Paradigmi di programmazione parallela

Reti di Calcolatori A.A. 2023/24

Prof.ssa Chiara Petrioli - Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Michele Mastrogiovanni - Dipartimento di Ingegneria Informatica

Ringraziamenti: - **Emanuele Giona** Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Meccanismi di comunicazione e sincronizzazione... Concretamente?

Applicazioni basate su multi-processing e multi-threading:

- Meccanismi di comunicazione
  - Dati locali ad un processo / thread devono essere resi disponibili a terzi
- Meccanismi di sincronizzazione
  - Accesso a risorse condivise, oppure dati parziali prodotti da un processo / thread devono essere utilizzati da terzi

L'implementazione di una specifica applicazione determina quali meccanismi usare e come sfruttarli per ottenere il risultato desiderato:

- Caratteristiche specifiche dello strumento
  Ad es. Comunicazione non-bloccante e bidirezionale; collaborazione su un oggetto presente su memoria condivisa; *ecc*.
- Struttura del programma parallelo
  Ad es. Creazione di thread al momento del bisogno oppure predisposizione di un numero di worker a cui inviare task; *ecc*.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Paradigmi di programmazione parallela

Come in altri ambiti dello sviluppo software, un'applicazione può essere ricondotta a casi generali di cui una soluzione efficiente è nota:

- Numero dei task
- Tipologia dei task
- Flusso dell'esecuzione
- Garanzie da mantenere durante l'esecuzione
- …

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: produttore-consumatore

In questo pattern ci sono due tipologie di task:

1. Produttore
   Il risultato di questo task è la generazione di dati intermedi
2. Consumatore
   Questo task usa i dati intermedi come proprio input

Una caratteristica fondamentale di questo pattern è che i dati vengono consumati.
Una volta prodotti, rimangono accessibili da processi / thread di tipo consumatore fino al loro primo accesso.
Dopo il primo accesso ai dati intermedi, questi non possono più essere utilizzati da altri task consumatori.

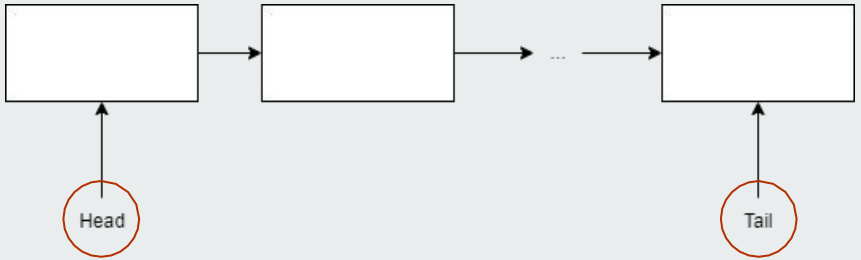
Questo paradigma viene comunemente implementato sfruttando una coda in cui un task produttore inserisce i dati intermedi, e da cui un task consumatore invece li preleva.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: produttore-consumatore

In un primo approccio:

- Assunzione di coda infinita
- Unico produttore
- Unico consumatore



Una soluzione potrebbe essere:

- Linked list
- 1 mutex

Ogni operazione che modifica i puntatori all'inizio e fine della coda devono essere protetti da un mutex

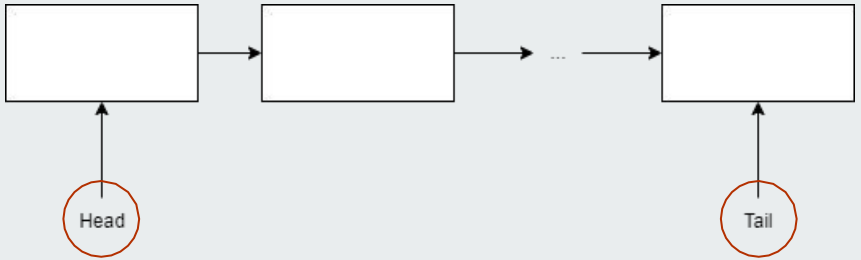
È una soluzione sufficientemente sicura?

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: produttore-consumatore

In un primo approccio:

- Assunzione di coda infinita
- Unico produttore
- Unico consumatore



Ogni operazione che modifica i puntatori all'inizio e fine della coda devono essere protetti da un mutex

Una soluzione potrebbe essere:

- Linked list
- 1 mutex

È una soluzione sufficientemente sicura?

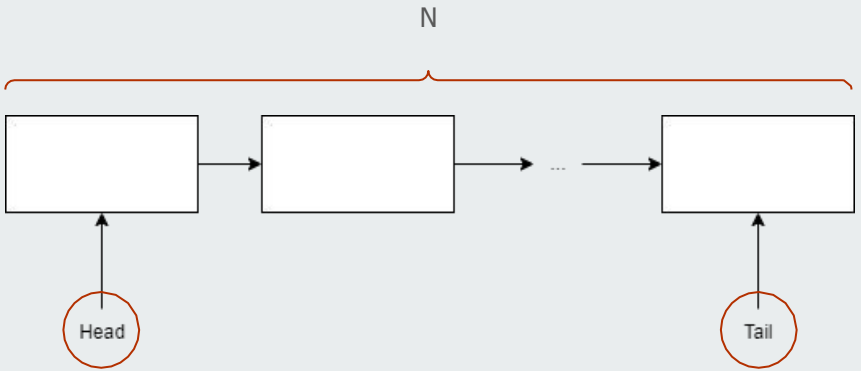
✓ Programmazione parallela: sì

× Un produttore malevolo o malfunzionante potrebbe inserire dati in maniera incontrollata, esaurendo le risorse

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: produttore-consumatore

- Coda di dimensione massima N
- Unico produttore
- Unico consumatore



Una soluzione potrebbe essere:

- Linked list
- Semaforo
  - Inizializzato a 0
  - Ogni inserimento equivale a +1
  - Ogni prelievo equivale a -1

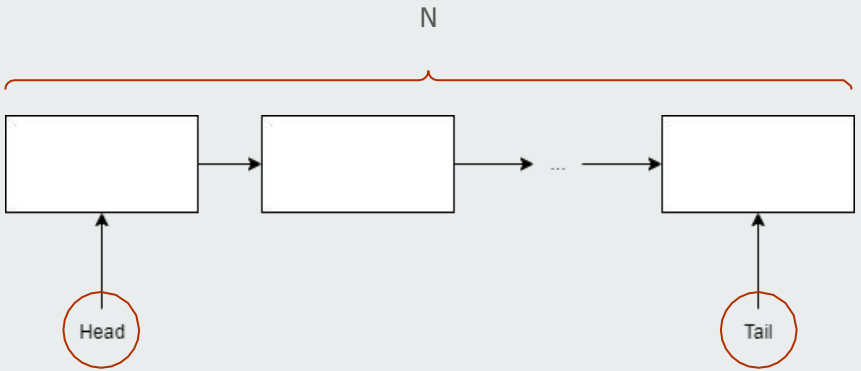
Il semaforo garantisce che il numero di elementi sia al più N e che i puntatori ad inizio e fine non vengano modificati contemporaneamente

- Ogni produttore si bloccherà se la coda ha raggiunto il limite N
- Ogni consumatore si bloccherà se la coda è vuota

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: produttore-consumatore

- Coda di dimensione massima N
- X produttori
- Y consumatori



Oltre al semaforo per la dimensione della coda, sono necessari 2 mutex per garantire un accesso atomico ai puntatori di inizio e fine coda

Una soluzione potrebbe essere:

- Linked list
- Semaforo + 2 mutex
  - Inizializzato a 0
  - Ogni inserimento equivale a +1
  - Ogni prelievo equivale a -1
- Ogni produttore si bloccherà se la coda ha raggiunto il limite N
  - Aggiuntivamente deve acquisire un lock per modificare il puntatore alla fine della coda
- Ogni consumatore si bloccherà se la coda è vuota
  - Aggiuntivamente deve acquisire un lock per modificare il puntatore all'inizio della coda

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: lettori-scrittori

In questo pattern ci sono due tipi di task:

1. Lettori
   I vari lettori si contendono l'accesso ad una risorsa da cui leggono dati, senza effettuare alcuna modifica sull'originale
2. Scrittori
   I vari scrittori invece effettuano modifiche sulla risorsa condivisa

I task lettori, non modificando la risorsa condivisa, possono procedere con le esecuzioni senza creare una sezione critica, mentre per ogni scrittore è necessario un accesso esclusivo.

Tuttavia i lettori bloccheranno l'esecuzione di uno scrittore: affinché sia possibile effettuare una modifica sulla risorsa, ogni lettore deve aver rilasciato la risorsa.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: lettori-scrittori

Una soluzione potrebbe essere:

- Contatore dei lettori
  - Il primo lettore "occupa" la risorsa: gli scrittori devono attendere
  - L'ultimo lettore "libera" la risorsa: gli scrittori possono proseguire
- 1 mutex + 1 semaforo binario (inizializzato a 1)
- Scrittori:
  - Attesa sul semaforo (-1) direttamente
  - *Operazioni sulla risorsa*
  - Segnale sul semaforo (+1) direttamente
- Lettori:
  - Acquisizione lock
  - Aggiornamento contatore +1 → se il contatore è 1: attesa sul semaforo (-1)
  - Rilascio lock
  - *Operazioni sulla risorsa*
  - Acquisizione lock
  - Aggiornamento contatore -1 → se il contatore è 0: segnale sul semaforo (+1)
  - Rilascio lock

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: lettori-scrittori

Una soluzione potrebbe essere:

- Contatore dei lettori
  - Il primo lettore "occupa" la risorsa: gli scrittori devono attendere
  - L'ultimo lettore "libera" la risorsa: gli scrittori possono proseguire
- 1 mutex + 1 semaforo binario (inizializzato a 1)
- Scrittori:
  - Attesa sul semaforo (-1) direttamente
  - *Operazioni sulla risorsa*
  - Segnale sul semaforo (+1) direttamente
- Lettori:
  - Acquisizione lock
  - Aggiornamento contatore +1 → se il contatore è 1: attesa sul semaforo (-1)
  - Rilascio lock
  - *Operazioni sulla risorsa*
  - Acquisizione lock
  - Aggiornamento contatore -1 → se il contatore è 0: segnale sul semaforo (+1)
  - Rilascio lock

Nuovi task lettori possono arrivare in ogni momento e rendere impossibile per uno scrittore di procedere con le proprie operazioni

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Pattern: lettori-scrittori

Per evitare starvation ai task scrittori:

- Contatore dei lettori
- 1 mutex + 2 semafori binari (inizializzati a 1)
  - Il nuovo semaforo funziona da selettore tra lettori e scrittori
- Scrittori:
  - Attesa sul semaforo selettore (-1)
  - Attesa sul semaforo (-1) direttamente
  - Segnale sul semaforo selettore (+1)
  - Segnale sul semaforo (+1) direttamente
- Lettori:
  - Attesa sul semaforo selettore (-1)
  - Segnale sul semaforo selettore (+1)
  - *Il resto della soluzione*

Traccia la presenza di uno scrittore in attesa

Operazioni sulla risorsa

Sblocca un qualunque task in attesa: garantisce che almeno uno scrittore possa alternarsi ai vari lettori

Blocca un nuovo lettore se c'è uno scrittore in attesa

In alcune applicazioni potrebbe anche essere opportuno dare priorità agli scrittori rispetto ai lettori.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sviluppo di soluzioni parallele

Lo sviluppo di soluzioni parallele può essere suddiviso in tre livelli:

1. Algoritmico
2. Implementativo
3. Esecutivo

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sviluppo di soluzioni parallele

Lo sviluppo di soluzioni parallele può essere suddiviso in tre livelli:

1. Algoritmico
   Livello più alto: identificare il tipo dei task all'interno dell'applicazione
   a. Task parallel → sotto-problemi indipendenti
   b. Data parallel → stessa operazione applicata a sotto-insiemi dei dati complessivi
2. Implementativo
3. Esecutivo

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sviluppo di soluzioni parallele

Lo sviluppo di soluzioni parallele può essere suddiviso in tre livelli:

1. Algoritmico
2. Implementativo

   Livello intermedio: identificare le tecniche di implementazione dell'algoritmo

   a. Fork/Join → ogni task parallelo viene risolto da thread creati dinamicamente al bisogno; il programma non procede finché questi thread non terminano. Molto comune nel caso di algoritmi data-parallel.
   b. Map/Reduce → simile a Fork/Join, ma concettualmente le operazioni di tipo map producono nuovi dati, mentre le operazioni di tipo reduce aggregano e possono avere dei side effect; alla fine, un unico risultato viene prodotto.
   c. Manager/Worker → thread lavoratori ricevono dei task da un unico gestore e comunicano solo con questo; il gestore cerca anche di bilanciare il lavoro tra i vari lavoratori. Molto comune per algoritmi task-parallel.

3. Esecutivo

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sviluppo di soluzioni parallele

Lo sviluppo di soluzioni parallele può essere suddiviso in tre livelli:

1. Algoritmico
2. Implementativo
3. Esecutivo

   Livello più basso: adattare il modello implementativo alle capacità hardware disponibili

   a. Supporto multi-processing o multi-thread
   b. Thread-pool e coda task → numero fisso di thread che esegue task da una coda
      - ✓ Creazione all'avvio del programma
      - ✓ Riutilizzo degli stessi thread → risparmio di risorse per creazione e distruzione durante l'esecuzione dei task
      - ✓ Consumo risorse più prevedibile → ottimizzazione da parte dello scheduler
      - × Condivisione della coda task → sincronizzazione necessaria
      - × Thread su core CPU diversi possono perdere i benefici della cache
      - × In assenza di tracciamento task / dataset e in caso di errori hardware → perdita progressi parziali

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sviluppo di soluzioni parallele

Considerazioni finali

- Costi di condivisione e sincronizzazione
- Task dell'applicazione noti a priori oppure determinati a runtime
- L'ottenimento di una soluzione necessita che tutti i task siano portati a termine
- Uniformità dei sotto-problemi → natura dei task
- Bilanciamento del lavoro → quantità dati tra processi / thread diversi
- Tipologia delle strutture dati → ristrutturazione può evidenziare parallelismo
- Conoscenza di un dato ordinamento dei task
- Overhead parallelismo *vs* esecuzione sequenziale all'interno di un dato processo / thread

Metriche importanti

- Throughput → unità di risultati finali prodotta al secondo

Fondamentale per calcolare speedup ottenuto dall'implementazione parallela rispetto ad una sequenziale

- Latenza → tempo trascorso dalla produzione del risultato finale dall'inserimento di dati di input

Importante per applicazioni real-time

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Riferimenti

- https://en.wikipedia.org/wiki/Parallel_programming_model
- https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf
- https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ProdCons.html
- https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ReadWrite.html
- https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ParallelDesign.html
- http://algogroup.unimore.it/people/marko/courses/programmazione_parallela/PP1617/04-Design_patterns.pdf

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma