**Progetto 10**

**Framework master/worker multithreading usando MPI e Pthreads**

*Simone Ruberto*

*Riccardo Ruberto*

Il framework creato permette di sfruttare il paradigma master/worker multithreading per effettuare in maniera parallela operazioni su matrici di grandi dimensioni.

* **Architettura:**

L’architettura utilizzata è di tipo *ibrido*, in quanto ci sono due tipi di comunicazioni da gestire.

La comunicazione tramite master e worker (processi) sfruttando l’infrastruttura fornita da MPI, che permette una comunicazione tra *sistemi a memoria distribuita*.

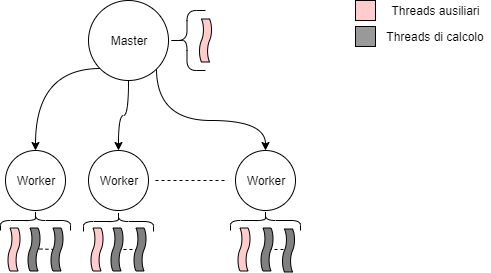
Ogni processo utilizzerà threads ausiliari e, nel caso dei worker anche threads per effettuare le operazioni richieste (threads di calcolo). Per la gestione dei threadutilizzeremo Pthread.

L’utilizzo di **code** ci permettono di accodare le task e eseguirle in ordine, vengono usate 2 code:

* **Actions:** coda contenente le azioni programmate, cioè quelle task che devono essere prese in carico dai workers.

La coda è di grandezza limitata e permette operazioni di push e pop, non accetta nuove task se i worker hanno tutti i threads di calcolo occupati.

* **Pending actions:** coda contente le task in esecuzione. Per ogni task in esecuzione avremo lo stato dei worker e gli output attesi da ciascuno e le richieste pendenti da parte di essi (visto l’utilizzo di chiamate non bloccanti).



* ***Master:***  il processo che gestisce il lavoro nel framework, farà uso di un thread ausiliario che stamperà a video il menù delle task disponibili e gestisce l’input della scelta.Si occuperà di estrarre le tasks da eseguire dalla coda *actions,* per ogni task estratta essa verrà aggiunta a *pending* actions non appena inizieràl’esecuzione.

Controllerà costantemente pending action in modo da rimanere aggiornato sullo stato di avanzamento delle azioni.

* ***Worker:*** il processo che si occupa di elaborare le task. Utilizza un thread ausiliare che sta in ascolto di nuove task da parte del master e invia al master i risultati elaborati dai threads di calcolo.

Il lavoro del worker viene suddiviso sui threads di calcolo liberi che eseguiranno le task.

Una volta completata una task i dati verranno spediti al master.

***Obbiettivi:***

* **MPI:**

Il *messagge passing* fornito da MPI permette di utilizzare funzioni bloccanti e non, quindi per massimizzare la reattività abbiamo deciso di utilizzare funzioni di tipo *Immediate*, non bloccanti (ad esempio per controllare quali worker hanno terminato l’esecuzione).

* **PTHREADS:**

Abbiamo evitato di creare situazioni di stallo all’interno dei singoli Worker, andando ad eliminare la necessità di *barrier* o *sezioni* *critiche.* Infatti i thread avendo a disposizione una memoria condivisa (nel processo) lavoreranno su spazi di memoria condivisi, ma ognuno su sezioni di memoria indipendenti, così da non utilizzare *primitive di sincronizzazione* per accedere ai dati condivisi.

I *mutex* vengono utilizzati solo dal processo master per prelevare in modo concorrente da actions (*coda azioni programmate*).

***Limitazioni:***

* **Utilizzo di Isend/Irecv anzichè Igatherv/Iscatter:**

Per limitare il numero di comunicazioni sarebbe stato utile poter utilizzare *Igatherv e Iscatter* ma abbiamo riscontrato dei problemi.

Igatherv non garantisce sempre l’arrivo dell’informazione, abbiamo testato con moltissime instanze e anche con un *MPI\_Wait* associato (simulazione modalità bloccante).

L’alternativa era utilizzare un *MPI\_Gatherv* bloccante oppure utilizzare più *Isend e Irecv.*

* **Limitazione dimensione matrici create:**

Abbiamo riscontrato errori nell’eseguire funzioni di allocazione della memoria dinamica (malloc) nel momento in cui il numero di colonne e di righe raggiungeva una notevole dimensione.

Abbiamo cercato di ovviare al problema controllando i valori di ritorno e all’occorrenza bloccare l’esecuzione del programma.

* **Numeri random:**

La creazione di numeri random tramite la funzione *rand()* usata dai threads di calcolo e andando a settare un seed generatore attraverso la funzione *rands(time())* non assicurano la generazione di numeri random , quindi le matrici generate avranno delle ripetizioni di valori.

I numeri generati saranno limitati a *sizeof(int)/2* per evitare di superare il valori rappresentabili da un int, questo avviene anche nella funzione *sumMatrix.*