Distribuiti Shared Memory

Orca-Linda

Gullì Daniel

Riferimenti

- Niholas Carriero, David Gelertner “How to Write Parallel Programs: A guide to the Perplex” from ACM Computing Surveys, vol. 21, No. 3, Sept. 1989, pp. 323-357, Copyright 1989, Association for Computing Machinery, Inc.
- ORCA: A LANGUAGE FOR PARALLEL

PROGRAMMING OF DISTRIBUTED SYSTEMS

Henri E. Bal M. Frans Kaashoek Andrew S. Tanenbaum Dept. of Mathematics and Computer Science Vrije Universiteit Amsterdam, The Netherlands

Introduzione

- I computer paralleli rappresentano una grande opportunità per quanto riguarda lo *sviluppo di sistemi ad alte prestazioni* e per *risolvere una vasta gamma di problemi* in diverse aree d'interesse.
- I computer paralleli sono composti da diversi elementi computazionali connessi tra loro :
  - per mezzo di una memoria condivisa (Multi Processor)
  - per mezzo di una rete d'interconnessione (Multi Computer)

Introduzione

- Il parallelismo originariamente è stato studiato da un ramo dei sistemi operativi. La prima idea era quella di fornire delle primitive per la sincronizzazione dei processi concorrenti.
- Queste idee sono state introdotte da Dijkstra, Brinch Hansen e Hoare, che proposero dei meccanismi primitivi come: ***semafori, regione critiche e monitor.***

Introduzione

- I tipici problema nella programmazione parallela sono:
  - la sincronizzazione
  - la creazione di processi
  - il comunication handling
  - il deadlock
  - la terminazione dei processi
- Questi problemi sorgono principalmente perché, quando un programma concorrente è in esecuzione, ci sono molti flussi di controlli attraverso i programmi, uno per ogni processo.

Introduzione

- L'unica classificazione di una certa utilità è quella dovuta a Flynn.
- La classificazione di Flynn si basa su due concetti chiave:

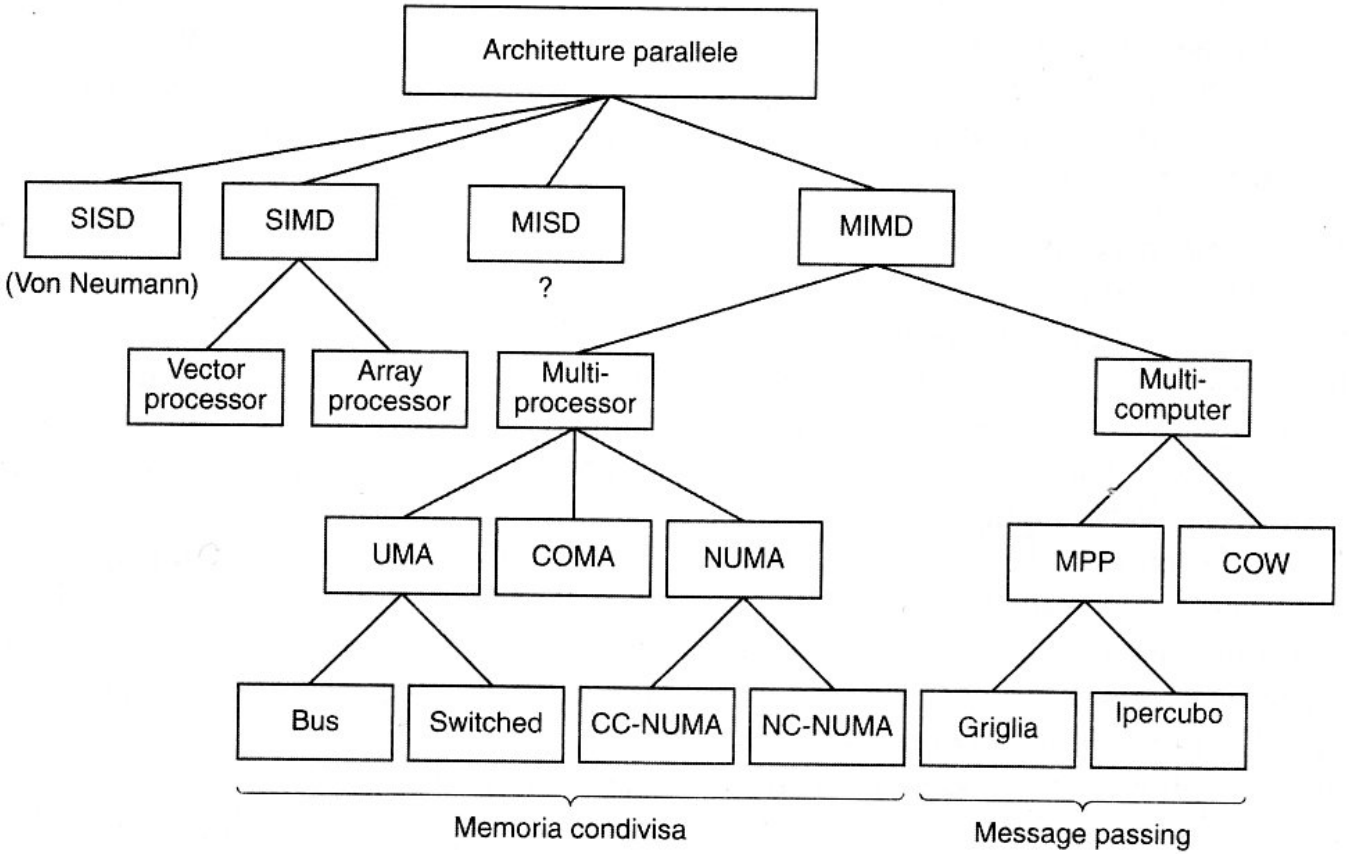
  -sequenza d'istruzione:

  Una sequenza d'istruzioni è in relazione al program counter. Un sistema con n CPU ha n program counter, dato che ha n sequenze d'istruzioni da calcolare.

  -sequenze di dati:

  Una sequenza di dati consiste di un insieme di operandi.

Introduzione



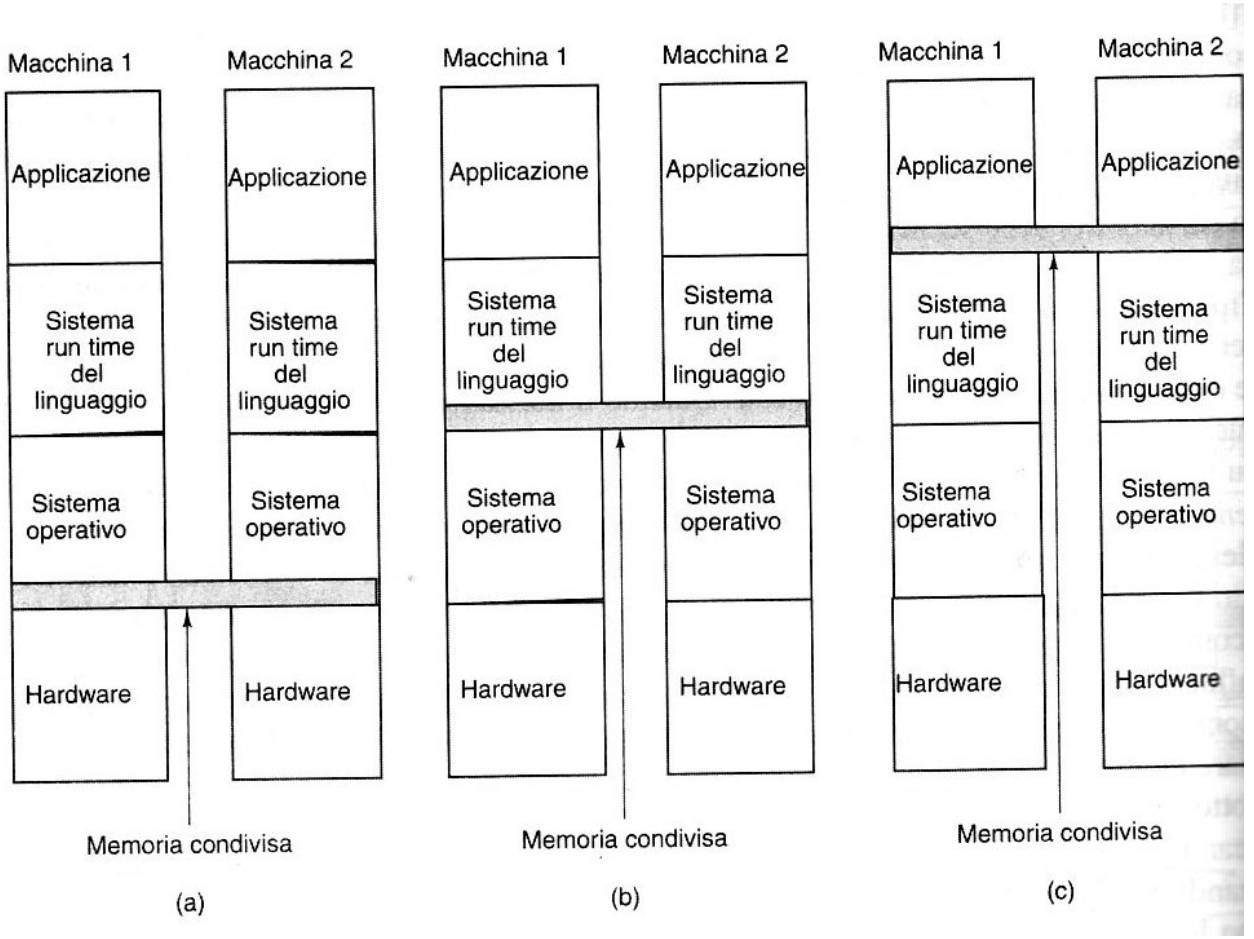
Introduzione

- Tra il Multi Computer e il Multi Processor conviene realizzare il primo poiché i computer sono più semplici e meno costosi da realizzare anche se i Multi Processor sono più facili da programmare. Questa osservazione ha condotto a produrre grandi sforzi per costruire sistemi ibridi che siano relativamente facili da programmare.

Introduzione

- I moderni sistemi informatici non sono monolitici, ma sono costruite a strati, ciò implica che la memoria condivisa può essere implementata a qualsiasi livello strutturale:
  - livello hardware
  - livello del sistema operativo
  - livello applicativo

Introduzione



Introduzione

- In tale approccio il linguaggio di programmazione fornirà una memoria condivisa che poi sarà implementata dal compilatore e dal sistema a runtime.
- Ciò è implementato nei sistemi:
  - Orca
  - Linda

Orca

- Orca è stato realizzato in modo da essere allo stesso tempo semplice, espressivo ed efficiente con una semantica molto chiara.
- I dati condivisi in Orca sono acceduti attraverso delle operazioni di alto livello definite dall'utente.
- Orca è un linguaggi procedurale e fortemente tipato. Le sue dichiarazioni sequenziali e le sue espressioni sono abbastanza convenzionali e sono molto simili a quelli di MODULA-2.
- *Un importante vantaggio di Orca è quello che il suo linguaggio è stato mantenuto il più semplice possibile*.

Orca

- L’idea chiave di Orca è quella di accedere alle strutture dati condivise con delle operazioni di alto livello quindi i dati sono incapsulati e tali modelli d’incapsulamento vengono chiamati oggetti.
- Gli oggetti in Orca hanno la stessa definizione degli oggetti di Java o dei package di ADA.

Orca

- Il parallelismo in Orca è esplicito, perché i compilatori non sono in grado di generare il parallelismo automaticamente.
- Inizialmente un programma Orca è costituito da un unico processo, ma nuovi processi possono essere creati esplicitamente attraverso l'uso dell'istruzione fork:

  - fork name (actual-parameters)[**on** (cpu-number)]
- Questa istruzione crea un nuovo processo che è un figlio del precedente.

Orca

- Se il padre passa una struttura dati come shared al figlio, allora la struttura dati sarà condivisa tra il padre e il figlio.
- Se un processo figlio è dichiarato come:

  process child (Id:integre; X:shared AnObjectType));

  begin ……... end;

  un nuovo processo figlio potrà essere creato come:

  MyObj: AnObjectType; *#dichiara un oggetto*

  #crea un nuovo processo figlio passandogli la cosante 12 #come valore del parametro e l'oggetto MyObj come #parametro condiviso.

  fork child (12,MyObj)
- Questo figlio può passare questo oggetto condiviso ai suoi figli, e così via, tutti gli oggetti che sono condivisi dai processi sono protetti da una variabile di lock.

Orca

- Un oggetto dati condiviso è una variabile di un tipo astratto di dato. Un tipo astratto di dato definito in Orca è costituito da due parti: una specification part un implementation part.
- La specification part definisce le operazioni applicabili agli oggetti dello stesso tipo. La specification part di un tipo di oggetto che incapsula un valore di un intero:

object specification IntObject;
operation Value(): integer; #ritorna il valore corrente
operation Assign(val: integer); #assegna un nuovo valore
operation Add(val: integer); #somma val al valore #corrente
operation Min(val: integer); #imposta il valore dell'ogget al #minimo tra il valore corrente e val
end;

Orca

- Implementation part contiene i dati usati per rappresentare gli oggetti di quel tipo, il codice d'inizializzazione, i dati di una nuova istanza di quel tipo e il codice d'implementazione delle operazioni. Parte dell'implementazione del tipo IntObject è qui sotto descritta:

Orca

```
object specification IntObject;
    x:integre; #il dato interno
    operation Value(): integer;
    begin
        return x; #ritorna il valore corrente
    end;
    operation Assign(val: integer);
    begin
        x=val; #assegna un nuovo valore
    end;
    ……..
    begin
        x:=0; #il valore iniziale dell'oggetto è zero
    end;
```

Orca

- Quando un oggetto viene creato, viene allocata la memoria per le variabili locali inoltre viene eseguito il suo codice d'inizializzazione.
- La sintassi per dichiarare un oggetto e applicargli un operazione è illustrata qui sotto:

```
X:IntObject;
tmp:integer;
X$Assign(3); #assegna 3 a X
X$Add(1); #incrementa X
tmp:=X$Value();#legge il corrente valore di X
```

- Orca supporta un singolo meccanismo per i dati astratti.
- La sua caratteristica (oggetto condiviso o non) sarà derivata dall'uso che si farà dell'oggetto, cioè da come sarà passato in un'istruzione di fork.

Orca

- Per gli oggetti che sono condivisi tra processi multipli, il problema è garantire la sincronizzazione.

- Esistono due tipi di sincronizzazione:
  - la mutua esclusione
  - la sincronizzazione condizionata.

Orca

- La mutua esclusione in Orca è realizzata implicitamente, tramite l'uso di operazioni indivisibili. Ogni operazione blocca l'intero oggetto a cui è applicata, svolge il suo compito e rilascia l'oggetto bloccato solo quando ha terminato, il modello garantisce la serializzabilità delle operazioni invocate.
- Orca non supporta operazioni indivisibili su collezioni d'oggetti.

Orca

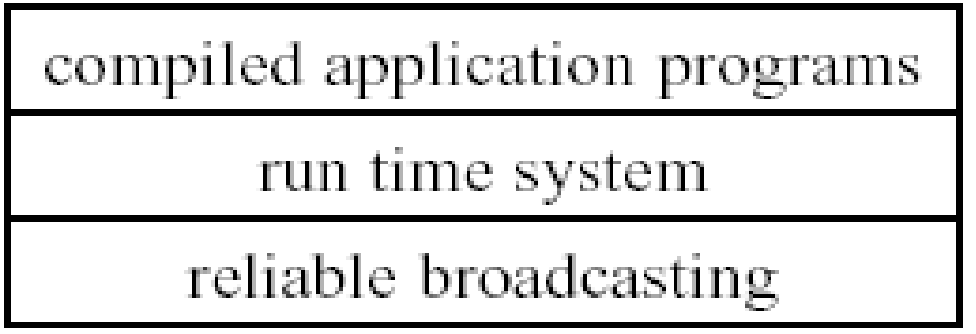
- Nella sincronizzazione condizionata un processo rimane nello stato di wait finché una certa condizione non diviene vera. Le operazioni bloccanti consistono in una o più comandi di guard:

```
operation op (formal-parameters): ResultType;
begin
   guard condition1 do stantements1 od;
   …
   guard conditionn do stantementn od;
end;
```

- Le condizioni sono espressioni booleane chiamate guardie.
- L'operazione inizialmente bloccata rimarrà tale fino a quando almeno una guardia non diventerà true, allora eseguirà il suo stantement.

Orca

- La migliore implementazione che è stata pensata è composta da tre strati software qui sotto riportati:



Orca

- Il primo strato riguarda le applicazioni, che sono scritte in Orca e compilate nel codice macchina dal compilatore Orca. Il codice eseguibile contiene chiamate al sistema run-time di Orca. Lo strato intermedio è il sistema a run-time (RTS), il quale implementa le chiamate primitive dello strato superiore. L'ultimo strato riguarda l'implementazione di una comunicazione affidabile, in modo che RTS non abbia problemi come la perdita di messaggi.

Orca

- Il codice prodotto dal compilatore possiede delle chiamate ad alcune routine di RTS le quali *gestiscono* i processi, i dati condivisi e le strutture dati complesse.
- E' molto importante distinguere tra le operazioni di lettura e scrittura. I compilatori analizzano il codice d'implementazione di ogni operazione e controllano se le operazioni modificano gli oggetti alle quali sono applicate.

Orca

- Se un programma Orca applica un'operazione ad un oggetto ricevuto, il compilatore genera una chiamata a RTS attraverso la primitiva INVOKE. Questa routine è chiamata nel modo seguente:

INVOKE (object, operation-descriptor, parameters ...);

- Il primo argomento identifica l'oggetto al quale l'operazione è applicata, mentre il secondo argomento è il descrittore delle operazioni, mentre i rimanenti parametri di INVOKE sono i parametri dell'operazione.

Orca

- **Per una maggiore efficienza RTS replica gli oggetti in modo da applicare le operazioni alle copie locali se possibile.**
- **RTS usa una replicazione completa degli oggetti, un aggiornamento delle repliche e l'implementazione della mutua esclusione attraverso il protocollo di aggiornamento distribuito.**

  -La replicazione completa è stata scelta perché è semplice e possiede buone prestazioni per molte applicazioni.

  -Aggiornamento delle copie anziché dell'invalidamento per due ragioni:

  1) Molte applicazioni contengono un grande quantità di dati, quindi l'invalidazione di una copia in tali circostanze è dispendioso, poiché in un successivo momento l'oggetto dovrà essere replicato e trasmesso per intero.

  2) In molti casi l'aggiornamento di una copia presenta un tempo di CPU e una larghezza di banda inferiore rispetto all'invalidazione.
- **La presenza di più copie dello stesso dato rappresenta il problema dell'inconsistenza.**

Orca

- Se l'aggiornamento non è un'operazione indivisibile allora differenti processori possono temporaneamente avere differenti valori per lo stesso dato, ciò è inaccettabile.
- RTS risolve tale problema utilizzando un protocollo per l'aggiornamento distribuito, che garantisce a tutti i processi di osservare i cambiamenti degli oggetti condivisi nello stesso ordine. Tale protocollo non utilizza i lock perché richiedono molto tempo e sono complicati, invece utilizza alcune primitive indivisibili della Reliable Broadcast che hanno le seguenti proprietà:
  1. L'affidabilità
  2. Consegna in ordine FIFO

Orca

- RTS usa un Object Manager per ogni processore, che è un processo leggero che si occupa di aggiornare le copie locali di tutti gli oggetti memorizzati nel processore.
- Il processo utente può leggere una copia locale senza l'intervento dell'object manager. Le operazioni di write su oggetti condivisi sono realizzate attraverso l'object manager. Un processo utente che spedisce un operazione di write si sospende fino a quando il messaggio non è stato manipolato dal suo object manager locale.

Orca vs. RPC

- L'aggiornamento delle copie con una RPC è molto complicato con una trasmissione indivisibile, poiché ogni replica deve essere aggiornata in un modo consistente, quindi si deve garantire la consistenza.
- RPC usa due fasi nel suo protocollo di aggiornamento. Prima tute le copie sono bloccate e aggiornate, dopo tutti gli aggiornamenti devono essere inviati gli ack, così inizia la seconda fase, tutte le copie devono essere sbloccate. *Questo protocollo è molto costoso rispetto a quello di Orca*, perché il tempo di aggiornamento dipende dal numero di copie.
- In RPC il costo di comunicazione è maggiore rispetto al protocollo di Orca.

Linda

- Se è vero che per inviare solo quanto è differente, anziché intere pagine (come per i sistemi DSM basati sul paging), migliora le prestazioni, allora probabilmente occorre anche chiedersi se le pagine siano realmente l'unità di misura più adatta alla condivisione della memoria.
- Linda è un sistema i cui processi girano su più macchine e con una *memoria condivisa distribuita e altamente strutturata*, tale memoria può essere acceduta attraverso delle primitive che possono essere aggiunte a linguaggi esistenti, quali C o FORTRAN, per trasformarli in linguaggi paralleli (che vengono ribattezzati con il nome di C-Linda e Fortran-Linda).

Linda

- Il modello della memoria in Linda si chiama tupla space (o TS), consiste di una collezione logica di tuple.
- Esistono due tipi di tuple:

  -Process tuples vengono valutate in modo attivo

  -Data tuple che sono passive “semplici dati”.
- I process tuples (sono eseguite tutte simultaneamente) scambiano dati li generano, li leggono, li consumano. La loro esecuzione finisce dentro un data tuple, che è indistinguibile dalle altre data tuple.

Linda

- Ci sono quattro operazioni per TS: ***out, in ,rd e eval*** e due varianti ***inp e rdp***.
- ***Out(t)*** causa che la tupla t sia inserita in TS, le tuple possono essere lette e tolte da TS facendo uso della primitiva ***in***, mentre ***rd*** è analoga a in con la differenza che non rimuove la tupla da TS, mentre ***eval*** forza la valutazione dei parametri in parallelo e l'inserimento della tupla risultante in TS. Usando l'operazione eval vengono creati in Linda i processi paralleli.
- ***Inp*** e ***rdp*** tentano di localizzare le tuple richieste e ritornano 0 se fallisce tale richiesta, altrimenti 1 e compieno il compito assegnato.

Linda

out (“a string”, 15.01, 17, “another string”)

out(0,1)

- tali tuple vengono generate e inserite in TS.

in (“a string”, ?f, ?i, “another string”)

- Tale operazione causa la ricerca di una tupla con quattro elementi il “a string” il secondo e il terzo possono essere qualsiasi mentre il quarto deve essere “another string” dopo aver trovato tale tupla gli elementi centrali vengono sostituiti da f e i, ma la tupla che ne deriva viene letta ma anche tolta da TS.

rd (“a string”, ?f, ?i, “another string”)

- compie le stesse operazioni di in(…) ma con la differenza che la tupla che ne deriva rimane in TS.

eval (“e”, 7, exp(7))

- crea tre elementi in una “live-tupla” che valuta la stringa “e” dell’intero 7 e chiama la funzione exp che eleva e alla settima potenza.

Linda

- Possiamo dividere convenzionalmente le strutture dati in un ambiente distribuito in tre categorie:

  -strutture i cui elementi sono identici o indistinguibili

  -strutture in cui gli elementi sono distinti per nome

  -strutture in cui gli elementi sono distinti per posizione:

  1. strutture in cui gli elementi hanno un accesso casuale
  2. strutture in cui gli elementi hanno un accesso ordinato

Linda

- In Linda un semaforo contatore è una collezione di elementi identici. Eseguire V su un semaforo sem:

out("sem")

eseguire a P

in("sem")

- inizializzare un semaforo al valore n si deve eseguire n volte l'operazione out("sem").
- I semafori non sono molto usati nella programmazione parallela.
- Un bag è una struttura dati che definisce due operazioni: aggiungi un elemento ed elimina un elemento.
- Tali bag non sono importanti nei programmi sequenziali ma estremamente importanti in quella parallela. Il semplice tipo di programma replicated-work dipende da un bag di task. Task sono aggiunti al bag usando:

out ("task", TaskDescription)

e tolti usando:

in ("task", ?NewTask)

Linda

Strutture in cui gli elementi sono distinti per nome

- Le applicazioni parallele spesso richiedono l'accesso a collezioni di elementi costituiti da elementi distinti per nome. Queste collezioni sono simili ai record di Pascal o alle struct in C. Possiamo immagazzinare ogni elemento in una tupla dalla forma:

(***name,value)***

- per leggerli si può usare

rd(name,?val)

- mentre per aggiornarli

in(name,?old)

out(name,new)

Linda

- Alcune applicazioni parallele fanno affidamento su delle "barriere di sincronizzazione": ogni processo appartenente a un gruppo dovrà aspettare sulla barriera fino a quando tutti gli altri processi non arriveranno in tale punto; allora tutti potranno procedere. Se il gruppo contiene n processi, possiamo creare una barriera chiamata barrier-37 così:

out("barrier-37",n)

- Quando un processo raggiunge la barriera di sincronizzazione allora esegue le seguenti istruzioni:

in("barrier-37",?val)

out("barrier-37",val-1)

rd("barrier-37",0)

- allora ogni processo decrementa il valore e aspetta che tale numero sia zero, per poi proseguire.

Linda

Strutture accedute tramite posizione

- Gli array distribuiti sono il centro delle applicazioni parallele in molti contesti, tali strutture possono essere realizzati tramite delle tuple che hanno la seguente forma:

(Array, indexfield, value)

- Quindi, la seguente etichetta

("V", 14, 123.5)

- individua il quattordicesimo elemento del vettore V, mentre la seguente etichetta:

("A", 12, 18, 5, 123.5)

- individua un elemento in un array con tre dimensioni.

Linda

- Esiste una struttura di dati ordinata che rappresenta il centro di molte applicazioni, tali strutture prendono il nome di Stream. Ci sono due tipi di Stream chiamati:
  - -in-Stream
  - -rd-Stream
- Gli stream sono una sequenza ordinata di elementi i quali sono inseriti da alcuni processi.
- In-Stream: ogni processo può rimuovere la testa del stream in ogni momento. Se più processi cercano di prelevare la testa dello stream tale operazione viene serializzata a run-time.
- Rd-stream: ogni processo può leggere dallo stream simultaneamente, ogni reading process legge il primo elemento dallo stream, dopo il secondo e poi così via.
- Un processo che cerca di rimuovere un elemento da uno stream vuoto rimane bloccato fino a quando lo stream non conterrà un elemento.

Linda

- Gli stream sono facili da realizzare in Linda, essi sono costituiti da una serie numerata di tuple:

("strm", 1, val1)

("strm", 2, val2)

- l'indice dell'ultimo elemento è preso in tail:

("strm", "tail", 14)

- Per inserire un nuovo elemento (NewElt) nello stream (strm) eseguire le seguenti istruzioni:

in ("strm", "tail", ?index)

/*consulta il puntatore tail */

out ("strm", "tail", index+1)

out ("strm", index, NewElt)

/*aggiungi un elemento */

Linda

- Per rimuovere la testa dello stream con un in-stream:

```
in (“strm”, “head”, ?index)
  /*consulta il puntatore head */
out (“strm”, “head”, index+1)
in (“strm”, index, Elt)
  /*rimuovi l’elemento */
```

- Nota che quando lo stream è vuoto si blocca. Mentre rd-stream viene realizzato così:

```
index=1;
<loop>{
  rd in (“strm”, index++, ?Elt);
.....
}
```

- Quando in un rd-stream un processo ha consumato lo stream, noi disponiamo ancora della testa dello stream che può essere acceduta e consumato da un nuovo processo.

Conclusioni

- Linda è una dei primi linguaggi che riconosce gli *svantaggi di un processo centrale per la gestione* della protezione dei dati condivisi.
- *Tuples diversi possono essere accedute indipendentemente da ogni processo*, così i processi possono manipolare tuple diverse delle stesse strutture dati simultaneamente, cioè letture multiple possono essere effettuate simultaneamente.
- Linda usa il modello Tuple Space per implementare le strutture dati distribuite. In generale *le strutture dati sono costituite da tuple multiple.*
- Il supporto dati di Linda per *le strutture dati distribuite possono essere accedute simultaneamente da processi multipli*, contrariamente a ciò che accade per i linguaggi object-based dove l'accesso alle strutture dati condivise è di tipo seriale.

Conclusioni

- In Tuple Space esistono un numero fisso di operazioni incorporate che sono eseguite indivisibilmente, ma *il supporto fornito per costruire operazioni indivisibile più complesse è troppo a basso livello*, mentre in Orca, i programmatori possono definire d'altra parte operazioni della complessità arbitraria su strutture dati condivise; tutte queste operazioni sono eseguite indivisibilmente, così che la mutua esclusione è realizzata dal sistema a run-time.
- Questo vuole dire che il lavoro dell'implementazione è affidato al compilatore e al sistema in fase d'esecuzione che devono individuare le operazioni che possono essere eseguite in parallelo e quelle che devono essere eseguite sequenzialmente.

Grazie

Fine

Gentile

Attenzione