da un «replicator» applicato al costrutto PAR. Come spiegato due numeri fa con questo meccanismo è possibile creare un numero qualsiasi di processi figli ognuno dei quali identificato da un ben preciso indice della variabile di ciclo. Come i «FOR» dei vari linguaggi sequenziali con la «semplice» differenza che le varie istanze di ciclo non sono più eseguite sequenzialmente ma trasformmate in altrettanti processi e in quanto tali eseguiti parallelamente. Ognuno dei processi creati (sempre figura 2) è un loop infinito (WHILE TRUE) su una coppia di operazioni che prelevano il messaggio in arrivo sul già citato canale (i)esimo e spediscono il messaggio così ricevuto sul canale (i+1)esimo.

Sorting parallelo

L'esempio appena visto funziona sì, ma crediamo che mai a nessuno venga in mente di implementarlo per davvero. Si tratta, infatti, del monumento allo spreco e, per questo, anche all'inefficienza. Quindi didatticamente validissimo ma ben ancorato a lavagne e pubblicazioni varie che, come noi, devono solo mostrare l'essenza di OCCAM. Ha infatti la caratteristica di essere sì particolarmente parallelo ma utilizza questo parallelismo non per migliorare le prestazioni del sistema mittente-destinatario ma solo per semplificare al massimo la stesura del programma. Pensate al fatto che un messaggio, nel caso di buffer vuoto e destinatario pronto a ricevere, prima di arrivare a destinazione deve «attraversare» più di cento processi (quante sono le posizioni del buf-

fer utilizzato) con altrettante operazioni di send e receive da eseguire. Non ve l'abbiamo confidato subito per non togliervi il gusto di studicchiarvi con noi il listato di figura 2.

Il prossimo esempio è, però, molto più serio. Discendente diretto del programma appena visto, ha l'importante proprietà di sfruttare il parallelismo nel giusto verso: incrementare la potenza di calcolo per ottenere in minor tempo i risultati voluti. È un semplice programma di ordinamento numerico che ha la particolare caratteristica di avere un tempo di computazione eccezionalmente basso, dovuto al fatto che grazie al parallelismo dei processi più operazioni vengono effettuate contemporaneamente.

Vedete a cosa servirebbe, subito, un chippetto con un milione di transputer dentro ? Ad ordinare, al volo, fino ad un milione di dati.

Già, i più svegli avranno già capito, è necessario creare un processo per ogni dato da ordinare: solo così, con un funzionamento pipeline del sorting è possibile ridurre drasticamente il tempo di esecuzione.

Come detto prima il funzionamento è simile a quello del buffer appena descritto. Ogni processo figlio ha però al suo interno una variabile temporanea nella quale mantiene il dato appena letto per confrontarlo col dato successivo. Di volta in volta leggerà un dato e con-

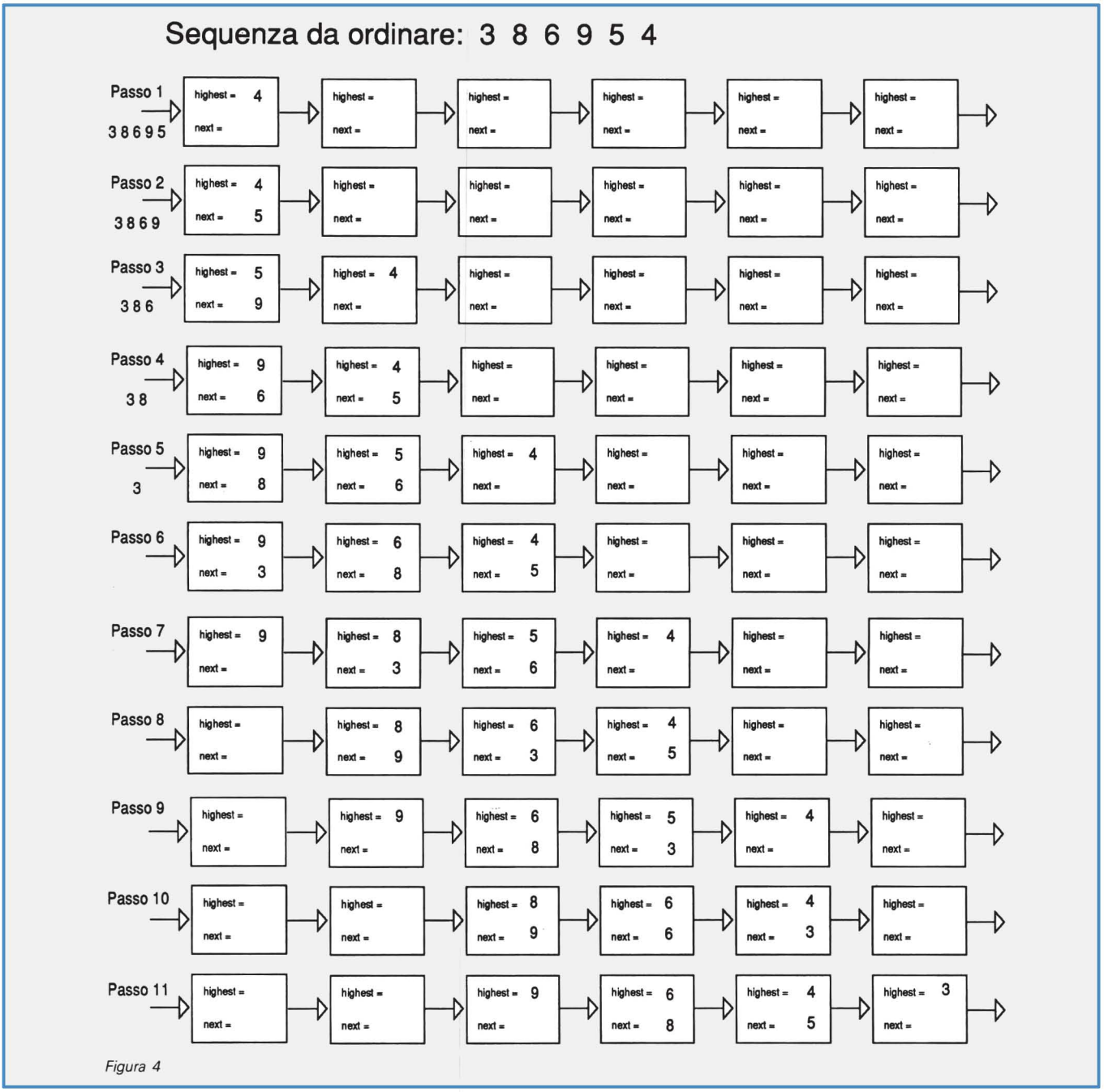


Figura 4

Figura 5A

```
INT highest:          -- variabile per il valore piu' alto ricevuto
SEQ
  input ? highest     -- il primo valore inizializza highest
  SEQ j=0 FOR 5       -- per i rimanenti 5 valori...
    INT next:
    SEQ
      input ? next          -- leggo il prossimo valore
      IF
        next <= highest     -- se e' minore di highest
          output ! next     -- lo inoltro al processo successivo
        next > highest      -- se e' maggiore di highest
          SEQ
            output ! highest  -- inoltro quest'ultimo
            highest := next   -- e assumo un nuovo highest
  output ! highest  -- per finire inoltro il "vero" highest
:
```

Figura 5B

```
[101]CHAN OF INT pipe: -- si dichiara un array di canali di interi
                       -- ogni processo lanciato utilizzera' un
                       -- canale di ingresso e un canale d'uscita
                       -- di quest'array implementando una pipeline

PAR i=0 FOR 100        -- lancio 100 volte il seguente processo
  INT highest:         -- variabile per il valore piu' alto ricevuto
  SEQ
    pipe[i] ? highest  -- il primo valore inizializza highest
    SEQ j=0 FOR 99     -- per i rimanenti 99 valori...
      INT next:
      SEQ
        pipe[i] ? next         -- leggo il prossimo valore
        IF
          next <= highest      -- se e' minore di highest
            pipe[i+1] ! next   -- lo inoltro al processo successivo
          next > highest       -- se e' maggiore di highest
            SEQ
              pipe[i+1] ! highest  -- inoltro quest'ultimo
              highest := next      -- e assumo un nuovo highest
    pipe[i+1] ! highest  -- per finire inoltro il "vero" highest
:
```

frontandolo col dato immagazzinato propagherà quello più alto o più basso tra i due a seconda se l'ordinamento è ascendente o discendente. Un po' come se gli operai di prima scaricassero pietre dal camion con la raccomandazione di far giungere all'ultimo della coda le pietre in ordine di grandezza. Certo, ci vorrebbero tanti operai quante sono le pietre, ma raggiungerebbero il loro scopo in un batter d'occhio e soprattutto senza fare confusione: ogni operaio la prima pietra che riceve la conserva in tasca e man mano che passano le altre pietre esegue un rapido test ponendole una per una nella tasca attualmente vuota e cedendo al successivo la più leggera tra le pietre nelle due tasche.

Tornando alla realtà, in figura 4 è mostrato un esempio «grafico» del procedimento di sort parallelo (pipeline per la precisione) applicato ad una sequenza di sei interi da ordinare. Saranno necessari 6 processi (il cui codice OCCAM è listato in figura 5A) ognuno dei quali proclamerà, in al più cinque passi il massimo elemento transitato al suo interno. Naturalmente, dato che il procedimento si svolge «in parallelo» in ogni passo saranno effettuati contemporaneamente più confronti all'interno di processi differenti ma eseguendo questi simultaneamente comunque utilizzeranno il tempo necessario ad un singolo confronto.

Seguiamo quello che succede in figura 4. I dati da ordinare sono:

3 8 6 9 5 4

con in testa il 4 e in coda il 3. Al primo passo il 4 è inoltrato al processo uno che, ricevendo il suo primo dato, lo pone d'uffico nella sua variabile temporanea «highest». Al passo due anche il 5 giunge al processo uno potendo così effettuare il primo confronto tra i primi due dati. Naturalmente 4 è più piccolo di 5 quindi il primo è spedito al processo due (che, come prima, lo archivia d'ufficio) mentre il secondo diventa il nuovo «highest» del processo uno. Al terzo passo «entra» il 9 che prende il posto del 5 che viene spedito al processo due mentre arriva anche un nuovo dato, il 6. Al passo quattro è così possibile effettuare contemporaneamente due confronti, all'interno del processo uno e del processo due. Il procedimento «pipeline» si ripete per tutti i rimanenti numeri da ordinare all'interno di tutti i processi da attraversare. Notate, già al passo sei, la possibilità di effettuare tre confronti contemporaneamente (cioè tutti i dati coinvolti simultaneamente in un confronto). All'uscita dell'ultimo processo i dati saranno in ordine crescente e cioè nell'esatta sequenza:

9 8 6 5 4 3

Per finire diamo uno sguardo al programma di figura 5B. È la versione definitiva del sort parallelo appena descritto e mette in ordine 100 numeri. Naturalmente per sort più consistenti è sufficiente aumentare tale valore nella dichiarazione iniziale, nel PAR successivo e nel SEQ subito dopo. Highest, come detto prima, è la variabile che contiene costantemente il più grande numero passato all'interno del processo. La seconda variabile, next, contiene il dato in transito da confrontare con highest. Quindi il processo (i)esimo preleva dal canale (i)esimo un dato depositandolo momentaneamente nella variabile next, confronta questo con il contenuto di highest, se è più piccolo spedisce sul canale (i+1)esimo il valore di next se è più grande quello di highest assegnando poi a questo il dato trattenuto. Quando tutti i dati sono passati, non resta che inoltrare anche highest che, a quel punto, conterrà il numero più grande dell'intera sequenza ricevuta.

MC