POOL2
Un linguaggio parallelo orientato agli oggetti

seconda parte

di Giuseppe Cardinale Ciccotti

Riprendiamo l'analisi del linguaggio POOL2 già introdotta il mese passato. Ricordiamo ai lettori meno assidui che il Parallel Object Oriented Language 2 nasce nell'ambito del progetto europeo ESPRIT 415.
L'affermazione della programmazione orientata agli oggetti e l'apparire di nuove macchine parallele rendono particolarmente interessanti il tipo di linguaggi a cui appartiene il POOL2.
In questa puntata in particolare ci proponiamo di valutare in che modo venga implementato il parallelismo all'interno del linguaggio. La scorsa puntata abbiamo avuto occasione di puntualizzare gli aspetti che il POOL2 ha mutuato dai paradigmi della programmazione orientata agli oggetti.
Per i più curiosi ci preoccuperemo di allegare un esempio di programmazione in modo da illustrare praticamente i concetti che esponiamo

Il parallelismo in POOL2

La maggior parte dei nostri lettori avrà sicuramente coscienza del fatto che la stragrande maggioranza delle architetture hardware destinate ad eseguire delle sequenze di codice sono strettamente seriali; eseguono cioè soltanto una istruzione di basso livello per volta. I lettori affezionati ad Appunti di Informatica e più attenti a ciò che accade nel mondo della microelettronica sanno anche che la tecnologia offre a chi voglia intraprendere l'avvincente avventura della programmazione parallela, nuove macchine in grado di eseguire più di una istruzione per volta. Se tentiamo come possibile di far cooperare più processori insieme per l'esecuzione di un programma sarà necessario che essi comunichino tra di loro a meno di qualche raro caso fortunato in cui l'algoritmo è così modulare da poter essere separato in parti totalmente indipendenti.

Un meccanismo assai noto per implementare le comunicazioni è quello dello «scambio dei messaggi». Questa tecnica è flessibile e potente tanto che trova diverse realizzazioni in special modo quando non sia prevista una spiccata specializzazione per i compiti a cui è destinata la macchina.

I linguaggi paralleli prevedono delle strutture e delle istruzioni particolari per gestire questo scambio di messaggi; tuttavia asserire che scambio di messaggi implichi di per sé la realizzazione del parallelismo è un errore.

Il caso della programmazione orientata agli oggetti ne è un chiaro esempio: tutti i linguaggi O.O. fanno esplicito riferimento al fatto che gli oggetti comunicano mandandosi messaggi, tuttavia la maggior parte dei linguaggi di programmazione parallela ha natura sequenziale, proprio perché destinata ad architetture seriali.

Schematicamente si può dire che essi seguono queste tre restrizioni.
1) L'esecuzione parte contemporaneamente all'attivazione di un oggetto.
2) Un oggetto che abbia mandato un messaggio, rimane in attesa fino a che il risultato del messaggio non sia ritornato.

```
SPEC UNIT Prio_Queue

CLASS PQ
%%Le istanze di questa classe sono code di priorità che %%memorizzano
interi

ROUTINE new() : PQ
%%Crea una nuova coda di priorità vuota

METHOD put(n : Int) : PQ
%% Memorizza l'intero n nella coda

METHOD get() : Int
%%Estrae l'intero maggiore dalla coda
%%Questo metodo non risponde se la coda è vuota

END PQ
```

Figura 1 - Specification Unit dove è definita la classe PQ e tutto quello che serve per utilizzarla.

```
IMPL UNIT Prio_Queue

CLASS PQ
%%Le istanze di questa classe sono code di priorità che
%%memorizzano interi

%%La routine new che crea la coda vuota, è definita implicitamente
%%perciò non necessita di ulteriori dichiarazioni

VAR   max : Int   %%Il maggiore degli elementi in coda
      rest : PQ   %%Coda che memorizza tutti gli altri elementi

%%Entrambe le variabili sono automaticamente inizializzate a NIL


METHOD put(n : Int) : PQ
%% Memorizza l'intero n nella coda
BEGIN
      RETURN SELF;       %%Ritorna il risultato come riferimento a se
                         %%stesso: il mittente può continuare
      IF max == NIL
      THEN max := n;
           IF rest == NIL THEN rest := PQ.new () FI
      ELSEIF max >= n
      THEN rest ! put (n)
      ELSE rest ! put (max)
           max :=n
      FI
END put


METHOD get() : Int
%%Estrae l'intero maggiore dalla coda
%%Questo metodo non risponde se la coda è vuota
BEGIN
      RESULT max; %%Ritorna il massimo intero della coda
      max := rest ! get_largest_or_NIL ()
END get


METHOD get_largest_or_NIL () : Int
%%Ritorna NIL se la coda è vuota. Altrimenti cancella il massimo e
%%lo ritorna.
BEGIN
      RESULT max;
      IF max == NIL
      THEN max := rest ! get_largest_or_NIL ()
      FI
END get_largest_or_NIL


BODY
      DO          %%Ciclo infinito
           IF max == NIL
           THEN ANSWER (put, get_largest_or_NIL)
           ELSE ANSWER (put, get, get_largest_or_NIL)
           FI
      OD
YDOB

END PQ
```

Figura 2 - Implementation Unit della Coda di priorità, i tre metodi utilizzati e il semplice body.

3) Un oggetto è attivo soltanto quando sta eseguendo un metodo in risposta ad un messaggio ricevuto.

Sotto tali condizioni appare evidente che ci può essere soltanto un oggetto attivo per volta e che molto spesso il controllo è trasferito da un oggetto ad un altro.

Ci sono diversi modi in cui è possibile introdurre il parallelismo nei linguaggi orientati agli oggetti.

La prima possibilità è quella di inserire il concetto di «processo» e le strutture per realizzarlo. Per processo si intende qui una o più procedure indipendenti che possano essere eseguite contemporaneamente ad altre. Il concetto di processo rilascia la restrizione numero 1); infatti ci potranno essere più processi attivi che eseguono un programma O.O. allo stesso tempo. Questi processi agiscono sulla stessa collezione di oggetti; al limite è possibile che eseguano il medesimo metodo nello stesso oggetto nello stesso momento. Questa maniera di utilizzare il parallelismo è stata adottata da qualche linguaggio che in origine era strettamente sequenziale come lo Smalltalk-80.

Se quest'approccio può sembrare attrattivo da un punto di vista teorico, non lo è in pratica. Esso infatti non risolve uno dei punti più controversi della programmazione parallela: il non determinismo associato con l'esecuzione di un programma. Purtroppo con questo sistema, la velocità con cui vengono eseguiti i processi non è prevedibile a priori né ancora peggio la velocità stessa di esecuzione è riproducibile. Essa dipende direttamente dall'ordine in cui vengono eseguiti i processi costituenti il programma; il numero di possibili ordini di esecuzione cresce molto rapidamente con il numero dei processi e delle interazioni fra essi. Oltre a ciò l'indeterminazione dell'esecuzione comporta l'indeterminazione dei risultati dell'esecuzione stessa.

Un certo livello di non determinismo è in qualche modo fisiologico in modo da sfruttare con flessibilità il parallelismo hardware disponibile, ma se diventa troppo elevato risulta quasi impossibile assicurare la correttezza dei programmi.

I principi più importanti per ridurre il non determinismo sono la sincronizzazione e la mutua esclusione. I linguaggi che utilizzano i processi necessitano allora di ulteriori strutture per ottenere la sincronizzazione e la mutua esclusione. Per tale fine, tali linguaggi forniscono-

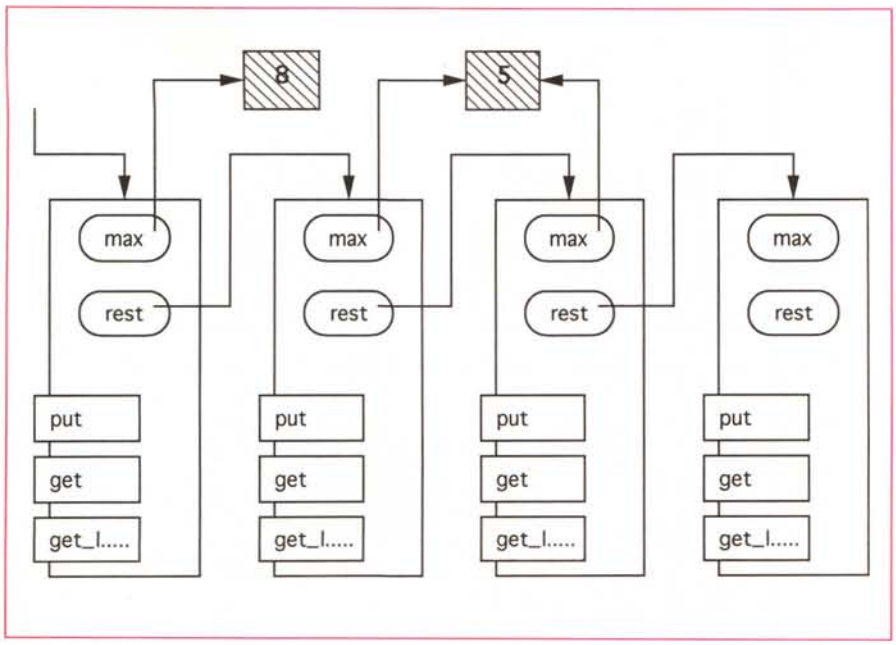

Figura 3 - Coda di priorità che contiene il numero 8 e due volte il numero 5, come vedete con riferimento allo stesso oggetto.

no qualche classe precostituita come per esempio i semafori. Il programmatore comunque, deve ricordare l'esistenza e l'uso corretto di tali strutture.

Approcci più promettenti possono essere ottenuti rilasciando le altre due restrizioni. Omettendo il vincolo numero 2), il mittente di un messaggio può continuare la propria attività senza dover attendere il risultato; viene cioè realizzata una comunicazione asincrona. In tal modo il mittente può continuare l'esecuzione in parallelo con il destinatario del messaggio. In tale schema il parallelismo è proporzionale al numero di messaggi spediti.

In POOL2 tuttavia viene adottato un altro approccio che deriva dal rilascio della condizione 3). Ogni oggetto ha un «body», un processo indipendente locale, che è attivato non appena l'oggetto è creato ed è eseguito in parallelo con i body di altri oggetti creati nel sistema. In un body, arbitrariamente localizzata può trovarsi un'istruzione che indica che l'oggetto vuole mandare o ricevere un messaggio. È chiaro che il parallelismo aumenta con il numero di oggetti creati nel sistema.

Il meccanismo fondamentale di comunicazione in POOL2 è lo «scambio di messaggi sincrono»; il mittente esegue un'istruzione che ha la forma

```
destinatario ! metodo (arg1, arg2, ..., argn)
```

Come si vede è esplicitamente indicato il destinatario, il nome del metodo ed un numero di argomenti che devono essere utilizzati dal metodo.

Il destinatario, per parte sua, esegue un'istruzione di risposta, che ha la forma seguente:

```
ANSWER (metodo1, metodo2, ..., metodon)
```

Essa indica che deve essere ricevuto un messaggio e che il nome del metodo dovrebbe appartenere alla lista dei parametri. La comunicazione è implementata come un rendezvous: il mittente e il destinatario si sincronizzano (il primo che vuole comunicare attende fino a che l'altro non è disponibile), e i parametri sono passati al metodo del destinatario che è allora eseguito. Non appena il metodo ritorna un risultato, tale risultato è passato indietro al mittente del messaggio. Il rendezvous quindi termina ed entrambi gli oggetti continuano la loro esecuzione in completa indipendenza. C'è da notare che la restituzione del risultato non avviene necessariamente alla fine del metodo: è possibile anzi che il metodo continui ben oltre la fine del rendezvous. È possibile così minimizzare il tempo in cui il mittente resta bloccato in attesa del risultato, solo di quanto strettamente necessario. In ogni caso l'istruzione ANSWER termina soltanto quando il metodo invocato è terminato.

POOL2 offre anche un meccanismo di comunicazione asincrona. In questo caso, nessun risultato è ritornato e il mittente continua l'esecuzione immediatamente dopo aver inviato il messaggio al destinatario. Questo meccanismo è derivato da quello sincrono: infatti basta predisporre per ciascun messaggio asincrono un oggetto buffer che riceva dal mittente il messaggio in modo sincrono e lo rispedisca al destinatario in modo ancora sincrono.

I meccanismi disponibili in POOL2 per ottenere il parallelismo offrono un elevato grado di flessibilità nella programmazione mentre permettono la costruzione di programmi semplici ed efficaci. Solo poche classi infatti richiedono body complessi, la maggior parte usa il body di default che risponde a tutti i messaggi in arrivo in sequenza. Anche con questi oggetti si può ottenere un significativo grado di parallelismo lasciando che i metodi ritornino loro risultati non appena possibile. La mutua esclusione nell'accedere ai dati interni ad un oggetto è garantita dall'incapsulamento degli stessi e che i metodi sono eseguiti in ordine strettamente sequenziale, in tale modo è evitata l'interferenza distruttiva di processi agenti sugli stessi dati.

Un semplice programma d'esempio

Per evitare che tutti i concetti che abbiamo finora espresso rimangano soltanto delle idee astratte, vediamo come si può realizzare un semplice programma in POOL2. Ci proponiamo di implementare una coda di priorità di interi, una struttura in cui possiamo inserire dei dati in qualsiasi ordine e che invece restituisce al momento dell'estrazione sempre il maggiore di quelli contenuti nella coda. Nelle figure che accompagnano quest'articolo troverete i vari moduli del codice della coda di priorità; in figura 1 abbiamo la Specification unit, nella quale è descritta la classe PQ e la sua interfaccia verso il mondo esterno; vale a dire tutto ciò che serve per utilizzare la classe nei nostri programmi.

Come si vede abbiamo una classe PQ con una routine new per creare una coda di prirorità vuota e i metodi put e get per inserire ed estarre gli interi. Il risultato del metodo put è in effetti superfluo però, essendo il metodo sincrono necessita di un risultato di ritorno perciò viene ritornato un riferimento dell'oggetto ricevente il messaggio.

In definitiva per creare una nuova istanza della classe PQ, basta un'istruzione del tipo:

```
q:= PQ.new()
```

mentre l'inserimento dell'elemento i nella coda q avviene in questo modo:

q ! put(i)

e l'estrazione di j attraverso:

j:= q ! get().

Consideriamo ora la implementation unit in figura 2, come vedete dà l'implementazione dettagliata della classe PQ. In figura 3 invece viene mostrato un certo numero di istanze della classe PQ. Vediamo che ciascuna istanza memorizza un solo intero; è importante notare che tutti i metodi di questa classe ritornano il loro risultato alla prima istruzione, in questo modo il mittente può continuare la sua attività mentre il metodo sta ancora processando il messaggio. In questo esempio è proprio questa la sorgente del parallelismo: mentre un intero in input viene propagato attraverso l'intera coda, il prossimo può già venire immesso, mentre vengono servite nel contempo le richieste in output.

Il metodo get_largest_or_NIL non appare nella specification unit e quindi non può essere utilizzato da altre unità. In questa unit necessitiamo di tale metodo perché in caso di coda vuota il metodo get è bloccante anche rispetto ad un eventuale messaggio put. Utilizzando invece get_largest_or_NIL, non blocchiamo la coda se vuota restituendo il valore speciale NIL, che fa riferimento ad un oggetto che non esiste.

Il body della classe PQ, molto semplice, controlla che se la coda non è vuota sia il metodo get a rispondere al messaggio. Il loop come si vede non termina esplicitamente ma con una garbage collection si può eliminare ogni oggetto a cui nessun altro oggetto faccia ulteriore riferimento.

È interessante notare che l'esempio mostra alcune espressioni che utilizzano diversi operatori spesso in modo abbreviato. Per esempio, l'espressione

max >= n

è un altro modo di scrivere

max ! greater_or_equal(n).

L'operatore == è un'eccezione, nel senso che rappresenta una chiamata a routine invece di una spedizione di un messaggio; l'espressione

max == NIL

equivale a

Int.id(max,NIL).

In POOL2 ogni classe ha associata automaticamente una routine id che, senza mandare un messaggio, controlla se i due argomenti si riferiscono allo stesso oggetto.

La unit Prio_Queue che abbiamo illustrato finora può essere adoperata per diversi scopi: per esempio può ordinare una sequenza di interi, come mostrato nella unit di figura 4.

Vediamo che l'esecuzione di un programma POOL2 è iniziato non appena l'identificatore globale root viene dichiarato e, con lo scopo di inizializzarlo, viene creato un nuovo elemento della classe Sorter. L'oggetto Sorter allora crea altri oggetti, nel nostro caso una coda di priorità, e avvia l'intero sistema.

```
IMPL UNIT Sorting

USE File_IO, Prio_Queue

GLOBAL root := Sorter.new()

CLASS Sorter
%%Un'istanza di questa classe leggerà interi dal canale standard di
%%input fino a che non ne incontra uno negativo. Allora stamperà
%%quelli positivi in ordine decrescente

VAR   pq := PQ.new ()
      n : Int := standard_in ! read_Int ()

BODY
      WHILE n >= 0
      DO pq ! put (n);
           n := standard_in ! read_Int ()
      OD;
      DO          %%Fino a deadlock
         standard_out ! write_Int (pq ! get (), 0) ! new_line ()
      OD
YDOB
END Sorter
```

Figura 4 - Implementation Unit di un programma che ordina una sequenza di interi, esemplare per chiarezza e semplicità.

Notate che il programma termina in deadlock quando l'oggetto Sorter prova ad estrarre un intero dalla coda ormai vuota. Sebbene appaia molto strano, questo è un modo abbastanza comune di terminare un programma POOL in quanto il sistema run-time si fa carico di individuare tali situazioni di blocco e di rimuovere tutti gli oggetti. Per evitarlo basterebbe inserire un simbolo terminatore di sequenza, per esempio il minimo intero negativo, e terminare non appena esso affiora dalla coda.

Per quanto riguarda le prestazioni teoriche quest'algoritmo ordina una sequenza di N interi in tempo lineare, rispetto al miglior algoritmo seriale che offre performance di O(NlogN).

Conclusioni

Termina qui questo veloce excursus sul linguaggio POOL2, come nostra abitudine utilizziamo le pagine di questa rubrica per illustrare gli aspetti un po' più teorici dell'informatica. Niente di meglio perciò, per gettare uno sguardo oltre la siepe, di un linguaggio che sposa due cavalli di battaglia della teoria dell'informazione alle soglie del 2000: parallel processing e object oriented. Nei prossimi appuntamenti, cercheremo di seguire questa strada, proponendo ai lettori più affezionati e animati da una sana curiosità scientifica, i più interessanti risultati delle ricerche in campo informatico.

MC

Bibliografia

P.H.M. America, J.J.M.M. Rutten, ***«A Parallel Object-Oriented Language: Design and Semantic Foundations»*** *in* ***«Languages for Parallel Architectures»*** *a cura di J.W. de Bakker, Wiley, 1989.*

C. Giustozzi, S. Polini, ***«OOP Object Oriented Programming — La programmazione degli anni '90»,*** *Technimedia, 1991.*