UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

RIASSUNTO CALCOLO PARALLELO E DISTRIBUITO
MODULO A

Anno Accademico: 2018/2019

Studente: Sabella Gianluca

Indice generale

1 Calcolo ad alte prestazioni

Il calcolo ad alte prestazioni viene utilizzato per risolvere problemi di grandi dimensioni in “tempo reale”. Il *tempo di esecuzione di un software* è dato da:

$$\tau = k * T(n) * \mu$$

- **k** è una costante;
- **T(n)** è la *complessità dell’algoritmo*, ovvero il numero di operazioni effettuate dall’algoritmo;
- **μ** è il *tempo di esecuzione*, su un calcolatore, di un’operazione floating point.

Un modo per diminuire il tempo di esecuzione del software è diminuire una tra **T(n)** e **μ**. Ovviamente incorriamo in vincoli: si può, infatti, ridurre T(n) scegliendo un algoritmo più performante, ma tuttavia non si può ottenere una complessità migliore di quella degli algoritmi ottimali; si può diminuire **μ**, che significa migliorare la tecnologia, ma la tecnologia può essere migliorata fino ad un certe punto e non oltre.

Un ulteriore modo per diminuire **τ** è quello di servirsi del calcolo parallelo. Il ***calcolo parallelo*** consiste nel ***decomporre la dimensione di un problema in sottoproblemi; tali sottoproblemi saranno risolti contemporanemante su diversi processori***.

Ovviamente è possibile realizzare diversi tipi di parallelismo a seconda dei calcolatori che si hanno a disposizione.

2 Tassonomia Di Flynn

Secondo la tassonomia di Flynn è possibile distinguere 4 tipologie di calcolatori:

- **SISD**: acronimo di **S**ingle **I**struction **S**ingle **D**ata. Ne fanno parte i calcolatori forniti di una sola unità di controllo (CU) e una sola unità aritmetica (ALU): i cosiddetti calcolatori *monoprocessore*. Tali calcolatori sono in grado di eseguire una sola istruzione per volta su un singolo dato. In questo caso non parliamo di parallelismo.
- **MISD**: acronimo di **M**ultiple **I**nstruction **S**ingle **D**ata. Ne fanno parte i calcolatori in grado di eseguire contemporaneamente più istruzioni su un singolo dato. Anche se apparentemente sembrerebbero non esistere calcolatori del genere un esempio potrebbero essere le architetture che implementano le *pipeline*. E’ evidente come, quindi, con tale classe di calcolatori è possibile realizzare un *parallelismo temporale*. Un esempio di implementazione di *pipeline* lo si può riscontrare a livello della unità aritmetica (*ALU*): ogni *ALU* è caratterizzata da più segmenti, ognuno preposto ad una singola operazione elementare; tali segmenti potranno, dunque, essere eseguiti concorrentemente. Più nel dettaglio ogni segmento effettua una specifica operazione su una porzione di dato; produce un risultato che verrà utilizzato dal segmento successivo; mentre tale segmento processa il risultato ricevuto, il segmento precedente, parallelamente, inizia a processare un’altra porzione di dato. I segmenti continueranno a lavorare tra di loro in parallelo finché non sarà stato processato l’intero dato. Le operazioni svolte potrebbero essere paragonate ad una catena di montaggio. Dal momento che il parallelismo realizzabile avverrà a livello dell’unità aritmetica è definito anche *parallelismo ON-CHIP*.

- **SIMD**: acronimo di **S**ingle **I**nstruction **M**ultiple **D**ata. Ne fanno parte i calcolatori forniti di una sola unità di controllo (*CU*) alla quale sono collegate diverse *ALU*, che agiscono su dati diversi. Questi calcolatori sono in grado di eseguire una singola istruzione su più dati contemporaneamente. Il parallelismo realizzabile è un *parallelismo spaziale*. Una volta che la *CU* avrà deciso l'istruzione da eseguire, le *ALU* eseguiranno in parallelo tale istruzione sul proprio insieme di dati. Per come è strutturata tale tipologia di calcolatori è possibile utilizzarli per realizzare anche un parallelismo temporale a livello di una singola *ALU* (pipeline). Anche in questo caso, quindi, il *parallelismo* realizzabile può essere considerato *ON-CHIP*.
- **MIMD**: acronimo di **M**ultiple **I**nstruction **M**ultiple **D**ata. Ne fanno parte i calcolatori in grado di eseguire contemporaneamente flussi di istruzioni diversi su diversi dati. Un esempio di calcolatori MIMD sono i sistemi multiprocessore; ovvero sistemi costituiti da più CPU, ognuna delle quali associata a una propria ALU: ogni CPU stabilirà una propria istruzione da eseguire, e ogni istruzione sarà eseguita in parallelo su insiemi di dati diversi per ogni ALU. Con questi calcolatori è possibile realizzare un *parallelismo asincrono*.

  Inoltre è possibile distinguere due diversi calcolatori MIMD a seconda che le CPU siano associate ognuna a diverse porzioni di memoria (**MIMD a memoria distribuita**), oppure collegate tutte ad una stessa memoria (**MIMD a memoria condivisa**). Nel primo caso uno svantaggio sta nel fatto che, essendo collegati a diverse porzioni di memoria, per realizzare il parallelismo occorre far comunicare i processori tra loro scambiandosi dati. Nel secondo caso, un vantaggio sta nel fatto che condividendo la memoria non occorre far comunicare i processori; tuttavia allo stesso tempo occorre sincronizzare l'accesso concorrente ai dati. Il parallelismo asincrono a memoria condivisa è definito anche *parallelismo ON-CHIP*; tuttavia se consideriamo un processore multicore in cui ogni core ha una cache privata, possiamo definire anche il *parallelismo* realizzabile da un calcolatore MIMD a memoria distribuita come *ON-CHIP*.

3 Calcolatori MIMD

Come già detto, secondo la tassonomia di Flynn tra i vari tipi di calcolatori vi sono i calcolatori MIMD (Multiple Instructione Multiple Data). Con tali calcolatori si è in grado di ottenere un parallelismo asincrono a memoria distribuita o condivisa.

3.1 MIMD a Memoria Condivisa e Open MP

Un parallelismo asincrono a memoria condivisa si può realizzare quando il calcolatore presenta diverse CPU, le quali lavoreranno in maniera concorrente, collegate tutte ad una stessa memoria (condivisa). In questo caso generare diversi processi (ovvero un programma in esecuzione), come nel caso della memoria distribuita, potrebbe non portare a benefici: diversi processori, infatti, non condividono la memoria e quindi si finirebbe per non sfruttare il vantaggio della memoria condivisa ed in oltre si aggiungerebbe lo svantaggio di rallentare l'esecuzione. Una soluzione può essere quella di generare un solo processo e in esso generare più thread (ovvero l'unità base di utilizzo della CPU), facendoli lavorare in parallelo. Thread diversi di uno stesso processo condividono la memoria: in questo caso per eseguire il parallelismo non serviranno scambi di dati dal momento che

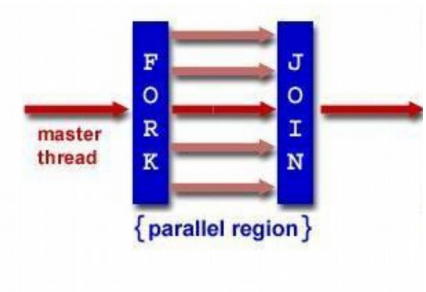
i thread accedono agli stessi dati, ma tuttavia occorrerà sincronizzarne l'accesso per evitare conflitti nell'accesso concorrente.

Il parallelismo a memoria condivisa può essere gestito tramite una particolare API (Application Program Interface): **Open MP** (**Open** specifications for **M**ulti **P**rocessing).

Un algoritmo, come sappiamo, può essere diviso in parti: parti eseguibili esclusivamente in maniera sequenziale e parti che potrebbero essere eseguite in maniera parallela. Una volta individuate tali "regioni parallelizzabili" è possibile sfruttare Open MP per ottenere un parallelismo completamente trasparente ed efficiente.

Un esempio dell'utilità di OpenMP lo si può verificare nell'esecuzione in parallelo e a memoria condivisa dell'algoritmo della somma di N numeri. Per poter permettere l'esecuzione in parallelo di tale algoritmo, ogni thread, dopo aver eseguito in parallelo il proprio sottoproblema, dovrà aggiornare una variabile condivisa in cui risiederà il risultato finale. Per poter permettere ad i thread di aggiornare la variabile tra di loro condivisa è indispensabile che l'accesso e l'aggiornamento ad essa siano sincronizzati, in modo che un solo thread alla volta vi acceda; in questo modo non vi saranno errori nell'aggiornamento. Con OpenMP tutto ciò può essere eseguito semplicemente tramite un solo comando in grado di permettere contemporaneamente l'esecuzione in parallelo dei thread e l'aggiornamento della variabile, senza rischio di aggiornamenti errati (*#pragma omp parallel for reduction(+:variabile){ blocco di istruzioni}*). Vediamo nel dettaglio il funzionamento di OpenMP.

Il modello di esecuzione parallela sfruttato da Open MP è quello **fork-join**. Tutti gli algoritmi cominciano con un solo thread, il thread master, che eseguirà la parte sequenziale iniziale; *appena comincia una regione parallela viene generato un team di thread che la eseguiranno parallelamente* (***fork***); *una volta che ogni thread del team avrà terminato le istruzioni all'interno della regione parallela, si sincronizzano e terminano, lasciando proseguire il solo thread master* (***join***).



Open MP è caratterizzato da:

- **Direttive per il compilatore**: si usano per creare un team di thread, stabilire quali istruzioni devono essere eseguite in parallelo e come queste devono essere distribuite tra i thread del team creato. Tali istruzioni sono in un formato speciale, comprensibile solo ad un compilatore dotato di supporto OpenMP.

  Una regione parallela è indicata dal costrutto

  #pragma omp

  Le direttive si applicano a tale costrutto. Tra le varie direttive vi sono:

- ***master***: in grado di far eseguire il blocco di istruzioni che lo sussegue dal solo thread master;
- ***critical***: ottimo per gestire le regioni critiche. Forza l'esecuzione del blocco di istruzioni che lo segue ad un thread alla volta. Si può inoltre assegnare un nome alla regione, così da renderla globale per tutto il programma;
- ***barrier***: forza i thread di uno stesso task ad attendere il completamento delle istruzioni PRECEDENTI da parte di tutti gli altri;
- ***parallel***: in grado di generare un team di thread ed avviare un'esecuzione parallela tra di loro. Tutte le istruzioni del blocco indicato da tale direttiva saranno svolti da tutti i thread del team in parallelo. Inoltre alla fine del blocco vi è una barriera implicita (a meno di diverse indicazioni esplicite del programmatore): i thread termineranno solo quando tutto il team avrà terminato l'esecuzione di ogni istruzione.

  A quest'ultima direttiva si possono affiancare le cosiddette direttive di **WorkSharing**: permettono di distribuire le varie istruzioni da eseguire in parallelo tra i diversi thread ed anche esse presentano una barriera implicita (a meno di diverse indicazioni esplicite del programmatore). Le direttive di **WorkSharing** sono:

  - ***for***: specifica che le istruzioni vengano divise tra i vari thread del team. Viene utilizzato il for e non il while dato che quest'ultimo necessita di una condizione di uscita, mentre con il primo già si conosce il numero di iterazioni da compiere;
  - ***sections***: genera diverse sessioni di costrutti ognuna delle quali viene eseguita da un thread del team;
  - ***single***: forza l'esecuzione del blocco di istruzioni ad un solo thread casuale. Tutti gli altri attendono che esso concluda l'esecuzione di tutto il blocco per poter terminare.

Le direttive possono essere arricchite con l'utilizzo di **clausole**. Tra le clausole principali vi sono:

- ***shared(list):*** indica che tutte le variabili in list sono pubbliche e quindi condivise tra i vari thread del team;
- ***private(list)***: indica che tutte le variabili in list sono private per ogni thread che le utilizzata. Ogni thread ne avrà dunque una copia. All'entrata e all'uscita dalla regione le variabili avranno un valore indefinito;
- ***firstprivate(list)***: indica che tutte le variabili in list sono private per ogni thread che le usa e inoltre il loro valore, all'inizio della regione parallela, è settato al valore che le originali (fuori del blocco) avevano prima di incontrare il costrutto;
- ***lastprivate(list)***: indica che le variabili in list sono priate per ogni thread che le usa e inoltre il valore delle variabili originali, dopo il blocco indicato dal costrutto, è settato all'ultimo valore che esse avevano all'interno del blocco;
- ***reduction(op: list)***: gli elementi in list sono combinati applicando su di essi l'operatore op. Ogni thread ha una copia privata delle variabili in list, e al termine del costrutto il risultato viene condiviso;

- ***nowait***: elimina la barriera implicita che vi è al termine del costrutto (e non della intera regione parallela). Una volta terminate le proprie istruzione, un thread non dovrà attendere la terminazione degli altri thread del team;
- ***schedule(x(n))***: indica il numero di istruzioni contigue che ogni thread dovrà avere (n) e il tipo di tale distribuzione (x), che può essere: *static*; *dynamic* o *guided*; *runtime*;

Non tutte le clausole, tuttavia, sono valide per ogni direttiva:

- for: private, firstprivate, lastprivate, schedule, nowait, reduction; collapse, ordered;
- sections: private, firstprivate, lastprivate, nowait, reduction;
- single: private, firstprivate, nowait; copyprivate.

- **Runtime Library Routines**: funzioni/routines utilizzate per modificare le variabili di controllo interne allo standard. Tra le varie vi sono:
  - **omp_get_num_threads()**: restituisce il numero di thread del team;
  - **omp_set_num_threads(x)**: setta a *x* il numero di thread da poter utilizzare;
  - **omp_get_thread_num()**: restituisce l'id del thread corrente;
  - **omp_set_dynamic(x)**: concede (se x = 1) o nega (se x = 0) al sistema di riadattare il numero di thread utilizzati;
  - **omp_get_num_procs()**: restituisce il numero di processi disponibili per il programma al momento della chiamata.
- **Variabili di ambiente**: permettono di modificare le variabili di controllo interne prima dell'esecuzione. Tra le varie vi è **OMP_NUM_THREADS(n)** che setta a n il numero di threads da usare nelle esecuzioni successive.

3.2 MIMD a Memoria Distribuita e MPI

Per poter effettuare un parallelismo a memoria distribuita occorre generare processori. I processori possono lavorare, infatti, in parallelo. Dal momento che ogni processore ha una propria memoria locale alla quale può accedere direttamente ma in maniera esclusiva è indispensabile che ogni processore possa conoscere i dati nella memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.

Uno strumento software per lo sviluppo di algoritmi in ambiente di calcolo MIMD a memoria distribuita, alla cui base vi è il concetto di trasferimento di dati, è **MPI** (Message Passing Interface).

In MPI ogni processore è collegato agli altri attraverso 3 diverse categorie:

- **gruppo**: ogni componente concorrente del programma (pezzo di codice) viene affidata ad un gruppo di processori. Ad ogni processore in ciascun gruppo è associato un identificativo, ovvero un intero tra 0 e (nproc-1) (con nproc numero di processori presenti nel gruppo). Processori appartenenti a uno o più gruppi possono avere identificativi diversi, ciascuno relativo ad uno specifico gruppo;

- **contesto**: ad ogni gruppo di processori è attribuito un identificativo detto contesto. Il contesto definisce l'ambito in cui avvengono le comunicazioni tra processori di uno stesso gruppo. Se un dato è inviato in un contesto, la ricezione deve avvenire nello stesso contesto.
- **communicator**: ad ogni gruppo di processori appartenenti ad uno stesso contesto è assegnato un ulteriore identificativo, il *communicator*. Esso *racchiude* tutte *le caratteristiche dell'ambiente di comunicazione* (come la topologia). Esistono due communicator principali: l'*inter-communicator* (in cui le comunicazioni avvengono all'interno di un gruppo di processori); *intra-communicator* (in cui le comunicazioni avvengono tra gruppi diversi di processori). Ogni processore fa parte di un communicator di default: MPI_COMM_WORLD.

3.2.1 Routine MPI

MPI prevede diverse funzioni standard, le routine:

- **MPI_Init(&argc, &argv)**;
  Tale routine è chiamata solo una volta e inizializza l'ambiente di esecuzione MPI e il communicator MPI_COMM_WORLD. Definisce l'insieme dei processori attivati (communicator). I due dati di input: argc ed argv sono gli argomenti del main;
- **MPI_Finalize()**;
  Tale routine determina la fine del programma MPI. Dopo questa routine non è possibile richiamare nessun'altra routine di MPI.
- **MPI_COMM_rank(MPI_COMM_WORLD, &menum)**;
  Tale routine permette al processore chiamante, appartenente al communicator MPI_COMM_WORLD, di memorizzare il proprio identificativo nella variabile menum.
- **MPI_COMM_size(MPI_COMM_WORLD, &nproc)**;
  Tale routine permette al processore chiamante di memorizzare nella variabile nproc il numero totale dei processori concorrenti appartenenti al communicator MPI_COMM_WORLD.
- **MPI_Send(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**;
  Tale routine permetta al processore che la esegue di spedire i primi count elementi di buffer, di tipo datatype, al processore con identificativo dest. Si tratta di una routine bloccante: una volta che un processore l'avrà invocata, per poter proseguire, dovrà attendere che l'intero dato si stato spedito al destinatario. Gli argomenti della routine sono:
  - *buffer indirizzo del dato da spedire;
  - count numero dei dati da spedire;
  - datatype tipo dei dati da spedire;
  - dest identificativo del processore destinatario;
  - comm identificativo del communicator;
  - tag identificativo del messaggio.
- **MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**;
  Tale routine è la reciproca di MPI_Send. Permette al processore che la esegue di ricevere i primi count elementi, del tipo datatype, in buffer dal processore che nel communicator comm ha identificativo source. Come MPI_Send, anche in questo caso si tratta di una

routine bloccante: una volta che un processore l'avrà invocata, per poter proseguire, dovrà attendere che l'intero dato si stato ricevuto dal mittente. Gli argomenti della routine sono:
  - *buffer indirizzo del dato in cui ricevere;
  - count numero dei dati da ricevere;
  - datatype tipo dei dati da ricevere;
  - source identificativo del processore da cui ricevere;
  - comm identificativo del communicator;
  - tag identificativo del messaggio;
  - status tipo predefinito di MPI che racchiude informazioni sulla ricezione del messaggio. E' un dato strutturato composto da: identificativo del processore che invia il dato, identificativo del messaggio e indicatore di errore.

- **MPI_GET_COUNT(MPI_Status *status, MPI_Datatype datatype, int *count)**;
  Tale routine permette al processore che la esegue di memorizzare nella variabile count il numero di elementi, di tipo datatype, che riceve dal messaggio e dal processore indicati nella variabile status. MPI_Status in C è un tipo di dato strutturato, composto da tre campi:
  - identificativo del processore da cui ricevere
  - identificativo del messaggio
  - indicatore di errore

- **MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)**;
  Tale routine permette al processore che la esegue di testare lo stato della comunicazione non bloccante, identificata da request. Può essere utilizzata dopo una chiamata MPI_Isend o MPI_Irecv per verificare la terminazione della ricezione o invio. La funzione MPI_Test ritorna l'intero flag:
  - flag = 1, l'operazione identificata da request è terminata;
  - flag = 0, l'operazione identificata da request NON è terminata;

- **MPI_Wait(MPI_Request *request, MPI_Status *status)**;
  Tale routine permette al processore che la esegue di controllare lo stato della comunicazione non bloccante, identificata da request, e di arrestarsi solo quando l'operazione in esame si è conclusa. In status si hanno informazioni sul completamento dell'operazione di Wait.

- **MPI_Isend(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)**;
  Come per MPI_Send, tale routine permette al processore che la esegue di spedire i primi count elementi di buffer, del tipo datatype, al processore con identificativo dest. L'oggetto request contiene informazione sulla fase di trasmissione del messaggio. Differentemente da MPI_Send non è bloccante: una volta invocata, il processore chiamante non dovrà attendere il termine dell'invio del dato. L'operazione termina quando il buffer è nuovamente riutilizzabile (inoltre si può controllare tramite la routine MPI_Test).

- **MPI_Irecv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)**;
  Come per MPI_Recv, tale routine permette al processore che la esegue di ricevere i primi count elementi in buffer, del tipo datatype, dal processore con identificativo source. L'oggetto request contiene informazione sulla fase di ricezione del messaggio (simile a status). Differentemente da MPI_Recv non è bloccante: una volta invocata, il processore chiamante non dovrà attendere il termine della ricezione. L'operazione termina quando il messaggio è stato ricevuto interamente (lo si può controllare tramite la routine MPI_Test).

- **MPI_Barrier(MPI_Comm comm)**;
  Questa routine fornisce un meccanismo sincronizzante per tutti i processori del communicator comm. Ogni processore si ferma fin quando tutti i processori di comm non eseguono MPI_Barrier.

- **MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)**;
  Il processore con identificativo root spedisce a tutti i processori del comunicator comm lo stesso dato memorizzato in *buffer. Count, datatype, comm devono essere uguali per ogni processore di comm:
  - *buffer indirizzo del dato;
  - count numero dei dati da spedire e ricevere;
  - datatype tipo del dato;
  - root identificativo del processore che spedisce il dato a tutti;
  - comm identificativo del communicator.

- **MPI_Gather(void *send_buff, int send_count, MPI_Datatype datatype, void *recv_buff, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)**;
  Ogni processore di comm spedisce il contenuto di *send_buff al processore con identificativo root (se compreso). Il processore con identificativo root concatena i dati ricevuti in recv_buff , memorizzando prima i dati ricevuti dal processore con identificativo 0, poi i dati ricevuti dal processore con identificativo 1, poi quelli ricevuti dal processore con identificativo 2, etc…:
  - *send_buff indirizzo del dato da spedire;
  - send_count numero dei dati da spedire;
  - datatype tipo dei dati da spedire;
  - *recv_buff indirizzo del dato in cui root riceve e in cui è presente al termine il risultato;
  - recv_count numero dei dati che root riceve da ogni singolo processore;
  - recv_type tipo dei dati che root riceve;
  - root identificativo del processore che riceve da tutti;
  - comm identificativo del communicator;

- **MPI_Gatherv (void *sendbuf, int sendcount, MPI_Datatype datatype, void *recv_buff, int recv_count, int *displs, MPI_Datatype recv_type, int root, MPI_Comm comm);**
  Come per la MPI_Gather, ogni processore spedisce il contenuto di *sendbuf al processore di identificativo root (se compreso). Quest'ultimo concatena i dati in *recv_buff. La differenza sostanziale è nella presenza di un ulteriore parametro, il vettore displs: displs[i] indica la posizione in cui inserire i primi recv_count dei dati inviati dal processore di identificativo "i", in recv_buff, partendo dalla sua posizione iniziale (0). (dei dati inviati dal processore di identificativo "i" saranno quindi posizionati i primi recv_count dati a partire da recv_buff[displs[i]]).

- **MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)**;
  Tale routine fa in modo che ogni processore di comm spedisce il contenuto di *send_buff ad ogni altro processore (se compreso). I dati vengono concatenati in *recv_buff , memorizzando prima i dati ricevuti dal processore 0, poi i dati ricevuti dal processore 1, ecc… E' una routine simile alla MPI_Gather con la differenza che i dati concatenati vengono inviati a tutti i processori e non solo a root.

- **MPI_Scatter(void *send_buff, int send_count, MPI_Datatype send_type, void *recv_buff, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)**;
  Il processore con identificativo root distribuisce i dati contenuti in send_buff tra tutti i processori del communicator comm (se compreso). Il contenuto di send_buff viene suddiviso in nproc segmenti ciascuno di lunghezza send_count. Il primo segmento viene affidato al processore con identificativo 0, il secondo al processore con identificativo 1, il terzo al processore con identificativo 2, etc…:
  - *send_buff indirizzo del dato da spedire;
  - send_count numero dei dati da spedire;
  - send_type tipo dei dati da spedire;
  - *recv_buff indirizzo del dato in cui ricevere;
  - recv_count numero dei dati da ricevere;
  - recv_type tipo dei dati da ricevere;
  - root identificativo del processore che spedisce a tutti;
  - comm identificativo del communicator.

- **MPI_Scatterv(void *send_buff, int send_count, int *displs, MPI_Datatype send_type, void *recv_buff, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm);**
  Tale routine come MPI_Scatter fa in modo che il processore con identificativo root distribuisca i dati contenuti in send_buff a tutti i processori di comm. Al processore con identificativo “i” vengono affidati i send_count elementi a partire dalla posizione send_buff[displs[i]].

- **MPI_Reduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)**;
  Tale routine permette di effettuare operazioni combinate. Tutti i processori di comm combinano i propri dati memorizzati in *operand utilizzando l’operazione op. Il risultato viene memorizzato in *result di proprietà del processore con identificativo root. Count, datatype, comm devono essere uguali per ogni processore di comm:
  - *operand indirizzo dei dati di ogni singolo processore su cui effettuare l’operazione;
  - *result indirizzo del dato contenente il risultato combinato;
  - count numero dei dati su cui effettuare l’operazione;
  - datatype tipo degli elementi da spedire;
  - op operazione effettuata;
  - root identificativo del processore che conterrà il risultato;
  - comm identificativo del communicator.

  La routine è particolarmente efficiente in quanto prevede comunicazioni che rispettano uno schema ad albero e sfrutta la proprietà associativa e commutativa.

- **int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)**;
  Tale routine permette di effettuare operazioni combinate. Tutti i processori di comm combinano i propri dati memorizzati in *sendbuf utilizzando l’operazione op. Il risultato viene memorizzato in *recvbuf posseduto da ogni processore. E’ quindi simile a MPI_Reduce con la differenza che il risultato verrà distribuito a tutti i processori di comm, e non solo quello di identificativo root.

- **MPI_Cart_create(MPI_Comm comm_old, int dim, int *ndim, int *period, int reorder, MPI_Comm *new_comm)**;
  Operazione collettiva che restituisce un nuovo communicator new_comm in cui i processi sono organizzati in una griglia di dimensioni dim. L'i-esima dimensione ha lunghezza ndim[i]. Se period[i]=1, la i-esima dimensione della griglia è periodica; non lo è se period[i]=0:
  - comm_old communicator di input;
  - dim numero di dimensioni della griglia;
  - *ndim vettore di dimensione dim contenente le lunghezze di ciascuna dimensione;
  - *period vettore di dimensione dim contenente la periodicità di ciascuna dimensione;
  - reorder permesso di riordinare i menum (1=si; 0=no);
  - *new_comm communicator di output associato alla griglia.

- **MPI_Cart_coords(MPI_Comm comm_grid, int menum_grid, int dim, int *coordinate)**;
  Operazione collettiva che restituisce a ciascun processo di comm_grid con identificativo menum_grid, le sue coordinate all'interno della griglia predefinita. Coordinate è un vettore di dimensione dim, i cui elementi rappresentano le coordinate del processo all'interno della griglia.
  - *comm_grid identificativo di communicator;
  - menum_grid identificativo del processore del quale restituire le coordinate;
  - *coordinate vettore contenente le coordinate del processore;
  - dim dimensione del vettore *coordinate.

3.3 Esempi Parallelismo Asincrono a Memoria Distribuita

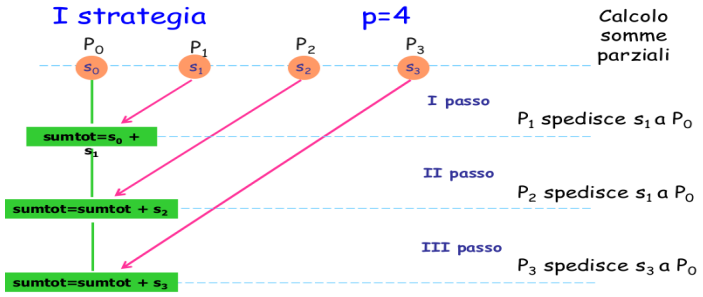
Possiamo considerare, come esempi, tre situazioni classiche che ci permettono di sfruttare al meglio il calcolo parallelo verificandone i vantaggi:

- somma di N numeri;
- prodotto di una matrice per un vettore
- prodotto tra due matrici

3.3.1 Somma di N numeri

Se volessimo calcolare la somma di N numeri su un calcolatore monoprocessore dovremmo effettuare N-1 addizioni. Sfruttando il parallelismo su un calcolatore MIMD con p processori si può dividere il problema in N/p sottoproblemi; ogni sottoproblema viene assegnato a uno dei p processori che lavorerà in parallelo agli altri; in fine è essenziale sommare i risultati ottenuti dai p processori ottenendo il risultato finale. Una volta che ogni processore avrà calcolato la sua somma parziale esistono 3 strategie di comunicazione per poter ottenere il risultato finale.

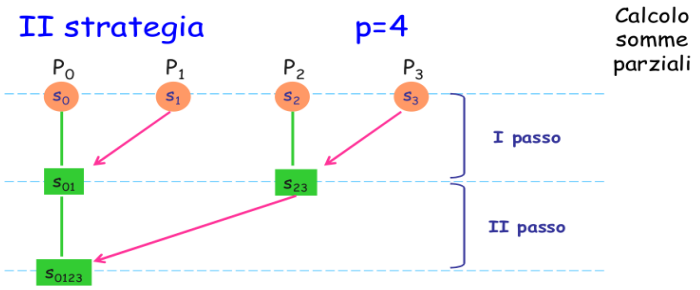
Strategia I
Ogni processore calcola la propria somma parziale in parallelo agli altri. Ognuno invierà tale somma parziale ad un unico processore prestabilito, il quale sarà il solo ad avere il risultato finale. I passi di comunicazione sono pari a p-1. E' evidente come in tale strategia non si sfrutta al meglio ogni processore: una volta inviata la propria somma parziale ogni processore terminerà di lavorare.



Strategia II

Ogni processore calcola la propria somma parziale in parallelo agli altri. La comunicazione delle somme parziali avverrà per coppie di processori; le coppie lavoreranno in parallelo: ad ogni i-esimo passo di comunicazione in ogni coppia un processore riceverà la somma parziale dall'altro che invierà la propria somma parziale. I passi di comunicazione sono pari a log(p). Al termine, anche in questo caso, solo un processore prestabilito avrà il risultato finale.

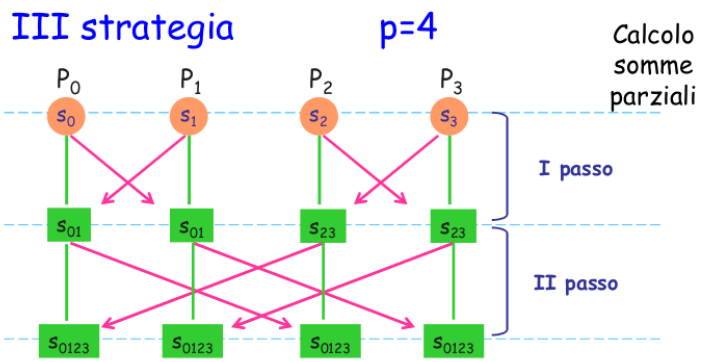
Rispetto alla strategia I è evidente come i processori vengano sfruttati meglio.



Strategia III

Ogni processore calcola la propria somma parziale in parallelo agli altri. Anche in questo caso, come per la strategia II, la comunicazione avverrà tra coppie di processori; le coppie, anche in questo caso, lavoreranno in parallelo. La differenza sostanziale è che ad ogni passo di comunicazione per ogni coppia i processori si scambieranno le somme parziali bilateralmente. I passi di comunicazione, anche in questo caso, sono pari a log(p). Al termine, quindi, ogni processore avrà la somma finale.

E' evidente come in questa strategia, sebbene vengano fatte più operazioni, il vantaggio è che ogni processore ha già il risultato finale e quindi se mai ad uno di questi servisse non ci sarà bisogno di ulteriori comunicazioni.



3.3.2 Prodotto di una Matrice per un Vettore

Se volessimo calcolare il prodotto di una matrice *A* di dimensione n*m con un vettore *b* di dimensione m su un calcolatore monoprocessore dovremmo effettuare un numero di operazioni pari a *(m prodotti + (m-1) somme)*n.* Sfruttando il parallelismo su un calcolatore MIMD con p processori, si può dividere il problema in sottoproblemi; ogni sottoproblema viene assegnato a uno dei p processori che lavorerà in parallelo agli altri; in fine si può unire i risultati ottenuti dai p processori in un unico risultato.

Esistono 3 strategie di calcolo delle operazioni parziali e delle comunicazioni di esse, tra i vari processori:

Strategia I
La matrice *A* è divisa per blocchi di righe, ogni blocco viene assegnato a un processore. Inoltre il vettore *b* è assegnato interamente ad ogni processore. Ogni processore avrà, quindi, n/p righe e m colonne di *A* e gli interi elementi di *b*. A questo punto ogni processore può calcolare, parallelamente agli altri, n/p prodotti scalari, che corrispondono a n/p componenti del vettore risultato.

Al termine è possibile, eventualmente, scambiare le componenti ottenute in modo che uno o tutti i processori abbiano il risultato finale (ovvero tutte le componenti), per un totale di log(p) passi di comunicazione.

Strategia II
La matrice A è divisa per blocchi di colonne, ogni blocco è assegnato a un processore. Ogni processore avrà n righe e m/p colonne di A. Il vettore b è diviso in m/p parti, ogni parte è assegnata a un processore. Così come è diviso il problema ogni processore può calcolare, parallelamente agli altri, solo un contributo per ogni componente del vettore risultato. Al termine dei calcoli parziali occorre che i processori si scambino i contributi così da poterli sommare e ottenere le componenti del vettore risultato, per un totale di log(p) passi di comunicazione.

Lo scambio e la somma dei contributi può avvenire in maniera analoga alla strategia II della somma se si vuole che il risultato sia posseduto da un solo processore prestabilito: coppie di processori saranno tali da lavorare in parallelo; per ogni coppia un processore invierà i propri contributi, l’altro riceverà e sommerà i contributi ricevuti ai suoi.

Se si vuole che le componenti finali siano possedute da tutti i processori, lo scambio sarà analogo alla strategia III della somma: coppie di processori saranno tali da lavorare in parallelo; in ogni coppia entrambi i processore invieranno i propri contributi all’altro processore, ed entrambi riceveranno e sommeranno i contributi ricevuti ai propri.

Strategia III
La matrice A è divisa per blocchi di righe e colonne, ogni blocco è assegnato a un processore. I processori sono distribuiti come una griglia di dimensione p/2 * p/2 (ovviamente occorre che il numero di processori sia un quadrato perfetto). Ogni processore avrà n/p/2 righe e m/p/2 colonne. Il vettore b è diviso in m/p/2 parti e ogni parte è assegnata ai processori che fanno parte della stessa colonna sulla griglia di processori (la prima parte ai processori sulla prima colonna della griglia, la seconda parte ai processori sulla seconda colonna della griglia, ecc...). Così come è diviso il problema ogni processore sarà in grado di calcolare, parallelamente a tutti gli altri, un contributo di n/p/2 componenti del vettore risultato. Al termine dei calcoli parziali occorre che i processori sulla

stessa riga della griglia di processori si scambino i contributi, così da poterli sommare ed ottenere le n/p/2 componenti complete.

In fine è possibile, eventualmente, come nel caso della strategia II scambiarsi le componenti così che uno o tutti i processori abbiano tutte le componenti del vettore risultato.

3.3.3 Prodotto tra due Matrici

Se volessimo calcolare il prodotto di una matrice *A* di dimensione n*m per una matrice B di dimensione m*r su un calcolatore monoprocessore dovremmo effettuare un numero di operazioni pari a *(m prodotti + (m-1) somme)*n*r.* Sfruttando il parallelismo su un calcolatore MIMD con p processori si può dividere il problema in sottoproblemi; ogni sottoproblema viene assegnato a uno dei p processori che lavorerà in parallelo agli altri; in fine si può unire i risultati ottenuti dai p processori in un unico risultato.

Esistono 3 strategie di calcolo delle operazioni parziali e delle comunicazioni di esse, tra i vari processori:

Strategia I
La matrice *A* è divisa per blocchi di righe e la matrice B è divisa per blocchi di colonne, ogni blocco viene assegnato a un processore. Ogni processore avrà, quindi, n/p righe e m colonne di *A* e m righe e r/p colonne di B. A questo punto ogni processore può calcolare, parallelamente agli altri, n/p * r/p prodotti scalari, che corrispondono a n/p * r/p componenti del vettore risultato. A questo punto è indispensabile che i processori si scambino i propri blocchi di colonne di B (ogni processore dovrà inviare il proprio blocco di colonne di B ai p-1 processori rimanenti, per un totale di p-1 passi di comunicazione). Dopo gli scambi, ogni processore sarà in grado di calcolare altre componenti. Al termine ogni processore avrà calcolato n/p * r componenti della matrice risultato.

Al termine è possibile, eventualmente, scambiare le componenti ottenute in modo che uno o tutti i processori abbiano il risultato finale (ovvero tutte le componenti), per un totale di log(p) passi di comunicazione.

Strategia II
La matrice *A* è divisa per blocchi di colonne e la matrice B è divisa per blocchi di righe, ogni blocco viene assegnato a un processore. Ogni processore avrà, quindi, n righe e m/p colonne di *A* e m/p righe e r colonne di B. Così come è diviso il problema ogni processore può calcolare, parallelamente agli altri, solo un contributo per ogni componente del vettore risultato (n*r). Al termine dei calcoli parziali occorre che i processori si scambino i contributi così da poterli sommare e ottenere le componenti del vettore risultato, per un totale di log(p) passi di comunicazione.

Lo scambio e la somma dei contributi può avvenire in maniera analoga alla strategia II della somma se si vuole che il risultato sia posseduto da un solo processore prestabilito: coppie di processori saranno tali da lavorare in parallelo; per ogni coppia un processore invierà i propri contributi, l’altro riceverà e sommerà i contributi ricevuti ai suoi.

Se si vuole che le componenti finali siano possedute da tutti i processori, lo scambio sarà analogo alla strategia III della somma: coppie di processori saranno tali da lavorare in parallelo; in ogni coppia entrambi i processore invieranno i propri contributi all’altro processore, ed entrambi riceveranno e sommeranno i contributi ricevuti ai propri.

Strategia III
Sia la matrice A che la matrice B sono divise per blocchi di colonne, ogni blocco viene assegnato a un processore. Ogni processore avrà, quindi, n righe e m/p colonne di *A* e m righe e r/p colonne di B. A questo punto ogni processore può calcolare, parallelamente agli altri, un contributo di alcuni componenti (n * r/p) della matrice risultato finale. A questo punto è indispensabile che i processori si scambino i propri blocchi di colonne di A (ogni processore dovrà inviare il proprio blocco di colonne di A ai p-1 processori rimanenti, per un totale di p-1 passi di comunicazione). Dopo gli scambi, ogni processore sarà in grado di calcolare altri contributi e sommare tutti i contributi ottenuti. Al termine ogni processore avrà calcolato n * r/p componenti della matrice risultato.

Al termine è possibile, eventualmente, scambiare le componenti ottenute in modo che uno o tutti i processori abbiano il risultato finale (ovvero tutte le componenti), per un totale di log(p) passi di comunicazione.

3.3.4 Bilanciamento

Sappiamo ormai come il parallelismo consista nel dividere un problema in sottoproblemi ed assegnare tali sottoproblemi ottenuti ad i vari processori, che li eseguiranno in parallelo. La distribuzione del carico di lavoro tra i processori è tuttavia un aspetto delicato del parallelismo. Una cattiva distribuzione del carico di lavoro tra i processori può causare tempi di attesa e di sincronizzazione che inevitabilmente tenderebbero a far aumentare il tempo di esecuzione. Il tempo di esecuzione può quindi dipendere anche dal bilanciamento.

Se consideriamo ad esempio il problema della somma di 100 numeri, con a disposizione 4 processori, possiamo tranquillamente dividere il problema in 4 sottoproblemi da 25 numeri senza incorrere in problemi sul tempo di esecuzione.

Se considerassimo, invece, il problema della somma di 102 numeri, con a disposizione 4 processori, potremmo distribuire il carico di lavoro in modo da diminuire il tempo di esecuzione. Se assegnassimo ad ogni processore 25 numeri ed a un solo processore 2 numeri in più tale processore dovrebbe effettuare 26 somme, il tempo di esecuzione in parallelo sarà quindi tempo di esecuzione del processore che impiega il maggior numero di operazioni, per un totale di 26 operazioni; se tuttavia distribuissimo il carico in modo tale che due processori abbiano 26 numeri e 2 processori ne abbiano 25, il tempo di esecuzione in parallelo sarebbe sempre il tempo di esecuzione del processore che impiega il maggior numero di operazioni, per un totale di 25 somme.

E’ evidente come contestualizzando il problema a numeri molto più grandi, bilanciando opportunamente il carico di lavoro si può ridurre notevolmente il tempo di esecuzione.

3.4 Speed-Up, Efficienza e Overhead

Come affermato nel capitolo 1, in un algoritmo sequenziale il tempo di esecuzione equivale al numero di operazioni eseguite. Quindi l’efficienza di un algoritmo sequenziale è data dal numero di operazioni impiegate per eseguire l’algoritmo.

In un algoritmo parallelo, invece, alcune operazioni sono eseguite in parallelo, ovvero concorrentemente; ciò significa che tali operazioni vengono eseguite in uno stesso passo temporale. Il tempo di esecuzione, quindi, non dipenderà esclusivamente dalle operazioni floating point effettuate e dunque l’efficienza non potrà essere misurata attraverso queste ultime.

Per poter definire l'efficienza in un algoritmo parallelo occorre, innanzitutto, definire il concetto di Speed-up. Quest'ultimo misura la riduzione del tempo di esecuzione di un algoritmo su p processore, rispetto al tempo di esecuzione dell'algoritmo su un processore:

$$\mathbf{S(p)} = \frac{T(1)}{T(p)}$$

Lo Speed-up si definisce ideale se è pari al numero di processori p:

$$\mathbf{S(p)_{ideale}} = p$$

Sebbene in linea generale parallelizzando un algoritmo su *p* processori ci si aspetta di ridurre il tempo di esecuzione di *p* volte, ovvero ci si aspetta che *T(1)* sia *p* volte *T(p)*, non sempre ciò avviene. Molto spesso *T(p)* risulta essere più grande di $\frac{T(1)}{p}$ e, quindi, spesso lo Speed-up non è ideale, ma inferiore all'ideale.

Per ciò che è stato detto può essere utile introdurre un ulteriore parametro: l' Overhead totale. Esso misura quanto lo Speed-up differisce da quello ideale:

$$\mathbf{Oh(p)} = (p*T(p)) - T(1) \implies \mathbf{T(p)} = \frac{Oh(p)+T(1)}{p}$$

Definendo lo Speed-up in funzione dell'Overhead otteniamo:

$$\mathbf{S(p)} = \frac{T(1)}{T(p)} = \frac{T(1)*p}{Oh(p)+T(1)}$$

Tuttavia lo Speed-up non basta per poter definire l'efficienza di un algorio parallello: sebbene all'aumentare del numero di processori lo speed-up aumenta e quindi si riduce sempre più il tempo impiegato per eseguire un algoritmo, esso si allontana sempre più dall'ideale, ovvero si allontana dalla miglior riduzione potenziale del tempo impiegato per eseguire un algoritmo.

Per questo motivo in un algoritmo parallelo l'efficienza può essere espressa come il rapporto tra lo Speed-Up e il numero di processori impiegati:

$$\mathbf{E(p)} = \frac{S(p)}{p}$$

$$\mathbf{E(p)_{ideale}} = S(p)_{ideale} / p$$

L'efficienza è un dato fondamentale, ci indica quanto un algoritmo sfrutta il parallelismo. Ci dà una misura di efficacia del parallelismo applicato all'algoritmo.

Tornando al concetto di tempo di esecuzione, esso può essere espresso come la somma del tempo impiegato per eseguire la parte sequenziale dell'algoritmo e del tempo impiegato per eseguire la parte potenzialmente parallelizzabile dell'algoritmo. Possiamo, dunque, scrivere:

$$\mathbf{T(1)} = T_s + T_p$$

$$\mathbf{T(p)} = T_s + T_p/p$$

(in T(p) la parte parallelizzabile è, infatti, eseguita in parallelo da p processori)

Ponendo:

$$\mathbf{T_s} = \quad a$$

$$\mathbf{T_p} = \quad 1-a$$

Otteniamo:

$$\mathbf{T(1)} = \quad 1$$

$$\mathbf{T(p)} = \quad a+(\frac{1-a}{p})$$

Riscrivendo tutto in funzione delle due parti del tempo di esecuzione, otteniamo:

- l'efficienza, l'overhead totale:

$$\mathbf{E(p)} = \quad \frac{S(p)}{p} \quad = \quad \frac{T(1)}{p*T(p)} \quad = \quad \frac{1}{p(a+(\frac{1-a}{p}))} \quad = \quad \frac{1}{a(p-1)+1}$$

$$\mathbf{Oh(p)} = \quad (p*T(p))-T(1) \quad = \quad (p(a+(\frac{1-a}{p})))-1 \quad = \quad a(p-1)$$

- entrambi in funzione l'uno dell'altro:

$$\mathbf{E(p)} = \quad \frac{T(1)}{p*T(p)} \quad = \quad \frac{T(1)}{Oh(p)+T(1)}$$

$$\mathbf{Oh(p)} = \quad (\frac{T(1)}{E(p)})-T(1)$$

- la ***Legge di Ware-Amdahl*** e la ***Legge di Ware-Amdahl generalizzata***:

$$\mathbf{S(p)} = \quad \frac{T(1)}{T(p)} \quad = \quad \frac{1}{a-(\frac{1-a}{p})}$$

$$\mathbf{S(p)} = \quad \frac{1}{\sum_{i=2}^{p}(\frac{ai}{i})} \quad \textit{, con } a_i \textit{ operazioni eseguite in parallelo da i processori}$$

Dalla legge di Ware e da tutto ciò che abbiamo detto precedentemente possiamo affermare che:

1) Fissata la dimensione del problema, all'aumentare del numero di processori, non solo non si raggiunge mai lo speed-up ideale ma le prestazioni degradano
2) Fissato il numero di processori, all'aumentare della dimensione del problema si tende sempre più allo speed-up ideale. Ovviamente non è possibile aumentare la dimensione del problema in maniera indefinita, dal momento che le risorse hardware sono limitate.

3.4.1 Tempo di esecuzione di un algoritmo parallelo su p processori

Per quanto detto nel paragrafo precedente il tempo di esecuzione di un algoritmo parallelo, distribuito su p processori, può essere scisso in 2 componenti:

1. **T_s**: tempo impiegato per eseguire la parte sequenziale dell'algoritmo
2. **T_p / p**: tempo impiegato per eseguire la parte parallelizzabile, in parallelo su p processori

Se consideriamo che il costo di una singola operazione floating point vale *tcalc*, il tempo di esecuzione di un algoritmo parallelo, distribuito su p processori, sarà pari a:

$$\mathbf{T(p)} = \left(Ts + \frac{Tp}{p}\right) tcalc$$

Tuttavia è fondamentale ricordare che un aspetto importante delle operazioni svolte in parallelo risiede nella loro condivisione dei risultati: al tempo di esecuzione va sommato il costo delle comunicazioni necessarie per ottenere un risultato finale. Considerando *tcom* il costo di una singola comunicazione, il tempo di esecuzione sarà pari a:

$$\mathbf{T(p)} = \left(\left(Ts + \frac{Tp}{p}\right) tcalc\right) + \left(\left(Tc\right) tcom\right) \qquad \text{con Tc >= 0}$$

UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

TUTTI GLI ESERCIZI DI

Parallel And Distributed Computing

DELLO STUDENTE **Andrea Capone N97000359**

Indice

Capitolo 1

Tassonomia di Flynn

1.1 26-01-2011 es 6 / 09-01-2018 es 5

Differenza tra architettura MIMD a memoria condivisa e architettura MIMD a memoria distribuita:

Le architetture **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata) sono la classe dei calcolatori in grado di eseguire flussi di istruzioni diverse su diversi tipi di dato. Ne fanno parte i sistemi multiprocessore ovvero quelli costituiti da più CPU.

È possibile distinguere due tipologie di calcolatori MIMD a seconda che la memoria sia distribuita o condivisa tra i diversi processori.

Nel primo caso, uno svantaggio sta nel fatto che i processori devono comunicare tra loro per scambiarsi i dati, nel secondo caso un vantaggio sta nel fatto che non occorre far comunicare i processori ma tuttavia è necessario gestire l'accesso concorrente ai dati.

1.2 22-02-2011 es 6 / 25-01-2016 es 4

Si descrivano i tre tipi di parallelismo studiati, evidenziando le differenze e servendosi di esempi.

Possiamo distinguere 3 tipi di parallelismo:

- **Parallelismo temporale** implementato da architetture **MISD** **M**ultiple **I**nstruction **S**ingle **D**ata, e ne fanno parte i calcolatori che implementano le pipeline.

  Un esempio di pipeline può essere visto all'interno di una ALU: Questa è formata da diversi segmenti che eseguono una operazione elementare e possono lavorare concorrentemente. Ogni segmento lavora su una porzione del dato, passa il risultato al segmento successivo e si prepara a lavorare su un'altra porzione di dato. I segmenti lavoreranno fino a quando tutto il dato non è stato processato.

- **Parallelismo spaziale** implementato da architetture **SIMD** **S**ingle **I**nstruction **S**ingle **D**ata, ne fanno parte i calcolatori dotati di più ALU e una singola CU. Una volta che la CU decide quale operazione effettuare, tutte le ALU eseguiranno concorrentemente quella operazione.
- **Parallelismo asincrono** implementato da architetture **MIMD** **M**ultiple **I**nstruction **M**ultiple **D**ata, sono la classe dei sistemi multiprocessore, ovvero quelli costituiti da più CPU: ogni CPU ha una propria ALU, per cui ogni CPU è in grado di eseguire flussi di istruzioni diverse su diversi tipi di dato in parallelo per ogni ALU.

1.3 16-12-2011 es 4 / 23-12-2020 es 4

Si consideri un'architettura parallela di tipo MISD

- Indicare il significato ed il tipo di parallelismo che è possibile implementare per tale classe;
- Specificare la differenza tra questo tipo di architettura e quella di tipo SIMD.

Le architetture **MISD** **M**ultiple **I**nstruction **S**ingle **D**ata, sono delle architetture in grado di eseguire un flusso di istruzioni su un singolo dato.

Ne fanno parte i calcolatori che implementano le pipeline. È possibile quindi realizzare un parallelismo di tipo **Temporale**.

La differenza con i calcolatori **SIMD** è che queste ultime sono dotate di una singola CU e diverse ALU che agiscono su diversi dati, per cui si può implementare un parallelismo **Temporale** se si considera una singola ALU e in generale, si implementa il parallelismo di tipo **Spaziale** nel quale, la CU decide l'istruzione che deve essere eseguita, e tutte le ALU la eseguiranno in parallelo.

1.4 04-02-2019 es 5

Si consideri la classificazione di Flynn relativamente alle architetture parallele. Descrivere l'architettura SIMD e citare almeno un tipo di parallelismo che questa realizza.

L'architettura di tipo **SIMD** **S**ingle **I**nstruction **M**ultiple **D**ata è un tipo di architettura nella quale sono presenti calcolatori dotati di una singola CU collegata a diverse ALU che agiscono su dati diversi

Un tipo di parallelismo che possono implementare è il parallelismo **Spaziale**, nel quale la CU deciderà l'istruzione da eseguire e le ALU la eseguiranno in parallelo, oppure il parallelismo temporale (pipeline) se si considera una singola ALU.

Capitolo 2

Routine MPI (MESSAGE PASSING INTERFACE)

2.1 26-01-2011 es 4

Si consideri la seguente routine MPI:

```
1
2 MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
     comm);
```

- cosa è possibile effettuare utilizzando tale routine?
- descrivere in dettaglio i parametri.

Questa routine permette al processore root di inviare a tutti i processori del communicator comm i primi count elementi di tipo datatype salvati in buffer.

I parametri di questa routine sono:

1. *buffer: il buffer in cui è salvato il dato da inviare;
2. count: numero di elementi da inviare;
3. datatype: il tipo di dato da inviare;
4. root: il processore che effettua il broadcast;
5. comm: il communicator.

2.2 22-02-2011 es 4 / 04-02-2019 es 3

Si consideri la seguente routine della libreria MPI:

```
1
2 MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int
     periods[], int reorder, MPI_Comm * comm_cart)
```

- cosa è possibile effettuare utilizzando tale routine?
- descrivere in dettaglio i parametri.

Questa routine è un'operazione collettiva che restituisce un nuovo communicator comm_cart in cui i processi sono organizzati in una griglia di dimensioni ndims.

L'i-esima dimensione ha lunghezza dims[i].

Se periods[i]=1, la i-esima dimensione della griglia è periodica, non lo è se periods[i]=0.

I parametri in input sono i seguenti:

1. **comm_old**: communicator di input;
2. **dim**: numero di dimensioni della griglia;
3. ***ndim**: vettore di dimensione dim contenente le lunghezze di ciascuna dimensione;
4. ***period**: vettore di dimensione dim contenente la periodicità di ciascuna dimensione;
5. **reorder**: permesso di riordinare i processori (1=si; 0=no);
6. **comm_cart**: communicator di output associato alla griglia.

2.3 16-12-2011 es 3 / 23-12-2020 es 3

Si consideri la seguente routine della libreria MPI:

```

MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm)
```

- cosa è possibile effettuare utilizzando tale routine?
- descrivere in dettaglio i parametri.

Questa routine permette di effettuare l'operazione collettiva op (ad esempio la somma o la media) tra tutti i processori del communicator comm dei primi count elementi del buffer *sendbuf di tipo datatype ed inviare il risultato al processore con rango root nella variabile *recvbuff

i parametri sono:

1. *sendbuff: indirizzo degli operandi;
2. *recvbuff: indirizzo del risultato;
3. count: numero di elementi da considerare;
4. datatype: tipo di dato;

5. op: tipo di operazione;
6. root: processore che riceve il risultato;
7. comm: communicator.

2.4 09-01-2018 es 3

Si consideri la seguente routine della libreria MPI:

```
1
2 MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- cosa è possibile effettuare utilizzando tale routine?
- descrivere in dettaglio i parametri.

Questa routine è una funzione di comunicazione che permette al processore root di ricevere dai processori del communicator comm sendcount elementi di tipo datatype del buffer *sendbuf.

Il processore root riceverà i dati in ordine partendo dal processore con rango più basso fino a quello più alto.

i parametri sono:

1. *sendbuf: buffer contenente il dato da inviare;
2. sendcount: numero di elementi da inviare per ogni singolo processore;
3. sendtype: tipo di dato inviato;
4. *recvbuff: indirizzo dove salvare i dati ricevuti;
5. recvtype: tipo di dato ricevuto;
6. root: processore che riceve i dati;
7. comm: communicator.

Capitolo 3

OMP (OPEN MULTI-PROCESSING)

3.1 25-01-2016 es 3 / 09-01-2018 es 4 / 04-02-2019 es 4

Illustrare lo schema di fork join in OPEN-MP e mostrare come si apre una regione parallela.

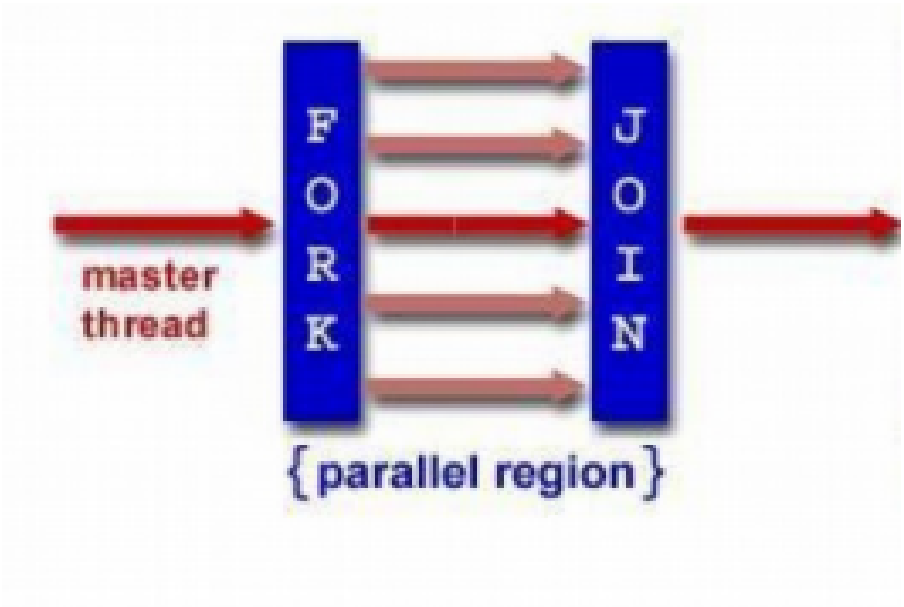
in Open-MP, si utilizza il modello di esecuzione parallela di tipo fork join:

L'esecuzione parte dal thread master che esegue la parte sequenziale.

Nel momento in cui si arriva alla regione parallela, viene creato un team di thread (fork) che eseguiranno in maniera concorrente la regione.

Quando ogni thread del team ha eseguito la regione parallela, torna il controllo al thread master (join) che prosegue con l'esecuzione sequenziale.

di seguito lo schema del fork-join:



Per aprire una regione parallela in OPEN-MP è sufficiente utilizzare il costrutto **#Pragma omp** e specificare direttive e clausole per gestire l'accesso concorrente alla memoria da parte dei thread.

3.2 22-02-2011 es 5

Si consideri l'API Open-MP.

- Si descrivano i costrutti work sharing;
- Si descrivano tre clausole che siano applicabili ognuna ad almeno un costrutto descritto al punto precedente.

Quando si apre una regione parallela e si aggiunge la clausola parallel (#Pragma omp parallel) è possibile affiancare dei costrutti detti work sharing.

Ci sono 3 tipi di costrutti work sharing e sono:

1. for, il quale permette di far eseguire la regione parallela dividendo le istruzioni tra tutti i thread del team. Si utilizza for e non while perché con il for si conosce già il numero di iterazioni.
2. sections, che permette di dividere la regione parallela in sezioni e ciascuna di esse viene eseguita da un thread del team;
3. single, che forza ad un singolo thread casuale l'esecuzione della regione parallela mentre gli altri attendono che termini.

Le direttive possono essere arricchite dall'utilizzo di clausole, per esempio abbiamo:

- private(list), che rende per ciascun thread del team le variabili di list private, nel senso che ognuno avrà una sua copia locale;
- reduction(op:operand), che permette a ciascun thread del di applicare l'operazione op sulla variabile operand. L'accesso a questa variabile è gestita concorrentemente;
- shared(list), che rende per ciascun thread le variabili di list condivise e quindi verrà gestito l'accesso concorrente a queste ultime.

Capitolo 4

Enunciati

4.1 22-02-2011 es 3

Si enuncino le definizioni di:

- Efficienza;
- Overhead totale,

e si esprima la relazione tra essi, riscrivendo la prima in funzione del secondo.

Per poter dare la definizione di efficienza, è necessario definire innanzitutto lo speed up, che misura la riduzione del tempo di esecuzione di un algoritmo su p processori rispetto al tempo di esecuzione su un processore:
$S(P) = \frac{T(1)}{T(P)}$, inoltre lo speedup si dice ideale se $S(P) = P$.
L'efficienza, misura quanto un algoritmo sfrutta il parallelismo ed è espressa come rapporto fra speed up ed il numero di processori:
$E(P) = \frac{S(P)}{P} \Rightarrow E(P) = \frac{T(1)}{P*T(P)}$.
Infine quest'ultima si dice ideale se uguale ad 1: $E(P) = 1$.

L'overhead totale, misura invece quanto lo speedup differisce dallo speedup ideale: $Oh(P) = (P * T(P)) - T(1)$.

Per cui: $T(P) = \frac{Oh(P)+T(1)}{P}$
Possiamo scrivere allora lo speedup in funzione dell'overhead totale:
$S(P) = \frac{T(1)}{T(P)} \Rightarrow S(P) = \frac{T(1)*P}{Oh(P)+T(1)}$
e quindi l'efficienza:
$E(P) = \frac{S(P)}{P} \Rightarrow E(P) = \frac{T(1)}{P*T(P)} \Rightarrow E(P) = \frac{T(1)}{P*\frac{Oh(P)+T(1)}{P}}$
$\Rightarrow E(P) = \frac{T(1)}{Oh(P)+T(1)}$

4.2 26-01-2011 es 2 / 25-01-2016 es 1

Si enunci la legge di Ware generalizzata, descrivendo nel dettaglio tutti i parametri.

Per poter enunciare la legge di Ware bisogna prima esprimere il concetto di tempo di esecuzione come la somma tra il tempo impiegato per eseguire la parte sequenziale e quello per eseguire la parte parallela:

$T(1) = T_S + T_P$ e $T(P) = T_S + \frac{T_P}{P}$.

Ponendo $T_S = a$ e $T_P = 1 - a$ si ottiene:

$T(1) = 1$ e $T(P) = a + (\frac{1-a}{P})$.

Legge di Ware:

$S(P) = \frac{T(1)}{T(P)} \Rightarrow S(P) = \frac{1}{a+(\frac{1-a}{P})}$.

Di conseguenza possiamo scrivere Oh(P) ed E(P):

$Oh(P) = P * T(P) - T(1) \Rightarrow Oh(P) = P * (a + (\frac{1-a}{P})) - 1$
$\Rightarrow Oh(P) = a(p-1)$.

$E(P) = \frac{T(1)}{P*T(P)} \Rightarrow E(P) = \frac{1}{P*(a+(\frac{1-a}{P}))} \Rightarrow E(P) = \frac{1}{P*a+1-a}$
$\Rightarrow E(P) = \frac{1}{a(p-1)+1}$.

Legge di Ware generalizzata:

$$S(P) = \sum_{k=2}^{p} \frac{a_K}{k},$$

con a_k operazioni eseguite in parallelo da k processori.