Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 11 luglio 2023

Durata della Prova: 2,5 ore

1. (3 punti) Si supponga che il tempo di comunicazione in una rete Omega, che connetta **16** nodi di input a **16** nodi di output, per una comunicazione punto-punto **tra due stadi adiacenti**, sia uguale a **Tw**. Si trascurino i tempi di computazione e startup del messaggio, e sia $\mathbf{T_1}$ il tempo di comunicazione tra l'input 3 e l'output 10 e $\mathbf{T_2}$ il tempo di comunicazione tra l'input 14 e l'output 1. Si avrà, in media:

    a. $\mathbf{T_1 < T_2}$
    b. $\mathbf{T_1 > T_2}$
    c. $\mathbf{T_1 = T_2}$
    d. Dipende dal numero di bit che sono pari ad **1** nell'operazione XOR tra il nodo sorgente e destinatario nei due casi

2. (3 *punti*) si consideri Il seguente pseudo codice eseguito in un contesto a memoria condivisa:

```
// dichiarazione del matrice a
. . .

int threadA(void)
{
    int s = 0;
    int i;
    for (i = 0; i < NROWS; ++i)
      for (j = 0; j < NCOLS; ++j)
        s += a[i][j];
    return s;
}
void threadB(void)
{
    int i;
    for (i = 1; i < NCOLS; ++i)
        a[i][0] = a[i-1][0] + 1;
}
```

```
void threadC(void)
{
    int i;
    for (i = 1; i < NROWS; ++i)
        a[0][i] = a[0][i-1] + 1;
}
```

Considerando NROWS e NCOLS “abbastanza grandi”, il codice presenterà problematiche di false sharing:

a. Eseguendo i thread A e B
b. Mai
c. Eseguendo i thread A e C
d. Eseguendo i thread B e C

3. (*fino a 10 punti*) Si vuole implementare un codice posix che esegua un task “principale” (`doTaskMain()` nel codice seguente) e <u>contemporaneamente</u> dei task secondari caratterizzati da alcuni vincoli d’esecuzione. In particolare, ad ogni task è assegnato un intero, detto livello (`level` nel codice seguente). Ogni task è composto da 2 sottotask: il primo (`doPriorityTask()`) deve essere eseguito tenendo conto del livello, ovvero, devono essere eseguiti i `doPriorityTask()` dei task di livello 0 prima di quelli del livello 1, poi quelli di livello 2 e così via. Il secondo sottotask (`doTask()`) deve essere eseguito **dopo** l’esecuzione del corrispondente `doPriorityTask()` ma non è richiesto alcun altro vincolo d’esecuzione rispetto agli altri task.
   Si consideri il seguente codice:

```
#define NUM_TASK_X_LEVEL XXX
#define NUMLEVELS XXX // 0, 1, 2...

struct task{
      int level;
      void doPriorityTask()
      void doTask()
};

task tasks[NUM_TASK_X_LEVEL*NUMLEVELS];

void doTaskMain()

void buildTasks()

...
```

```
int main(int argc, char* argv[]) {

      buildTasks();
      ...
    return 0;
}
```

`buildTasks()` riempie opportunamente l'array `tasks`, e si consideri tale funzione come **già** implementata. Lo stesso valga per `doTaskMain()` e per i metodi `doPriorityTask()` e `doTask()` di ogni oggetto `task`. Si completi il codice relativo al `main`, introducendo le dichiarazioni di variabili e le funzioni che si ritengono opportuni
Per semplicità, si considerino gli oggetti `task` dell'array `tasks` come ordinati per livello.

4. (*5 punti*) Si consideri una matrice NxN memorizzata in modo contiguo in memoria, e sia k un intero positivo. Si vogliono spedire tutte le k^2 porzioni di matrice ottenute partizionando sia le N righe che le N colonne per k partizioni. Si consideri il caso generale in cui N NON è divisibile per k.
   Volendo spedire ogni porzione con una singola send MPI, quanti datatype di tipo TypeVector è necessario, al minimo, definire?

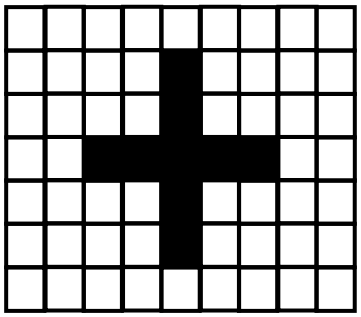
   a. 1
   b. k^2
   c. k
   d. 4

5. (*fino a 9 punti*)

Si consideri il seguente codice che realizza l'esecuzione di un automa cellulare di dimensione NROWSxNCOLS in parallelo con MPI, nel quale il dominio dell'automa è partizionato sia su X che sulla Y (X_PARTITIONS sulla X e Y_PARTITIONS sulla Y).

```
#define NCOLS XXX
#define NROWS XXX
#define X_PARTITIONS XXX
#define Y_PARTITIONS XXX
#define nsteps XXX

int rank, nproc;
int rankUp, rankDown, rankLeft, rankRight;
...

void init(){
...
}


void swap(){
...
}


void exchBord(){
 ...
}


int main(int argc, char *argv[]) {

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &nproc);


      rankUp = getUpNeighbour();
      rankDown = getDownNeighbour();
      rankLeft = getLeftNeighbour();
      rankRight = getRightNeighbour();


      init();

      initAutoma();

      for(int s=0;s<nsteps;s++){
            exchBord();
            transFunc();
            swap();
      }

      MPI_Finalize();

      return 0;
}
```

Si considerino già realizzate le funzioni `initAutoma()`, `transFunc()`, `getUpNeighbour()`, `getDownNeighbour()`, `getLeftNeighbour()`, `getRightNeighbour()` che implementano rispettivamente l'inizializzazione dell'automa cellulare, l'applicazione della funzione di transizione e il calcolo dei rank dei nodi "vicini". Si fornisca una implementazione delle funzioni: `init()`, `exchBord()`, `swap()` e della dichiarazione delle variabili necessarie in modo da gestire come vicinato quello mostrato nella seguente figura (Von-Neumann-like di raggio 2):



N.B. Si utilizzi **<u>una sola</u>** operazione di send e di receive per bordo tramite utilizzo oculato dei TypeVector

.

Signature Posix

```
//creazione thread
int pthread_create(pthread_t * thread,
                       const pthread_attr_t * attr,
                       void * (*start_routine)(void *),
                       void *arg);


// join
int pthread_join( pthread_t thread,void** value_ptr );


//mutex
int pthread_mutex_init(pthread_mutex_t *mutex,
     pthread_mutex_attr *attr);
int pthread_mutex_lock(pthread_mutex_t* mutex );
int pthread_mutex_unlock(pthread_mutex_t* mutex );
int pthread_mutex_destroy(pthread_mutex_t *mutex);

//condition
int pthread_cond_init( pthread_cond_t *cond,
     pthread_condattr_t *cond_attr )
int pthread_cond_destroy( pthread_cond_t *cond )
pthread_cond_wait(&a_c_v,&a_mutex);
```

```
pthread_cond_signal (pthread_cond_t *cond)
pthread_cond_broadcast (pthread_cond_t *cond)
```

Signature MPI

```
MPI_Init (&argc,&argv);

MPI_Comm_size (comm,&size);

MPI_Comm_rank (comm,&rank);

MPI_Finalize ();

int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm );

int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status );

MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count
);

int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request );

int MPI_Wait (MPI_Request  *request, MPI_Status   *status);

int MPI_Test (MPI_Request  *request, int *flag, MPI_Status  *status)

int MPI_Type_vector(int block_count, int block_length, int stride,
MPI_Datatype old_datatype, MPI_Datatype* new_datatype);

int MPI_Type_commit(MPI_Datatype* datatype);

int MPI_Type_free(MPI_Datatype* datatype);
```