SAPIENZA
UNIVERSITÀ DI ROMA

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

Programmazione di Sistemi Embedded e Multicore

Autore:
Simone Lidonnici

29 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
);
/* msg_size non va scritto in byte,
ma indica proprio il numero di elementi */
```

- `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
);
/* msg_size non va scritto in byte,
ma indica proprio il numero di elementi */
```

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.

Se ci trovassimo in una situazione in cui due processi devono sia inviare che ricevere dei dati, se entrambi chiamassero prima la `MPI_Send` e poi la `MPI_Recv` ci sarebbe il rischio di deadlock (risolvibile anche con le send non blccanti spiegate al paragrafo successivo). Per questo esiste una funzione che permette sia di ricevere che di mandare dati contemporaneamente, senza il rischio di deadlock, chiamata `MPI_Sendrecv`:

```
int MPI_Sendrecv(
    void* send_buf_p,
    int send_buf_size,
    MPI_Datatype send_buf_type,
    int dest,
    int send_tag,
    void* recv_buf_p,
    int recv_buf_size,
    MPI_Datatype recv_buf_type,
    int source,
    int recv_tag,
    MPI_Comm communicator,
    MPI_Status* status_p
);
```

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` e `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 all'interno di un comunicatore di comunicare insieme, ciò permette di risparmiare tempo e deadlock vari. Bisogna seguire delle regole quando si usano le comunicazioni collettive:

- Tutti i processi devono chiamare la stessa funzione collettiva
- Il datatype deve essere uguale in tutti i processi e anche il processo di destinazione
- Le comunicazioni collettive non hanno tag quindi vengono eseguite in ordine in base a come vengono chiamate

Una funzione collettiva è `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.

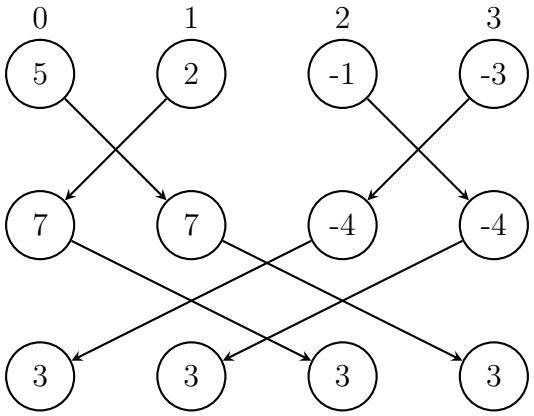
```
int MPI_Reduce(
    void* input_data_p  // dati in input
    void* output_data_p  // dati in output
    int count  // numero di dati in input
    MPI_Datatype  // tipo dei dati in input
    int dest_process // rank del processo con il risultato finale
    MPI_Comm  // comunicatore
)
```

Il puntatore al buffer dove verranno messi i dati di ouput va specificato in ogni processo (può essere `NULL`) anche quelli che non avranno il risultato finale.
Un'altra funzione collettiva è `MPI_Bcast`, che permette a un singolo processo di inviare dei dati a tutti gli altri nel comunicatore.

```
int MPI_Bcast(
    void* data_p  // dati in input o otput
    int count  // numero di dati in input
    MPI_Datatype  // tipo dei dati in input
    int source_process // rank del processo che invia i dati
    MPI_Comm  // comunicatore
)
```

Esiste anche una funzione che unisce `MPI_Reduce` e `MPI_Bcast` chiamata `MPI_Allreduce`, che aggrega i dati secondo un'operazione e poi manda il risultato a tutti i processi. Se fosse implementata effettivamente come una Reduce e poi una Bcast sarebbe molto inefficiente, quindi viene implementata con un pattern a farfalla.

Ad esempio una somma verrebbe effettuata con il pattern:



2.3.1 Comunicazioni collettive su array

Per un vettore di un processo in modo equo tra tutti i processi di un comunicatore, possiamo usare la funzione `MPI_Scatter` che permette di decidere quanti elementi mandare ad ognuno degli altri processi:

```
int MPI_Scatter(
    void* send_buf_p, // utile solo nel processo che invia
    int send_count,
    MPI_Datatype send_type,
    void* recv_buf_p,
    int recv_count,
    MPI_Datatype recv_type,
    int src_proc,   // rank del processo che invia
    MPI_Comm communicator
);
/* send_count e recv_count indicano il numero
di elementi da inviare e ricevere per ogni processo,
non il numero totale */
```

Gli elementi vengono mandati in ordine di rank, cioè se invio ad ogni processo 3 elementi il rank 0 avrà gli elementi 0-2, il rank 2 gli elementi 3-5 ecc..., e il processo che invia avrà comunque gli elementi che gli spettano nel buffer di ricezione. Ad esempio se chiamo la funzione:

```
MPI_Scatter(buff,3,MPI_INT,dest,3,MPI_INT,0,MPI_COMM_WORLD);
```

e il processo con rank 0 ha nel buffer:

Rank 0	0	1	2	3	4	5	6	7	8

dopo la chiamata della funzione i buffer di destinazione dei processi avranno:

Rank 0	0	1	2
Rank 1	3	4	5
Rank 2	6	7	8

Si può sostituire nel processo che invia il vettore il buffer di destinazione con `MPI_IN_PLACE` per far mettere la parte di vettore del processo chiamante nello stesso buffer che contiene il vettore completo, sovrascrivendolo.
Per unire poi i sottovettori posseduti da ogni processo in un unico vettore, possiamo usare la funzione `MPI_Gather`:

```
int MPI_Gather(
    void* send_buf_p,
    int send_count,
    MPI_Datatype send_type,
    void* recv_buf_p, // utile solo nel processo che riceve
    int recv_count,
    MPI_Datatype recv_type,
    int dest_proc,   // rank del processo che riceve
    MPI_Comm communicator
);
/* send_count e recv_count indicano il numero
di elementi da inviare e ricevere per ogni processo,
non il numero totale */
```

Anche in questo caso i sottovettori vengono uniti in ordine in base al rank, cioè i primi elementi saranno del rank 0 i secondi del rank 1 ecc.... Esiste anche una funzione che permette di fare `MPI_Gather` e `MPI_Bcast` contemporaneamente, chiamata `MPI_Allgather` (implementata in un modo più efficente rispetto a una gather e poi una broadcast).

2.3.2 Comunicazioni collettive su matrici

Una matrice $n \times m$ allocata in modo statico viene memorizzata in memoria come un vettore con nm elementi ordinati per righe, quindi per eseguire delle comunicazioni collettive (ma anche send e recv) basta specificare un numero di elementi di $n \cdot m$. Per allocarle dinamicamente possiamo fare in due modi:

- Lista di puntatori a righe:

```
int** m;
m = (int**) malloc(sizeof(int*)*n_righe);
for(int i=0;i<n_righe,i++){
    a[i] = (int*) malloc(sizeof(int)*n_col);
}
```

- Array monodimensionale:

```
int* m = (int*) malloc(sizeof(int)*n_col*n_righe);
```

Il primo metodo ci permette di accedere ad un elemento nella riga i e nella colonna j tramite `m[i][j]`, ma non ci permette di inviarle tramite una sola comunicazione, ma bisogna eseguirne una per ogni riga.
Con il secondo metodo per accedere ad un elemento nella riga i e nella colonna j dobbiamo scrivere `m[i*n_col+j]`, però ci permette di inviare la matrice con una sola comunicazione di $n \cdot m$ elementi, risparmiando tempo.

Altre funzioni che possono essere utili per vettori e matrici sono:

- **Reduce-Scatter**: esegue l'operazione specificata sui vettori dei processi e poi divide il vettore risultante tra i processi
- **Alltoall**: permette di dividere i vettori dei processi combinandoli, cioè ogni processo avrà all'inizio del suo vettore risultante una parte del vettore del processo con rank 0 (molto utile per fare la trasposizione di matrici). Per esempio se i processi hanno i vettori:

Rank 0	0	1	2
Rank 1	3	4	5
Rank 2	6	7	8

  dopo la chiamata della funzione i buffer di destinazione dei processi avranno:

Rank 0	0	3	6
Rank 1	1	4	7
Rank 2	2	5	8

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.

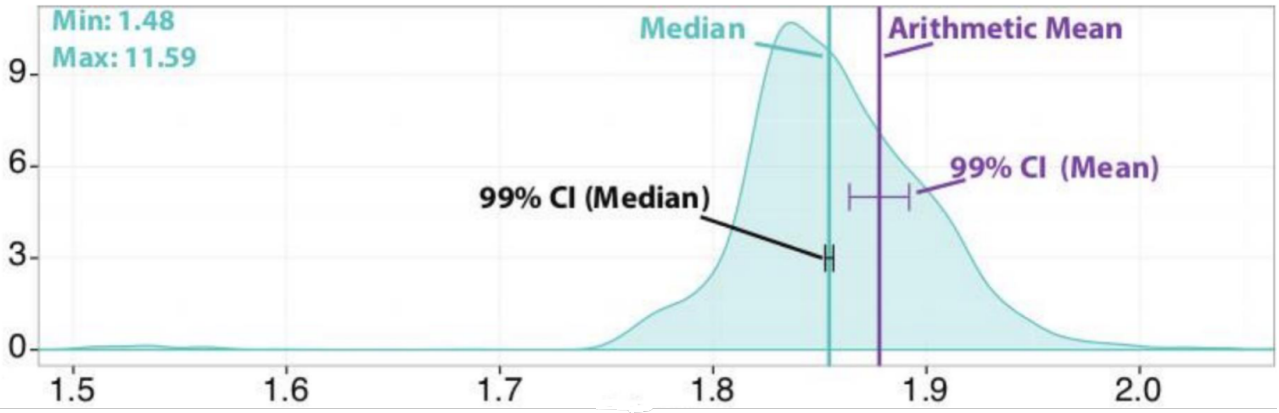
- **Loop parallelism:** usato per trasformare codice sequenziale in codice multiprocesso, si esegue rompendo i cicli loop in sottocicli indipendenti. I loop però devono avere una particolare forma per supportare questo pattern.

2.5 Calcolo delle performance

Per sapere quanto è efficente un programma parallelo bisogna innanztutto sapere quanto tempo ci mette ad essere eseguito, per fare ciò esiste una funzione `MPI_Wtime` che ritorna il tempo attuale (in base al tempo di accensione della macchina o altre variabili interne). Per sapere il tempo di esecuzione di un programma chiamiamo `MPI_Wtime` all'inizio e alla fine dell'esecuzione e facciamo la differenza. Essendo però un programma parallelo ogni processo calcolerà il proprio tempo, bisogna quindi prendere il maggiore, tramite una `MPI_Allreduce`.
I vari processi però potrebbero iniziare in momenti diversi, per assicurarsi che i processi inizino insieme esiste una funzione chiamata `MPI_Barrier`, che una volta chiamata da un processo, non permette di andare avanti finchè tutti i processi non hanno chiamato la funzione. Questo non assicura che i processi usciranno dalla funzione contemporaneamente, ma solo che il primo processo uscirà quando ttti ci saranno entrati (è comunque una stima abbastanza buona).

2.5.1 Statistiche sui tempi

Una sola misurazione non è abbastanza per sapere quanto è efficiente il programma, è meglio misurare più volte e riportare un grafico della distribuzione dei tempi.
Esempio:



Speedup

Dato il tempo del programma seriale $T_s(n)$ e il tempo del programma parallelo con p processi $T_p(n,p)$, definiamo lo **speedup** come:

$$S(n,p) = \frac{T_s(n)}{T_p(n,p)}$$

Il valore ideale sarebbe $S(n,p) = p$, in questo caso si dice che il programma ha **speedup lineare**.

Da notare che $T_s(n)! = T_p(n,1)$, e solitamente $T_s(n) \leq T_p(n,1)$. Questa seconda tempistica $T_p(n,1)$ si usa per calcolare la scalabilità, cioè:

$$Sc(n,p) = \frac{T_p(n,1)}{T_p(n,p)}$$

Efficienza

L'**efficienza** ci dice quanto il programma parallelo spreca risorse rispetto al programma sequenziale:

$$E(n,p) = \frac{T_s(n)}{T_p(n,p) \cdot p}$$

Il valore ideale sarebbe $E(n,p) = 1$, cioè con p processi il programma va p volte più veloce.

2.5.2 Strong e Weak scaling

Ci sono due ipi di scaling che un programma può avere:

- **Strong scaling:** quando fissata una dimensione del problema, aumentando il numero di processi il programma ha un'efficienza vicina a 1.
- **Weakscaling:** quando aumentando la dimensione del problema in modo parallelo rispetto al numero di processi il programma ha un'efficienza vicina a 1.

Legge di Amdahl

La legge di Amdahl ci permette di stimare quale sarà lo speedup massimo del nostro programma, in base alla quantità di programma che possiamo parallelizzare, data α la percentuale di programma che si può parallelizzare:

$$T_p(n,p) = (1-\alpha)T_s(n) + \alpha\frac{T_s(n)}{p}$$

Da questo possiamo calcolare lo speedup:

$$S(n,p) = \frac{T_s(n)}{(1-\alpha)T_s(n) + \alpha\frac{T_s(n)}{p}}$$

Ponendo l'equazione con p tendente a infinito:

$$\lim_{p\to+\infty} S(n,p) = \frac{1}{1-\alpha}$$

Questa legge da un massimo speedup solo nel caso di strong scaling.

Legge di Gustafson

La legge di Gustafson ci permette di stimare lo speedup massimo del nostro programma considerando il weak scaling, cioè all'aumentare dei processi aumenterà anche la percentuale α di parte parallelizzabile:

$$S(n,p) = (1-\alpha) + \alpha p$$

Questo viene anche chiamato **speedup scalato**.

2.6 Datatype custom

I datatype custom permettono di rappresentare un insieme di dati in memoria, sapendo sia il tipo degli elementi sia la loro posizione relativa in memoria. In questo modo sia la funzione che invia i dati che la funzione che li riceve sanno come posizionarli correttamente in memoria, sennò potrebbe succedere che in base a diverse versioni di sistema o di compilatore vengano immagazzinati in modo diverso. In pratica un datatype custom è un insieme di datatype con il loro offset relativo (in base al primo elemento).
La funzione per creare un nuovo Datatype è `MPI_Type_create_struct`:

```
int MPI_Type_create_struct(
    int count, // numero di elementi
    int array_lengths[], // lunghezza degli elementi
    MPI_Aint array_offset,  // offset relativo degli elementi
    MPI_Datatype array_types[], // tipo degli elementi
    MPI_Datatype* new_type_p  // output del nuovo tipo
);
```

Prima di usarla bisogna però chiamare la funzione `MPI_Type_commit`.
Esempio:
Data la struct:

```
typedef struct T{
    float a;
    float b;
    int n;
};
```

con indirizzi delle variabili:

Variabile	Indirizzo
a	24d
b	40d
n	48d

possiamo definire un Datatype custom per questa struttura con la funzione:

```
MPI_Type_create_struct(
    count = 3,
    array_length = [1,1,1],
    array_offset = [0,16,24],
    array_types = [MPI_DOUBLE,MPI_DOUBLE,MPI_INT],
    new_type_p
);
```

Solitamente l'indirizzo di una variabile `a` è uguale al suo puntatore (`&a`), ma in alcuni casi potrebbero differire, quindi per sicurezza conviene usare la funzione `MPI_Get_address` per ottenere l'indirizzo di una determinata variabile dal suo puntatore. Quando si è finito di usare il nuovo datatype definito si può liberare la memoria con `MPI_Type_free`.

3

Pthreads

3.1 Sistemi a memoria distribuita

I sistemi a **memoria distribuita** sono formati diversi nodi (detti anche server o blade), interconnessi da una rete. Ogni nodo è formato da una CPU multicore, una o più GPU e delle interfaccie di rete (NIC) che connettono il nodo alla rete. Avendo ogni nodo la propria memoria, per parallelizzare l'intera serie di nodi conviene usare MPI, invece per parallelizzare i core all'interno della CPU di ogni noda conviene usare un approccio con memoria condivisa come **Pthreads** (o OpenMP).

3.2 Threads

I processi sono un istanza di un programma, ognuno con la propria memoria, i threads sono analoghi ad un processo "meno pesante" (light-weight) che costa meno da creare e hanno memoria condivisa tra loro. Ovviamente i thread hanno comunque stack diversi perchè potrebbero eseguire parti di codice differenti. La libreria **POSIX Threads**, abbreviata con Pthreads è una libreria standard per i sistemi operativi basati su UNIX e va linkata con il programma in C che si vuole eseguire con diversi threads importando l'header `pthread.h`. Un programma che utilizza Pthread si compila con `gcc` ma aggiungendo alla fine `-lpthread` per specificare la libreria di Pthread, poi si runna come qualsiasi programma C.

3.2.1 Creare un thread

Per creare un thread si usa la funzione `pthread_create`:

```
int pthread_create(
    pthread_t* thread_p,  // puntatore al thread creato
    const pthread_attr_t* attr_p, // attributi del thread
    void* (*start_routine) (void*), // funzione da eseguire
    void* arg_p  // argomenti della funzione
);
/* gli attributi possono anche essere settati a NULL per
creare un thread senza attributi particolari /*
```

Il tipo `pthread_t` è l'oggetto che rappresenta il thread e contiene tutte le informazioni per identificare il thread associato. La funzione che deve eseguire il thread indica proprio un **puntatore a funzione**, cioè un puntatore che indica la locazione di memoria in cui inizia la determinata funzione. Un puntatore a funzione può essere creato e utilizzato:

```
void* func(int a){
    // testo funzione
}

void main(){
    void(*func_ptr)(int)=func;
    *func_ptr(10); // come chiamare la funzione dal puntatore
}
```

Nel nostro caso la funzione che passiamo in input quando creiamo il thread deve ritornare un **void*** e avere come argomento un **void***, un `void*` è un puntatore che permette di essere castato a qualsiasi altro tipo di puntatore. Oltre la funzione anche gli argomenti che passiamo in input a `pthread_create` devono essere di tipo `void*`, quindi se volessimo passare più di un argomento dovremmo creare una struct e passare il puntatore alla struct (castato a `void*`) a `pthread_create` per poi ricastarlo come puntatore alla struct dentro la funzione eseguita dal thread.

Quando si vuole aspettare che un thread finisca per mandare avanti l'esecuzione del codice si può usare la funzione bloccante `pthread_join`:

```
int pthread_join(pthread_t thread, void** valur_ptr)
```

La funzione attende la fine del determinato thread passato in input (se il thread ha già terminato l'esecuzione bisogna comunque chiamare la funzione per distruggerlo) e ritorna il valore della funzione eseguita del thread.
I thread, come i processi in MPI, hanno un ID univoco che può essere ottenuto tramite `pthread_self` che ritorna l'ID del thread chiamante e si possono confrontare due thread tramite `pthread_equal` che ci dice se l'ID dei due thread è uguale (sono lo stesso thread).