Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 20 settembre 2023

Durata della Prova: 2,5 ore

1. (4 *punti*) L'esecuzione del seguente programma Posix:

```
void* run(void * arg){
      int* p = (int*)arg;
      sleep(1);
      sleep(*p);
      return NULL;
}

int main(int arg, char* argv[])
{
      pthread_t thid[3];
      int i=4;

      do{
            i--;
            sleep(1);
            pthread_create(&thid[3-i], NULL, &run, &i);
      } while(i>1);

      for(int i = 0; i< 3; i++)
            pthread_join(thid[i], NULL);

}
```

su una architettura quad-core, durerà all'incirca:

   a. Sempre poco più di 4 secondi
   b. Sempre poco più di 5 secondi
   c. A volte poco più di 4 e a volte poco più 5 secondi
   d. A volte poco più di 5 e a volte poco più 6 secondi

2. (*2 punti*) Dato un ipercubo di dimensione 4, sia il tempo di comunicazione punto-punto tra due nodi adiacenti uguale a **Tw**. Trascurando i tempi di computazione, il tempo impiegato da un messaggio inviato dal nodo 1 al nodo 15 è uguale in media a:
      a. Tw
      b. 3 * Tw
      c. Tw * Tw
      d. 2 * Tw

3. (*2 punti*) Sia **4 * p * log p** l'overhead T_o generato da un problema parallelo che gira su p processori e **n** il suo tempo seriale T_s. Affinché l'efficienza venga mantenuta al 60% (cioè 0.6), la relazione tra n e p deve essere :
    e. n = 2 * p * log p
    f. n = 16 * log p
    g. p = 4 * n
    h. n = 6 * p * log p

4. (*fino a 9 punti*) Il seguente codice esegue una lista di "job" in modo parallelo, un job per thread. Ogni job è realizzato tramite un oggetto di tipo `job` e l'elenco dei job, il cui numero è pari a `NJobs`, è contenuto nell'array `jobs` il quale viene inizializzato dalla funzione `buildJobs()`. Ogni oggetto di tipo `job` contiene anche gli interi `type` (valori da 1 a `NType`) e `waitfor` che specificano i vincoli di esecuzione tra i thread dei job. In particolare, posto `waitfor` pari ad un dato intero X , se X è uguale a 0 il job può essere eseguito immediatamente, in caso contrario dovrà aspettare la terminazione di tutti i job già lanciati con `type` = X.

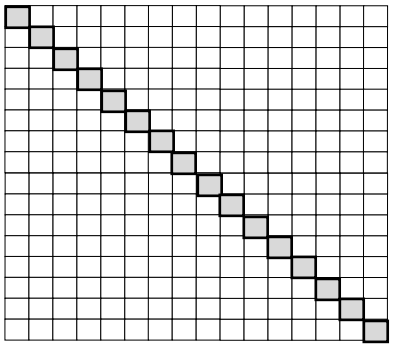
```
const int NType=XXX, NJobs=XXX;

struct job{
   int type;
   int waitFor;
   void doJob()
};

job jobs[NJobs];

void init() {
   ...
}

void* run(void* arg) {
   ...
}

int main(int argc, char* argv[]) {
    buildJobs();
    pthread_t th[NJobs];

    init();
```

```
    for (int i = 0; i < NJobs; i++)
        pthread_create(&th[i], NULL, &run, &(jobs[i]);

    for (int i = 0; i < NJobs; i++)
        pthread_join(th[i], NULL);

  return 0;
  }
```

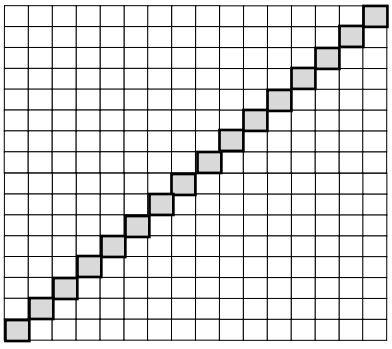
Si consideri la funzione `buildJobs()` e i metodi `doJob()` di ogni oggetto `job` come già realizzati e si fornisca una implementazione delle funzioni `init()` e `run()` nonché dichiarazioni di variabili necessarie in modo da garantire i vincoli di esecuzione sopradescritti.

5. (*fino a 4 punti*)

Si consideri uno scambio di dati tramite send/receive MPI tra 2 nodi entrambi ospitanti una matrice quadrata di interi, in modo che il contenuto della diagonale principale della matrice del nodo 0 venga copiato nella diagonale secondaria della matrice del nodo 1 (si veda la seguente figura):



Nodo 0



Nodo 1

Si consideri il seguente codice:

```
#define N 200
int rank;
int mat[N][N];

void exchDiagonal(){
 if (rank == 0){
     MPI_Type_vector(N, 1, A, MPI_INT, &SendDT);
     MPI_Type_commit(&SendDT);
     MPI_Send(mat + B, 1, SendDT, 1, 69, MPI_COMM_WORLD);

} else
```

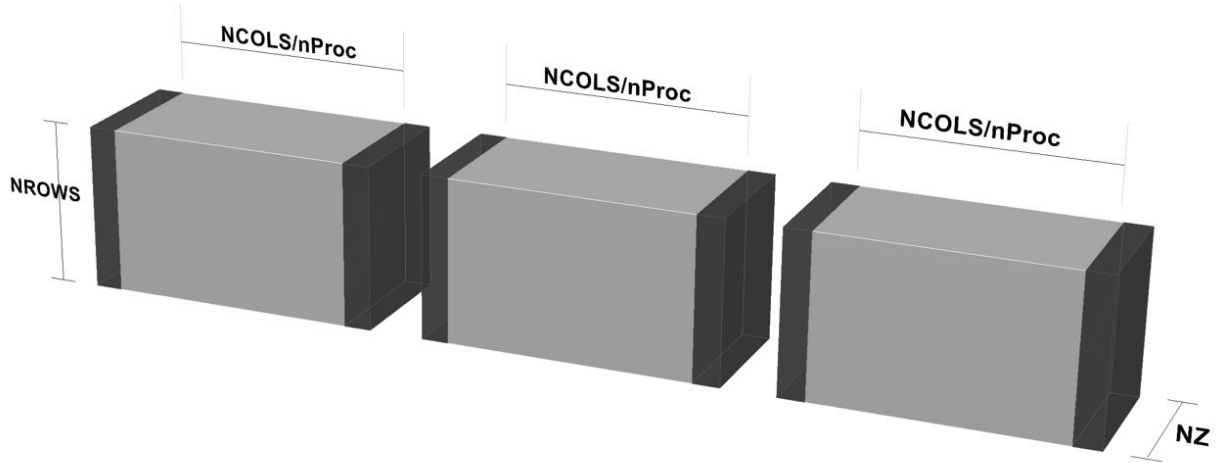
```
  {
     MPI_Type_vector(N, 1, C, MPI_INT, &RecvDT);
     MPI_Type_commit(&RecvDT);
     MPI_Recv(mat + D, 1, RecvDT, 0, 69, MPI_COMM_WORLD, NULL);

}

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

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


      exchDiagonal();

      MPI_Finalize();

      return 0;
}
```

Per quali valori di A, B, C e D viene eseguito lo scambio dati richiesto?

a. A=N+1, B=0, C=N-1, D=N
b. A=N, B=N, C=N+1, D=0
c. A=N, B=0, C=N, D=0
d. A=N+1, B=N, C=N-1, D=N

6. (*fino a 9 punti*)

Si consideri il seguente codice che realizzi l'esecuzione di un automa cellulare 3D di dimensione NROWS x NCOLS x NZ in parallelo con MPI, nel quale il dominio dell'automa è partizionato lungo la X (si veda figura seguente).




```
#define NROWS XXX
#define NCOLS XXX
#define NZ XXX
#define v(r,c,z) ((r)*(NCOLS/nProc+2)*NZ+(c)*NZ+(z))

int* readM;
int* writeM;

int nsteps=XXX;

int rank, rankLeft, rankRight;

int nProc;

...


void init(){
...
}

void exchBord(){
...
}


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

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

```
        readM = new int[NROWS*(NCOLS/nProc+2)*NZ];
        writeM = new int[NROWS*(NCOLS/nProc+2)*NZ];

        rankLeft = (rank-1+nProc)%nProc;
        rankRight = (rank+1)%nProc;

        init();

        initAutomata();

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

        delete[] readM;
        delete[] writeM;

        MPI_Finalize();
        return 0;
}
```

Si considerino già realizzate le funzioni `initAutoma()`, `transFunc()` e `swap()` che implementano rispettivamente l'inizializzazione dell'automa cellulare, l'applicazione della funzione di transizione e lo swap delle matrici. Si fornisca una implementazione delle funzioni `init()` e `exchBord()` e della dichiarazione delle variabili necessarie in modo da consentire l'esecuzione parallela dell'AC. In particolare, si richiede che la spedizione/ricezione di un bordo (la parte in grigio scuro nella figura) avvenga tramite una sola operazione di send/recv tramite l'utilizzo oculato di MPI_Type_vector.

.

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);
```