SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Programmazione di Sistemi Embedded e Multicore

Autore:
Simone Lidonnici

8 ottobre 2024

Indice

1

Programmazione parallela

1.1 Perché usare la programmazione parallela

Dal 1986 al 2003 le performance dei microprocessori aumentavano di circa il 50% l'anno, dal 2003 questo aumento è diminuito fino ad arrivare al 4% l'anno. Questa diminuzione è causata dal fatto che l'aumento delle performance dipende dalla densità dei transistor, che diminuendo di grandezza generano più calore e questo li fa diventare inaffidabili. Per questo conviene avere più processori nello stesso circuito rispetto ad averne uno singolo più potente.

1.2 Scrivere programmi paralleli

Per utilizzare al meglio i diversi processori, bisogna volutamente scrivere il programma in modo da usare il parallelismo, in alcuni casi è possibile convertire un programma sequenziale in uno parallelo ma solitamente bisogna scrivere un nuovo algoritmo.
Esempio:
Computare n valori e sommarli tra loro.
Soluzione sequenziale:

```
sum=0;
for(i=0;i<n;i++){
    x=ComputeValue(...);
    sum+=x;
}
```

Soluzione parallela:
Avendo p core ognuno eseguirà $\frac{n}{p}$ somme

```
sum=0;
start=...; // definiti in base al core
end=...;   // che esegue il programma
for(i=start;i<end;i++){
    x=ComputeValue(...);
    sum+=x;
}
```

Dopo che ogni core avrà finito la sua somma, per ottenere la somma totale si può designare un core come **master core**, a cui tutti gli altri core invieranno la propria somma e che eseguirà la somma totale.

Questa soluzione però non è ottimale perchè il master core esegue una ricezione e una somma per ogni altro core ($p-1$ volte), mentre gli altri sono fermi. Per migliorare l'efficienza si sommano a coppie i risultati di ogni core:

- il core 0 somma il risultato con il core 1, il core 2 con il core 3 ecc...
- si ripete con solo i core 0, 2 ecc...
- si continua creando uno schema ad albero binario

Questo secondo metodo è molto più efficiente perchè il numero di ricezioni e somme che esegue il core che ottiene il risultato finale sono $\log_2(p)$.

1.2.1 Tipi di parallelismo

Ci sono due principali tipi di parallelismo:

- **Task parallelism**: Diversi task vengono divisi tra i vari core e ogni core esegue operazioni diverse su tutti i dati (il parallelismo temporale nei circuiti è un tipo di task parallelism)
- **Data parallelism:** Vengono divisi i dati tra i core e ogni core esegue operazioni simili su porzioni di dati diverse (il parallelismo spaziale nei circuiti è un tipo di task parallelism)

Esempio:
Bisogna correggere 300 esami ognuno con 15 domande e ci sono 3 professori disponibili.
In questo caso:

- Data parallelism:
  - Ogni assistente corregge 100 esami
- Task parallelism:
  - Il primo professore corregge le domande 1-5 di tutti gli esami
  - Il secondo professore corregge le domande 6-10 di tutti gli esami
  - Il terzo professore corregge le domande 11-15 di tutti gli esami

1.2.2 Come scrivere programmi paralleli

Se i core possono lavorare in modo indipendente scrivere un programma parallelo è molto simile a scriverne uno sequenziale, in pratica però i core devono comunicare, per diversi motivi:

- **Comunicazione:** un core deve inviare dei dati ad un altro
- **Bilanciamento del lavoro:** bisogna dividere il lavoro in modo equo per far si che un core non sia sovraccaricato rispetto agli altri, sennò tutti i core aspetterebbero il più lento
- **Sincronizzazione:** ogni core lavora a velocità diversa e bisogna controllare che uno non vada troppo avanti rispetto agli altri

Per creare programmi paralleli useremo 4 diverse estensioni di C:

1. **Message-Passing Interface (MPI)**: Libreria
2. **Posix Threads (Pthreads)**: Libreria
3. **OpenMP**: Libreria e Compilatore
4. **CUDA**: Libreria e Compilatore

1.3 Tipi di sistemi paralleli

I sistemi paralleli possono essere divisi in base alla divisione della memoria:

- **Memoria Condivisa:** Tutti i core hanno accesso alla memoria del computer e si coordinano modificando locazioni di memoria condivisa
- **Memoria Distribuita:** Ogni core possiede una propria memoria e per coordinarsi devono mandarsi dei messaggi

Possono essere anche divisi in base al numero di **Control Unit**:

- **Multiple-Instruction Multiple-Data (MIMD):** Ogni core ha la sua CU e può lavorare indipendentemente dagli altri
- **Single-Instruction Multiple-Data (SIMD):** I core condividono una sola CU e devono eseguire tutti le stesse operazioni o stare fermi

Le libreria nominate precedentemente si collocano in una tabella:

	SIMD	MIMD
Memoria Condivisa	CUDA	Pthreads OpenMP CUDA
Memoria Distribuita		MPI

1.3.1 Concorrenti vs Paralleli vs Distribuiti

I sistemi possono essere:

- **Concorrenti:** più task possono essere eseguite in ogni momento, possono essere anche sequenziali
- **Paralleli:** più task cooperano per risolvere un problema comune, i core sono condividono la memoria o sono connessi tramite un network veloce
- **Distribuiti:** un programma che coopera con altri programmi per risolvere un problema comune, i core sono connessi in modo più lento

I sistemi paralleli e distribuiti sono anch'essi concorrenti.

1.4 Architettura di Von Neumann

Per poter scrivere codice efficiente bisogna conoscere l'architettura su cui si sta eseguendo il codice ed ottimizzarlo per essa.
L'architettura di Von Neumann è composta da:

- **Memoria principale:** Insieme di locazioni, ognuna con un indirizzo e del contenuto (dati o istruzioni)
- **CPU/Processore/Core:** Control Unit, che decide le istruzioni da eseguire, e **datapath**, che eseguono le istruzioni. Lo stato di un programma in esecuzione viene salvato nei registri, un registro molto importante è il **Program Counter (PC)** dove viene salvato l'indirizzo della prossima istruzione da eseguire
- **Interconnessioni:** Usate per trasferire dati tra CPU e memoria, tradizionalmente con un bus

Una macchina di Von Neumann esegue un istruzione alla volta, operando su piccole porzioni di dati contenuti nei registri. La CPU può leggere (fetch) dati dalla memoria o scriverci (store), la separazione tra CPU e memoria è conoscuta come **Bottleneck di Von Neumann**, cioè le interconnessioni determinano la velocità con cui i dati vengono trasferiti.

2
MPI

La programmazione parallela tramite **MPI** utilizza un approccio **Single-Program Multiple-Data (SPMD)** in cui si compila un solo programma eseguito da più processi e tramite degli if-else si specifica quale processo deve eseguire quale parte di codice. I processi non hanno memoria condivisa quindi comunicano tramite **messaggi**.
Per poter utilizzare MPI bisogna importare l'header `mpi.h` che permetterà di utilizzare le funzioni MPI, per esempio:

- `MPI_Init`: esegue il setup necessario

```
int MPI_Init(
    int* argc_p,    // puntatore al numero di argomenti
    char*** argv_p  // puntatore agli argomenti in input
);
```

- `MPI_Finalize`: termina la parte multiprocesso del programma e dealloca la memoria utilizzata

```
int MPI_Finalize(void);
```

Per compilare il programma si utilizza:

```
mpicc -g -Wall file.c -o exe
// g usato per debbuging e Wall per avere i warning
```

Per eseguirlo invece:

```
mpiexec --oversubscribe -n 4 ./exe
// n indica il numero di processi creati
/* oversubscribe permette di creare n processi
anche su una macchina con meno di n core*/
```

2.1 Comunicatori

Un **comunicatore** è un insieme di processi che può scambiarsi messaggi, `MPI_Init` crea un comunicatore ganerico che comprende tutti i processi chiamato `MPI_COMM_WORLD`. All'interno di un comunicatore ogni processo ha un suo **rank** che lo identifica, che va da 0 a n-1 per un comunicatore con n processi. Alcune funzioni utili che riguardano i comunicatori sono:

- `MPI_Comm_size`: restituisce il numero di processi nel comunicatore

```
int MPI_Comm_size(
    MPI_Comm comm,    // comunicatore in input
    int* comm_sz_p    // variabile di output
);
```

- `MPI_Comm_rank`: restituisce il rank del processo chiamante all'interno del comunicatore

```
int MPI_Comm_rank(
    MPI_Comm comm,    // comunicatore in input
    int* my_rank_p    // variabile di output
);
```

L'ordine con cui vengono eseguiti i processi è casuale, quindi il processo con rank 1 potrebbe terminare prima del processo con rank 0.

2.2 Messaggi

Per poter comunicare tra loro i processi devono scambiarsi dei messaggi tramite due funzioni, `MPI_Send` e `MPI_Recv`. I messaggi devono essere **nonovertaking**, cioè se un mittente manda due messaggi ad un destinatario, l'ordine deve essere mantenuto.

- `MPI_Send`:

```
int MPI_Send(
    void* msg_buf_p,  // puntatore ai dati da inviare
    int msg_size,     // numero di elementi da inviare
    MPI_Datatype msg_type,  // tipo di dati da inviare
    int dest,  // rank del destinatario nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm   // comunicatore
);
/* il numero di elementi da inviare non va scritto in byte,
ma proprio in numero */
```

- `MPI_Recv`:

```
int MPI_Recv(
    void* msg_buf_p,  // variabile in output
    int msg_size,     // numero di elementi da ricevere
    MPI_Datatype msg_type,  // tipo di dati da ricevere
    int source,  // rank del mittente nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm,  // comunicatore
    MPI_Status* status_p   // status dell'operazione
);
/* il numero di elementi da ricevere non va scritto in byte,
ma proprio in numero */
```

Per quanto riguarda i datatype, di base esiste un datatype per ogni tipo di c, però possono esserne creati di nuovi nel caso si voglia inviare strutture o tipi particolari. I tag invece servono per differenziare dei messaggi mandati alla stessa destinazione, per esempio se abbiamo due tipologie di messaggi con scopi diversi.
Un messaggio inviato dal processo q al processo r verrà ricevuto correttamente se:

- il comunicatore di `q.MPI_Send` e `r.MPI_Recv` è lo stesso
- la destinazione di `q.MPI_Send` è r e il mittente di `r.MPI_Recv` è q (si può omettere il mittente in `r.MPI_Recv` inserendo come parametro `MPI_ANY_SOURCE`)
- il datatype di `q.MPI_Send` e `r.MPI_Recv` è lo stesso
- il tag di `q.MPI_Send` e `r.MPI_Recv` è lo stesso (si può omettere in `r.MPI_Recv` inserendo come parametro `MPI_ANY_TAG`)
- il numero di elementi inviati da `q.MPI_Send` è minore di quelli ricevuti da `r.MPI_Recv` (si può omettere la grandezza in `r.MPI_Recv`)

Il parametro `MPI_Status` da informazioni sull'operazione di ricezione del messaggio se non si è specificato il mittente, il tag oppure la grandezza del messaggio. In quest'ultimo caso è possibile usare la funzione `MPI_Get_count` con input lo status e il datatype ricevuto per ottenere la quantità di dati che si è ricevuta dal messaggio.

2.2.1 Modalità di comunicazione Point-to-Point

`MPI_Send` usa la modalità di comunicazione **standard**, cioè che decide autonomamente in base alla grandezza del messaggio se bloccare la chiamata, aspettando che ci sia qualche processo pronto abbia ricevuto il messaggio, oppure ritornare prima di ricevere la conferma di ricezione, copiando in un buffer temporaneo il messaggio. Il primo metodo viene usato se il messaggio è molto grande e non si può allocare un buffer di quelle dimensione, questo fa diventare `MPI_Send` **localmente bloccante**.
Ci sono altre tre modalità possibili (tra parentesi il nome della funzione che usa questa modalità):

- **Buffered** (`MPI_Bsend`): in questa modalità l'operazione è sempre localmente bloccante e ritornerà non appena il messaggo verrà copiato sul buffer, che deve essere fornito dall'utente.

- **Sincrona** (`MPI_Ssend`): in questa modalità l'operazione è globalmente bloccante e ritorna solo dopo che il messaggio è stato ricevuto dal destinatario. Permette al mittente di sapere a che punto è il ricevente.
- **Ready** (`MPI_Rsend`): in questa modalità l'invio ha esito positivo solo se il ricevente è pronto a ricevere sennò fallisce ritornando un errore. Permette di ridurre il numero di operazioni di handshaking.

2.2.2 Comunicazione non bloccante

Le comunicazioni bloccanti sono considerate poco performanti perchè il mittente potrebbe bloccarsi, le comunicazioni non bloccanti invece permettono di massimizzare la concorrente e processare altro mentre si inviano o ricevono dati. Il difetto è che per sapere se un'operazione è stata completata o no bisogna chiederlo esplicitatamente, nel mittente per sapere se possiamo riutilizzare il buffer del messaggio, nel ricevente per sapere se possiamo iniziare a processare il messaggio ricevuto. Le comunicazioni non bloccanti posso essere accoppiate con qualunque delle modalità di comunicazione: `MPI_ISend`, `MPI_Ibsend`, `MPI_Issend`,`MPI_Irsend`. Le comunicazioni non bloccanti esistono sia con `MPI_Isend` che con `MPI_Irecv`:

- `MPI_Isend`:

```
int MPI_Send(
    void* msg_buf_p,  // puntatore ai dati da inviare
    int msg_size,     // numero di elementi da inviare
    MPI_Datatype msg_type,  // tipo di dati da inviare
    int dest,  // rank del destinatario nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm,   // comunicatore
    MPI_Request *req  // output
);
```

- `MPI_Irecv`:

```
int MPI_Recv(
    void* msg_buf_p,  // variabile in output
    int msg_size,     // numero di elementi da ricevere
    MPI_Datatype msg_type,  // tipo di dati da ricevere
    int source,  // rank del mittente nel comunicatore
    int tag,   // tag opzionale
    MPI_Comm comm,  // comunicatore
    MPI_Request* *req   // output
);
```

Viene aggiunto nella `MPI_Isend`, e sostituito a `MPI_Status` nella `MPI_Irecv`, un parametro `MPI_Request` che serve per controllare se l'operazione è finita dopo la chiamata della funzione. Questo controllo si può fare con diverse funzioni:

- `MPI_Wait`: aspetta fino a quando la comunicazione non è terminata

```
int MPI_Wait(
    MPI_Request *req  // request della comunicazione
```

```
    MPI_Status   // variabile in output
)
```

- `MPI_Test`: controlla se la comunicazione è terminata e ritorna una flag con valore 0 o 1

```
int MPI_Wait(
    MPI_Request *req  // request della comunicazione
    int *flag    // flag che indica il completamento
    MPI_Status   // variabile in output
)
```

Esistono anche altre funzioni che prendono in input una lista di `MPI_Request` come:

- WaitAll
- TestAll
- WaitAny
- TestAny

2.3 Comunicazioni collettive

Le comunicazioni collettive sono comunicazioni che permettono a tutti i processi di comunicare insieme, come `MPI_Reduce`, che aggrega i dati secondo un'operazione specificata di tipo `MPI_Op`. Sono già implementate le operazioni basilari come somma, or, xor ecc... ma possono anche esserne create di nuove.

```
int MPI_Reduce(
    void* input_data_p  // dati in input
    void* output_data_p  // dati in output
    int count  // numero di dati in input
    MPI_Datatype  // tipo dei dati in input
    int dest_process // rank del processo con il risultato finale
    MPI_Comm  // comunicatore
)
```

Il puntatore al buffer dove verranno messi i dati di ouput va specificato in ogni processo anche quelli che non avranno il risultato finale.

2.4 Come progettare un programma parallelo

Per progettare un programma parallelo si utilizza la metodologia di Foster, che consiste in 4 fasi:

1. **Partizionamento:** Identificare delle task che possono essere eseguite in parallelo (che non hanno dipendenze da altre task)
2. **Comunicazione:** Daterminare quali dati devono scambiarsi i diversi task
3. **Agglomerazione o aggregazione:** Raggruppare i singoli task in task più grandi, per giustificare il costo della creazione di un nuovo processo
4. **Mapping:** Assegnare i task composti ai vari processi per minimizzare le comunicazioni necessarie

Esempio:
Se si hanno in input dei numeri decimali e si vuole creare un istrogramma per sapere quanti numeri sono contenuti in ogni intervallo intero di tipo `[i,i+1]`, il modo migliore di parallelizzare il programma con n processi, sarebbe dividere i dati in input in p parti e creare un istogramma locale in ogni processo, sommandoli tutti alla fine.

Possiamo dividere i pattern di programmazione parallela in due grandi categorie:

- **Globalmente Parallela, Localmente Sequenziale (GPLS):** il programma può svolgere diverse task in modo concorrente, con ogni task eseguito in maniera sequenziale. Alcuni pattern in questa categoria sono:
  - Single Program, Multiple Data (SPMD)
  - Multiple Program, Multiple Data (MPMD)
  - Master Worker
  - Map reduce
- **Globalmente Sequenziale, Localmente Parallela (GSLP):** il programma viene eseguito come un programma sequenziale, con alcune parti eseguite in parallelo. Alcuni pattern in questa categoria sono:
  - Fork/Join
  - Loop parallelism

2.4.1 Pattern di tipo GPLS

- **Single Program, Multiple Data (SPMD):** I programmi SPMD tengono tutta la logica in un singolo programma, un tipico esempio di come questo tipo di programmi funziona:

  1. Inizializzazione
  2. Si ottiene un ID unico: numerati da 0 che identificano i thread o processi usati. Alcuni sistemi, tipo CUDA, usano vettori come identificatori
  3. Esecuzione: ogni processo eseguire parti di codice diversi in base al suo ID
  4. Terminazione: si libera lo spazio e si salvano i risultati

- **Multiple Program, Multiple Data (MPMD):** nei casi in cui la memoria richiesta per tutti i processi sia troppa oppure si utilizzano multiple piattaforme diverse, in cui SPMD fallirebbe, si utilizza MPMD. Ha la stessa esecuzione di SPMD ma consiste nell'avere differenti programmi in base alle diverse piattaforme.

- **Master Worker:** i processi vengono divisi in due tipi: master e workers, il master deve:

  – Dare i dati per far lavorare i workers
  – Ottenere i risultati della computazione dai workers
  – Eseguire le operazioni di I/O, come accedere ad un file
  – Interagire con l'utente

  Questo tipo di pattern è utile per bilanciare il lavoro, perchè non ci sono scambi di dati dai workers, ma il master potrebbe causare bottleneck (risolvibile creando una gerarchia di master)

- **Map reduce:** variazione del pattern Master Worker, usato dal motore di ricerca di Google, in cui il master coordina tutta l'operazione, i workers eseguono due tipi di task:

  – Map: applicare una funzione ai dati, dividendoli in set di risultati parziali
  – Reduce: ottenere i risultati parziali e ne crearne uno finale

2.4.2 Pattern di tipo GSLP

- **Fork/Join:** all'inizio dell'esecuzione c'è un singolo processo o thread padre, che creerà figli in modo dinamico durante l'esecuzione o userà un pool di thread statico che eseguiranno le task. I figli devono tutti aver finito per far continuare l'esecuzione al padre. Viene usato da OpenMP/Pthread.

- **Loop parallelism:** usato per trasformare codice sequenziale in codice multiprocesso, si esegue rompendo i cicli loop in sottocicli indipendenti. I loop però devono avere una particolare forma per supportare questo pattern.