Parallel Lab

OpenMP

I dati sono condivisi in maniera globale tra le thread, possono poi esserci i dati **private** che appartengono alle singole thread che li creano, si usa **#pragma omp** per le direttive.

- #pragma omp parallel{}: crea una regione di N threads(solitamente numero di core) dove ciò che c'è nella regione viene eseguito da tutte le thread.
    - specificando **num_threads(n)** posso dire quante thread voglio manualmente
- **#pragma omp parallel for**: è un work-sharing construct, deve stare in una regione, per accorciare si può usare **parallel** prima di **for**, ciò che fa è dividere il lavoro equalmente tra le N thread disponibili
- **#pragma omp for ordered** con dentro **#pragma omp ordered:** permette di eseguire un blocco di codice dentro un loop parallelizzato in maniera sequenziale
- **#pragma omp parallel shared(var1, var2, ...)**: serve a indicare che le variabili tra parentesi sono condivise tra tutti i threads, dato che openmp si basa sul modello di programmazione shared memory, molte variabili sono shared di default.
- **#pragma omp parallel private(var1, var2, ...)**: serve a indicare che le variabili tra parentesi sono private, ossia ogni thread ne ha una propria copia, **sono undefined all'ingresso e all'uscita della regione.**
    - **#pragma omp parallel firstprivate(var1, var2, ...):** tutte le variabili private tra parentesi, **dentro** la regione parallela, prendono il valore che ha la variabile prima della regione parallela
    - **#pragma omp parallel lastprivate(var1, var2, ...):** tutte le variabili tra parentesi, **fuori** dalla regione parallela, assumono il valore che avevano nell'ultimo istante prima di uscire dalla regione parallela
- funzioni utili a runtime:
    - omp_get_max_threads() → max thread creabili
    - omp_get_num_threads() → numero di thread presenti
    - omp_get_thread_num() → numero della thread che chiama funzione
    - omp_set_num_threads() → imposta il numero delle thread delle prossime parallel region

SCHEDULING

- **#pragma omp for schedule(static)**: è il for che divide equalmente le iterazioni facendo Lunghezza/N
- se si specifica anche **chunk_size**, il for viene diviso in blocchi di chunk size assegnati alla round robin ai thread
- **#pragma omp for schedule(dynamic):** usa la coda di lavoro interna per dare un blocco di lavoro a ogni thread, quando un thread ha finito prende il prossimo pezzo di lavoro dalla coda, anche qui si può assegnare la **chunk_size**, introduce ulteriore **overhead,** utile quando usato con ordered o in loop dove le iterazioni potrebbero richiedere tempi diversi
- **#pragma omp for schedule(guided):** simile al dinamico ma data una chunk_size questa va a diminuire man mano che passano le iterazioni per fittare il load balancing
- **#pragma omp for schedule(auto):** si delega la decisione sullo scheduling al compilatore
- **#pragma omp for schedule(runtime):** la decisione sullo scheduling è decisa a runtime

BLOCCHI

- **#pragma omp single**: forza un blocco di codice a essere eseguito da un solo thread, tutti gli altri attendono con una barriera l'esecuzione di questo blocco da parte del singolo thread
- **#pragma omp master**: stessa cosa del single ma il thread che esegue il blocco è il thread master
- **#pragma omp parallel sections** con all'interno **#pragma omp section:** serve per indicare porzioni di codice che possono runnare in parallelo o meno. Ciò che c'è all'interno di una singola **section** invece deve essere runnato sequenzialmente
- **#pragma omp task:** specifica una task, probabilmente verrà usata per algoritmini ricorsivi quindi è bene tenerla in un **omp single** in quanto altrimenti verrà creata N volte al posto che una

Crea

SINCRONIZZAZIONE

- **#pragma omp taskwait:** lavora similmente a una barriera, il flow di esecuzione di un thread che ha creato delle task viene stoppato fino a che tutte le task non hanno finito di eseguire
- **#pragma omp barrier:** serve a specificare che tutti i threads devono bloccarsi a quella data barriera fintantochè tutti i thread non l'hanno raggiunta, facili e banali da utilizzare ma costose e potrebbero non scalare con molti processori.
- **#pragma omp (sections | for | single) nowait**: serve per specificare di non utilizzare la barriera di default alla fine del (sections | for | single)
- **#pragma omp critical:** specifica che un certo blocco di codice va eseguito un thread alla volta, non ci sono barriere implicite ne prima ne dopo. Si specifica name critical (critical con stesso nome stramo associate)
- **#pragma omp atomic:** utilizzata per specificare che una variabile sia aggiornata atomicamente, utilizzabile solo dove la variabile viene aggiornata con una semplice espressione.
- **lock**
  - omp_init_lock(): inizializza il lock
  - omp_set_lock(): prova a settare il lock, altrimenti attende e poi setta
  - omp_unset_lock(): unsetta il lock
  - omp_test_lock(): testa il lock, 0 se è settato, 1 altrimenti non ferma esecuzione
  - omp_destroy_lock(): distrugge il lock

ALTRO

- **#pragma omp flush(var1, var2, ...):** utilizzato se vogliamo essere sicuri che il valore visto da un thread sia uguale a quello visto dagli altri
- **#pragma omp parallel reduction(op, var1, var2, ...):** permette di accumulare una variabile condivisa senza atomic
- **#pragma omp simd:** fa la roba strana la della conversione a fp per velocizzare

CUDA

- **__global__ void MyKernel(){}**: serve per definire un kernel
- **MyKernel<<>>()**: serve per eseguire il kernel
  - dove **<<dim3 dimGrid(x,y,z), dim3 dimBlock(x,y,z)>>** (num blocchi, num thread x Blocco)
- **threadIdx.x/y/z, blockIdx.x/y/z** per l'identificatore della thread
  - stessa cosa per blockDim e gridDim
- molto spesso troveremo **globalThreadIDX = blockIdx.x * blockDim.x + threadIdx.x**
  - ovviamente va fatto per tutte le dimensioni se siamo su 2+D
- **cudaMalloc(indirizzo puntatore src, dimensione da allocare):** utilizzato per allocare memoria nel device
- **cudaMemCopy(dest, src, dimensione bytes, direzione):** usato per trasferire dati dall'host allo spazio allocato nel device e per trasferire risultato dal device all'host
  - direzione può essere: **cudaMemcpyHostToDevice** o **cudaMemcpyDeviceToHost**
- **CudaFree(puntatore)**
- **SAFE_CALL(chiamata a funzione):** per controllare errori durante l'esecuzione della funzione, non si usa con il kernel
- **CHECK_CUDA_ERROR**: usata dopo il kernel per trovare errori
- _synchthreads(): barriera per i thread in un blocco, a livello di warp la sincronizzazione esplicita non serve perchè si esegue in stile simd e quindi c'è già sincronizzazione implicita
- **cudaDeviceSynchronize()**: usato per fare sincronizzazione tra host e device, il kernel è asincrono non bloccante, quindi l'host continuerebbe ad eseguire(nei nostri codici non lo vediamo, se apriamo però CHECK_CUDA_ERROR nella libreria che abbiamo troviamo che viene fatta una chiamata a quest'ultimo)
- **gpu occupancy:** rateo di warp attivi per multiprocessore / massimo numero possibile di warp attivi. Tra ciò che limita l'occupancy abbiamo: utilizzo dei registri e della shared memory e dimensione del blocco. Importante è ricordare che

alta occupazione non implica alte performance

MPI

Utilizza i **comunicatori** per definire quali collezioni di processi possono comunicare tra di loro, più precisamente due processi possono comunicare se hanno un comunicatore in comune, in un comunicatore un processo ha un id detto **rank**.

- **MPI::COMM_WORLD:** comunicatore che unisce tutti i processi
- **MPI::Init(&argc, &argv):** inizializza l'ambiente MPI
- **MPI::COMM::Get_Rank():** restituisce il rank del processo nel COMM
- **MPI::COMM::Get_size()**: numero di processi nel COMM
- **MPI::Get_processor_name(char* name, int& resultlength)**: da nome e lunghezza del nome del processore
- **MPI::Finalize()**: termina l'ambiente mpi
- **MPI::COMM::Abort(int errorcode)**: termina tutti i processi associati con COMM tutto il COMMWORLD
- bool MPI::Is_initialized(): true se è stato chiamato l'init
- double MPI::Wtime(): tempo passato in secondi
- double MPI::Wtick(): bo

La comunicazione point to point di mpi coinvolge solo due MPI task, una che fa la send e una che fa la receive, possono essere bloccanti o meno.

C'è un system buffer, area dove transitano i dati gestita da mpi e un application buffer gestita invece dallo user.

Una send bloccante ritorna solo dopo che è safe modificare l'application buffer.

Una receive bloccante ritorna solo dopo che i dati sono arrivati e pronti da utilizzare.

Bisogna prestare attenzione che se abbiamo due sender e un receiver, solo uno dei due send verrà preso dal receive, l'altro no.

- Status.Get_source(): id del processore che manda il messaggio
- Status.Get_tag(): tag del messaggio
- int Status.Get_count(MPI::Datatype& datatype): numero di elementi ricevuti

COMUNICAZIONI BLOCCANTI PTP

- **MPI::COMM::Send(void* buf, int count, MPI::Datatype& datatype, int dest, int tag):**
  send bloccante, ritorna dopo che l'application buffer del sending task è libero
- **MPI::COMM::Recv(void* buf, int count, MPI::Datatype& datatype, int source, int tag, [&status]):**
  receive bloccante, riceve un messaggio e blocca fino a che i dati non sono presenti nell'application buffer
- **MPI::COMM::Sendrecv(void* sendbuf, int sendcount, MPI::Datatype& senddatatype, int dest, int sendtag, void* recvbuffer, int recvcont, MPI::Datatype& recvtype, int source, int recvtag, [&status]):**
  send-receive bloccante
- **MPI::COMM::Ssend(void* buf, int count, MPI::Datatype& datatype, int dest, int tag):**
  send sincrona bloccante, ritorna quando l'application buffer del sender è libero e il destination ha iniziato a ricevere
- **MPI::COMM::Rsend(void* buf, int count, MPI::Datatype& datatype, int source, int tag):**
  - continua quando solo se la recive corrispondente è avvenuta
- si può fare self mesagging utilizzando il comunicatore predefinito MPI_COMM_SELF

COMUNICAZIONI NON BLOCCANTI PTP

Utili per fare overlap e guadagnare performance

- **MPI::COMM::Isend(void* buf, int count,**
  MPI::Datatype& datatype, int dest, int tag):
  send non bloccante, non si dovrebbe modificare l'application buffer fino a una chiamata di wait o test che è stata completata
- **MPI::COMM::Irecv(void* buf, int count,**
  MPI::Datatype& datatype, int source, int tag):
  receive non bloccante, il programma deve chiamare wait o test per capire quando la receive termina e il messaggio e disponibile nell'application buffer
- **MPI::COMM::Issend(void* sendbuf, int sendcount,**
  MPI::Datatype& datatype, int dest, int tag):
  send sincrona non bloccante, qui wait o test indicano quando il processo destinazione ha ricevuto il messaggio
- **Request.Wait([&status]):** blocca il processamento fino a che una specificata send o receive non bloccante completano
- **Request.Test([&status]):** controlla lo stato di una send o receive non bloccante
- HANDLING DI ERRORI VARIO

COMUNICAZIONI COLLETTIVE

Coinvolgono tutti i processi nello scope di un comunicatore, sono tutte **bloccanti**, ne esistono di vario tipo:

- **sincronizzazione:** i processi aspettano fino a che tutti i membri del gruppo hanno raggiunto il punto di sincronizzazione
  - **MPI::COMM::Barrier():** crea una barriera in un gruppo
- **data movement:** broadcast, scatter, gather
  - **MPI::COMM::Bcast(void* buffer, int count,**
    MPI::Datatype& datatype, int root):
    il processo che ha root come rank manda il messaggio a tutti i processi
  - **MPI::COMM::Scatter(void* sendbuf, int sendcount,**
    MPI::Datatype& sendtype, void* recvbuf,
    int recvcount, MPI::Datatype& recvtype, int root):
    distribuisce messaggi diversi dalla task che ha ROOT come rank a tutti gli altri, SENDBUF è un array, **sendcount/recvcount** è solitamente pari e numero di elementi nell'array diviso numero di processi(se sendbuf ha 10 elementi e ho 5 processi, sendcount e recvcount saranno 2, ossia 2 pezzi di array per processo)
  - **MPI::COMM::Gather(void* sendbuf, int sendcount,**
    MPI::Datatype& sendtype, void* recvbuf,
    int recvcount, MPI::Datatype& recvtype, int root):
    la task che ha ROOT come rank raccoglie vari messaggi dal gruppo, i messaggi ricevuti sono ordinati sul rank, vale lo stesso discorso per recvcount e sendcount
- **riduzioni**
  - **MPI::COMM::Reduce(void* sendbuf, void* recvbuf, int count,**
    MPI::Datatype& datatype, MPI:OP, int root):
    applica un'operazione di riduzione e mette il risultato nel buffer del task che ha ROOT come rank
- **metodi intermedi**
  - **MPI::Comm::Allgather(void* sendbuf, int sendcount,MPI::Datatype& sendtype,void* recvbuf, int recvcount, MPI::Datatype& recvtype)** ogni task fa il gather dei dati delle altre task e poi distribuisce i dati nel proprio buffer alle altre task nel comunicatore
  - **MPI::Comm::Alltoall(void* sendbuf, int sendcount,MPI::Datatype& sendtype,void* recvbuf, int recvcount,MPI::Datatype& recvtype)** tutte le task di un comunicatore fanno lo scatter
  - **MPI::Comm::Allreduce(void* sendbuf, int sendcount,MPI::Datatype& sendtype,void* recvbuf, int recvcount,MPI::Datatype& recvtype)** tutte le task fanno una reduce con i dati mandati dalle altre task del comunicatore

Se non utilizziamo datatypes di MPI come ad esempio MPI::INT, possiamo comunque definire dei nostri datatype che in MPI sono chiamati derived datatypes, 4 passi:

- costruire il datatype → decido se **contiguos, vector, indexed o struct**
- allocare il datatype
- usare il datatype
- deallocare il datatype

1. Costruzione del datatype
   - **Newtype Oldtype.Create_contiguous(int count)**
     Crea un nuovo tipo di dato che è una sequenza di `count` elementi consecutivi di tipo `Oldtype`.
   - **Newtype Oldtype.Create_vector(int count, int blocklength, int stride)**
     Crea un nuovo tipo di dato che rappresenta una serie di blocchi regolarmente distanziati. Ogni blocco contiene `blocklength` elementi di tipo `Oldtype`, e i blocchi sono separati da una distanza `stride` (misurata in termini di numero di elementi di `Oldtype`).
   - **Newtype Oldtype::Create_indexed(int count, int array_of_blocklengths[],int array_of_displacements[])**
     Crea un nuovo tipo di dato che ha `count` blocchi di dimensioni variabili. Ogni blocco ha una lunghezza specificata in `array_of_blocklengths` e inizia ad un offset specificato in `array_of_displacements` (misurato in unità di `Oldtype`).
   - **MPI::Datatype MPI::Datatype::Create_struct(**
     int count,int array_of_blocklengths[], MPI::Aint array_of_displacements[], MPI::Datatype array_of_types[])
     crea l'equivalente di una struttura C/C++ mappando i tipi corrispondenti
   - **MPI::Datatype.Commit()** fa il commit del nuovo datatype al sistema, **obbligatorio**
   - **MPI::Datatype.free()** fa la deallocazione del datatype specificato

- **metodi avanzati** fanno tutti la stessa cosa dei corrispettivi normali, l'unica differenza è che adesso al posto di mandare un elemento dell'array per task nel comunicatore posso specificare il numero di elementi
- **MPI::Comm::Scatterv(void* sendbuf, int array_of_blocklengths[], //this is sendcount in scatter()**
  int array_of_displacements[], MPI::Datatype& sendtype,
  void* recvbuf, int recvcount, MPI::Datatype& recvtype, int root)
- **MPI::Comm::Gatherv(void* sendbuf, int sendcount MPI::Datatype& sendtype, void* recvbuf, int array_of_blocklengths[], //this is recvcount in gather() int array_of_displacements[], MPI::Datatype& recvtype, int root)**
- **MPI::COMM::Allgatherv(void* sendbuf, int sendcount, MPI::Datatype& sendtype, void* recvbuf, int recvcounts[],int displs[],MPI::Datatype& recvtype)**
- **MPI::COMM::Alltoallv(void* sendbuf, int sendcounts[], int displs[],**
  MPI::Datatype& sendtype, void* recvbuf, int recvcounts[], int rdispls[],
  MPI::Datatype& recvtype)
  tutti mandano e ricevono da tutti

Size diverse

Le prestazioni di un sistema MPI dipendono da diversi fattori che possono essere raggruppati in tre categorie:

- la piattaforma/architettura
- le caratteristiche dell'applicazione
- l'implementazione stessa di MPI.

Per quanto riguarda la **piattaforma** su cui viene eseguito MPI, i principali elementi da considerare sono la velocità della CPU e il numero di core disponibili, la configurazione della memoria e della cache, le caratteristiche della rete.

Dal lato **applicativo**, sono fondamentali l'efficienza e la scalabilità dell'algoritmo utilizzato, fattori come le operazioni di I/O, la dimensione dei messaggi, il bilanciamento del carico tra i processori, i pattern di utilizzo della memoria e il tipo di routine MPI (bloccanti, non bloccanti o collettive) possono avere un impatto notevole.

Infine, le **implementazioni specifiche di MPI** influiscono sulle prestazioni attraverso aspetti come la bufferizzazione dei messaggi, ossia la gestione dello spazio di memoria usato per immagazzinare i dati tra un'operazione di invio e la ricezione corrispondente. Esistono anche diversi **protocolli di passaggio dei messaggi**, come il protocollo **Eager**, che permette di completare un'operazione di invio senza aspettare l'ack del ricevente, e il protocollo **Rendezvous**, che richiede invece una conferma per completare l'invio. La sincronizzazione tra mittente e destinatario può avvenire attraverso il **polling o gli interrupt.**

In conclusione, MPI offre un controllo esplicito sulla comunicazione, permettendo di ottenere un'alta efficienza grazie alla sovrapposizione tra calcolo e comunicazione. Inoltre, MPI scala bene su un numero molto elevato di processori, è portabile e le sue implementazioni attuali sono ottimizzate ed efficienti. Tuttavia, lo sviluppo delle applicazioni con MPI è complesso e richiede molto tempo, poiché sono necessarie modifiche estese al codice seriale. Implementare un bilanciamento dinamico del carico è inoltre difficile.

MPI è particolarmente efficace per problemi a grana grossa, in cui il problema può essere scomposto in sottoproblemi relativamente indipendenti e la comunicazione tra i task è minima. Al contrario, risulta meno efficiente per problemi a grana fine, dove i costi di comunicazione tendono a dominare.