1 MPI

1.1 Inizializzazione MPI con comunicatore

```
int r = MPI_Init(NULL, NULL);
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

1.2 Chiusura MPI

```
MPI_Finalize();
```

1.3 Lancio errore

```
MPI_Abort(MPI_COMM_WORLD, r);
```

1.4 Bloccanti

```
int MPI_Send(
    void* msg_buf_p,          // Puntatore ai dati
    int msg_size,             // Numero di elementi nel messaggio
    MPI_Datatype msg_type,    // Tipo di dato nel messaggio
    int dest,                 // Rank processo a cui inviare
    int tag,
    MPI_Comm communicator     // Comunicatore
);

int MPI_Recv(
    void* msg_buf_p,          // Puntatore ai dati
    int buf_size,
    MPI_Datatype buf_type,
    int source,               // Rank processo da cui riceve
    int tag,
    MPI_Comm communicator,
    MPI_Status* status_p      // Utilizziamo MPI_STATUS_IGNORE
);
```

1.5 Non bloccanti

- `Isend`
- `Irecv`

1.6 Tag

- `MPI_ANY_SOURCE` è possibile ricevere da chiunque
- `MPI_ANY_TAG` è possibile ricevere msg con qualsiasi tag

1.7 Tipi di dato

- MPI_CHAR
- MPI_SIGNED_CHAR
- MPI_UNSIGNED_CHAR
- MPI_BYTE
- MPI_WCHAR
- MPI_SHORT
- MPI_UNSIGNED_SHORT
- MPI_INT
- MPI_UNSIGNED
- MPI_LONG
- MPI_UNSIGNED_LONG
- MPI_LONG_LONG_INT
- MPI_UNSIGNED_LONG_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE

1.8 Comunicazione tra processi

Ogni processo utilizza due buffer invio e ricezione per scambiarsi dati:

```
MPI_Alltoall(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT, MPI_COMM_WORLD);
```

1.9 Inizializzo richiesta

```
MPI_Request request_send_r, request_send_l;
```

Da inserire dopo le `send` e `recv` per bloccare il processo affinché non vengono completate:

```
MPI_Wait(&request_send_l, MPI_STATUS_IGNORE);
MPI_Wait(&request_send_r, MPI_STATUS_IGNORE);
```

1.10 Richieste inizializzate mediante lista

```
MPI_Request lista_richieste[4];
```

Attende che tutte le `recv` e `send` della lista vengano completate:

```
MPI_Waitall(4, lista_richieste, MPI_STATUS_IGNORE);
```

1.11 Broadcast

Broadcast dei valori della variabile `a` a tutti i processi:

```
MPI_Bcast(a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

1.12 Reduce

Ogni processo prende l'indirizzo della variabile calcolata e la somma a `int_totale`:

```
MPI_Reduce(&int_locale, &int_totale, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
```

1.12.1 Operazioni di riduzione

- `MPI_MAX` Massimo
- `MPI_MIN` Minimo
- `MPI_SUM` Somma
- `MPI_PROD` Prodotto
- `MPI_LAND` AND logico
- `MPI_LOR` OR logico
- `MPI_LXOR` XOR logico
- `MPI_BAND` Bitwise AND
- `MPI_BOR` Bitwise OR
- `MPI_BXOR` Bitwise XOR

1.13 Calcolo tempo esecuzione

Ritornano dei `double`:

```
inizio = MPI_Wtime();
/*    code    */
fine = MPI_Wtime();
```

1.14 Scatter

Root invia il vettore spezzato in parti uguali:

```
1 MPI_Scatter(a, len_vect / size, MPI_INT, MPI_IN_PLACE, len_vect / size
      , MPI_INT, 0, MPI_COMM_WORLD);
```

Ogni processo dovrà farlo per ottenere il vettore `a`:

```
1 MPI_Scatter(NULL, len_vect / size, MPI_INT, a, len_vect / size,
      MPI_INT, 0, MPI_COMM_WORLD);
```

1.15 Gather

Ogni processo invia il proprio pezzo di vettore `c` al vettore finale:

```
1 MPI_Gather(c, len_vect / size, MPI_INT, finale, len_vect / size,
      MPI_INT, 0, MPI_COMM_WORLD);
```

1.16 Broadcast del vettore

Broadcast del vettore `x` a tutti i processi:

```
MPI_Bcast(x, colonne, MPI_INT, 0, MPI_COMM_WORLD);
```

1.17 Strutture

```
MPI_Datatype t;
int block_lengths[3] = {1, 1, 1}; // Numero di elementi per tipo
MPI_Aint displacements[3] = {0, 16, 24}; // Offset di memoria
MPI_Datatype types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

Genera struttura:

```
MPI_Type_create_struct(3, block_lengths, displacements, types, &t);
```

Necessario per essere ottimizzato per le comunicazioni:

```
MPI_Type_commit(&t);
```

Libera memoria struttura:

```
MPI_Type_free(&t);
```

2 PTHREAD

2.1 Creazione di un Thread

```
int pthread_create(
    pthread_t *thread,                  // Puntatore alla struttura che identifica il thread
    const pthread_attr_t *attr,         // NULL o attributi del thread
    void *(*start_routine) (void *),    // Puntatore alla funzione che il thread eseguira'
    void *arg                           // Argomento della funzione
);
```

Esempio di utilizzo:

`pthread_create(NULL, NULL, my_function, (void*)10);`

Questa chiamata crea un nuovo thread che esegue `my_function` con l'argomento `10`.

2.2 Attesa della Terminazione di un Thread

`int pthread_join(pthread_t thread, void **value_ptr);`

Questa funzione fa attendere il thread chiamante fino alla terminazione del thread specificato.

2.3 Identificazione del Thread Corrente

`pthread_t pthread_self(void);`

Restituisce l'ID del thread chiamante.

2.4 Confronto di ID di Thread

`int pthread_equal(pthread_t t1, pthread_t t2);`

Confronta gli ID di due thread e restituisce un valore diverso da zero se sono uguali.

2.5 Semafori

2.5.1 Inizializzazione del Semaforo

`int sem_init(sem_t *sem, int pshared, unsigned int value);`

Inizializza un semaforo con un valore iniziale.

2.5.2 Attesa su un Semaforo

`int sem_wait(sem_t *sem);`

Decrementa il semaforo. Se il valore è 0, il thread viene bloccato fino a quando il semaforo non viene incrementato.

2.5.3 Segnalazione su un Semaforo

```
int sem_post(sem_t *sem);
```

Incrementa il semaforo e, se ci sono thread in attesa, ne sblocca uno.

2.5.4 Ottenere il Valore del Semaforo

```
int sem_getvalue(sem_t *sem, int *sval);
```

Recupera il valore corrente del semaforo.

2.5.5 Distruzione del Semaforo

```
int sem_destroy(sem_t *sem);
```

Rimuove il semaforo e libera le risorse.

2.6 RWLock (Read-Write Lock)

2.6.1 Inizializzazione del RWLock

```
1 int pthread_rwlock_init(pthread_rwlock_t* rwlock, const
      pthread_rwlockattr_t* attr);
```

Inizializza un read-write lock.

2.6.2 Bloccare in Modalità Lettura

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock);
```

Blocca il rwlock per letture condivise.

2.6.3 Bloccare in Modalità Scrittura

```
int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock);
```

Blocca il rwlock per scritture esclusive.

2.6.4 Sbloccare il RWLock

```
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock);
```

Sblocca il rwlock.

2.6.5 Distruzione del RWLock

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock);
```

Distrugge il rwlock e libera le risorse.

2.7 Mutex

2.7.1 Inizializzazione del Mutex

```
1 int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *attr);
```

Inizializza un mutex.

2.7.2 Bloccare il Mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Blocca il mutex. Se il mutex è già bloccato, il thread chiamante viene messo in attesa.

2.7.3 Sbloccare il Mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Sblocca il mutex.

2.7.4 Distruzione del Mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Rimuove il mutex e libera le risorse.

2.7.5 Esempio di Utilizzo del Mutex

```
pthread_mutex_lock(&mutex);
sum += 4 * my_sum;
pthread_mutex_unlock(&mutex);
```

Questo esempio dimostra come proteggere una sezione critica utilizzando un mutex.

3 OpenMPI

3.1 Esecuzione della funzione `Hello()` in parallelo con `num_thread` thread

```
#pragma omp parallel num_threads(num_thread)
Hello();
```

Oppure, puoi impostare il numero di thread in questo modo:

```
omp_set_num_threads(int num_threads);
```

3.2 Clausole di Condivisione delle Variabili

Quando aggiungi una pragma, puoi specificare come le variabili sono condivise tra i thread:

- **private(variabile)**: Ogni thread ha la propria copia della variabile, indipendente da quella esterna.
- **shared**: Tutte le variabili sono condivise tra i thread.
- **none**: Nessuna variabile è condivisa; tutte devono avere uno scope esplicito.
- **reduction(operatore:variabile)**: Le variabili sono private, tranne `variabile` che è utilizzata per la riduzione con l'`operatore` specificato.
- **firstprivate**: Le variabili sono private e inizializzate con il valore della variabile esterna.
- **lastprivate**: Le variabili sono private e il loro valore viene copiato nella variabile esterna dopo il blocco.
- **threadprivate**: Le variabili sono private e mantengono il loro valore tra diverse regioni parallel.
- **copyin**: Le variabili sono private e inizializzate con il valore della variabile esterna all'inizio del blocco.
- **copyprivate**: Simile a `copyin`, ma i valori vengono copiati anche al termine del blocco.

3.3 Operazioni di Riduzione

`reduction(operatore:variabile)`

Operatori supportati: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`. Ogni thread esegue l'operazione e accumula il risultato in una singola variabile.

3.4 Controllo della Concorrenza

- **critical**: Simile a un lock, garantisce che solo un thread acceda alla sezione critica alla volta.
- **atomic**: Garantisce l'esecuzione atomica di operazioni specifiche, migliorando le prestazioni rispetto a `critical` quando applicabile.

3.5 Parallelizzazione dei Cicli

- **for**: Parallelizza l'esecuzione del ciclo `for`, anche quelli annidati, a seconda della posizione della pragma.
- **collapse(numero_for)**: Insieme a `for`, OpenMPI gestisce la parallelizzazione di più cicli annidati.

3.6 Misurazione del Tempo di Esecuzione

```
double start = omp_get_wtime();
/* codice */
double stop = omp_get_wtime();
```

Calcola il tempo di esecuzione del codice tra le due chiamate a `omp_get_wtime()`.

4 CUDA

4.1 Chiamata alla funzione `hello`

Esecuzione della funzione `hello` in un singolo blocco con 10 thread. La chiamata è asincrona, quindi è necessario sincronizzare la GPU per attendere il completamento dell'esecuzione.

```
hello<<<1, 10>>>();
cudaDeviceSynchronize();
```

4.2 Decoratori per funzioni CUDA

- `__global__`: Indica che la funzione è eseguita sulla GPU e può essere chiamata dalla CPU.
- `__device__`: Indica che la funzione è eseguita sulla GPU ma può essere chiamata solo da altre funzioni eseguite sulla GPU.
- `__host__`: Indica che la funzione è eseguita sulla CPU.

4.3 Numero di GPU disponibili

Ottiene il numero di GPU disponibili nel sistema.

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
```

4.4 Proprietà delle GPU

Ad ogni GPU è assegnata una struttura per le relative informazioni.

```
struct cudaDeviceProp {
    char name[256]; // Nome del device
    int major; // Major compute capability number
    int minor; // Minor compute capability number
    int maxGridSize[3]; // Dimensioni massime della griglia
    int maxThreadsDim[3]; // Dimensioni massime dei blocchi
    int maxThreadsPerBlock; // Numero massimo di thread per blocco
    int maxThreadsPerMultiProcessor; // Numero massimo di thread per
       SM
    int multiProcessorCount; // Numero di SM
    int regsPerBlock; // Numero di registri per blocco
    size_t sharedMemPerBlock; // Shared memory disponibile per blocco
       in byte
```

```
12      size_t totalGlobalMem; // Memoria globale disponibile sul device
            in byte
13      int warpSize; // Dimensione del warp in thread
14  };
```

4.4.1 Ottenere le proprietà di una GPU

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, numero_gpu);
```

4.4.2 Accesso alle singole proprietà

```
prop.name
prop.major
// Altri campi...
```

4.5 Gestione della memoria GPU

4.5.1 Allocazione della memoria

Alloca memoria sulla GPU specificata.

```
cudaMalloc((void**)&d_A, size);
```

4.5.2 Copia della memoria tra GPU e CPU

Copia i dati tra CPU e GPU.

```
cudaMemcpy(A, B, size, direzione);
```

Direzioni di copia disponibili

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyDefault`

NOTA: I puntatori `A` e `B` allocati sono diversi tra GPU e CPU, ma con CUDA 6 è stata introdotta la Unified Memory che permette di avere un unico spazio di memoria.

4.5.3 Liberare la memoria

Libera la memoria precedentemente allocata sulla GPU.

```
cudaFree(A);
```