Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 9 febbraio 2023

Durata della Prova: 2 ore e 30 minuti

1. (*3 punti*) Si supponga che il tempo di comunicazione in una rete Omega, che connetta **P** nodi di input a **P** nodi di output, per una comunicazione punto-punto tra due stadi adiacenti, sia uguale a **Tw**. Trascurando i tempi di computazione, si indichi quale è il **minimo** tempo impiegato da un messaggio inviato da un qualsiasi nodo di input per raggiungere un qualsiasi nodo di output.

2. (*4 punti*) Si consideri il seguente codice:

```
void* run(void * arg){
        int i = *(int*)arg;
        if (v[i]){
                pthread_mutex_lock(&mutex1);
                pthread_mutex_lock(&mutex0);
        } else{
                pthread_mutex_lock(&mutex0);
                pthread_mutex_lock(&mutex1);
        }

        pthread_mutex_unlock(&mutex0);
        pthread_mutex_unlock(&mutex1);

}
int main(int arg, char* argv[])
{
        pthread_t thid[numThread];
        pthread_mutex_init(&mutex0,NULL);
        pthread_mutex_init(&mutex1,NULL);
        for(int i = 0; i< numThread; i++){
                int * p = new int;
                (*p)=i;
                int ris = pthread_create(&thid[i], NULL, &run, p);
                if (ris){
                        printf("errore creazione thread\n");
                        exit(-1);
                }
        }
        for(int i = 0; i< numThread; i++)
                pthread_join(thid[i], NULL);
        pthread_mutex_destroy(&mutex0);
        pthread_mutex_destroy(&mutex1);
}
```

Quali delle seguenti valorizzazioni del vettore **v** garantisce l'assenza di possibile deadlock:

a. false, false, true, false, false, false
b. true, false, true, false, true, false
c. false, false, false, false, false, false
d. Non è possibile evitare il deadlock

3. (*4 punti*) Si consideri la seguente porzione di codice MPI:

```
MPI_Type_vector(NBlocks, BlockSize, Stride, MPI_BYTE, &newDT);
MPI_Isend(&v[0], 1, newDT, rank, label, MPI_COMM_WORLD, &request);
```

Quanti byte di memoria devono essere allocati al **minimo** per l'array v affinchè si possa escludere un segmentation fault? (n.b. si consideri BlockSize<=Stride)

a. BlockSize*NBlocks
b. Stride*NBlocks
c. (Stride-1)*NBlocks+BlockSize
d. 0

4. (3 *punti*) Il tempo impiegato per l'operazione di **compare/split** in un algoritmo di sorting parallelo implementato su **p** nodi e **n** elementi è proporzionale a:

a. n^2/p
b. Dipende dal particolare algoritmo parallelo di sorting
c. p log n
d. n/p

5. (*fino a 9 punti*) Si vuole parallelizzare un automa cellulare tramite thread posix. In particolare, si vuole realizzare una versione in cui ogni thread amministrerà una cella diversa (il numero di thread è quindi pari al numero di celle del dominio dell'automa).

Si consideri il seguente codice:

```
#define NCOLS 6
#define NROWS 6
#define nsteps 99


struct Cella{
```

```
      int readM[3][3];
      int writeM[3][3];
};

Cella domain[NROWS*NCOLS];

…

void init(){
…
}

void synch(int i){
…
}

void* run(void * arg){
      int i = *(int*)arg;

      for(int s=0;s<nsteps;s++){
            transFunc(i);
            synch(i);
            swap(i);

      }
}

int main(int arg, char* argv[])
{
      init();
      initAutoma();

      pthread_t thid[NROWS*NCOLS];
      for(int i = 0; i< NROWS*NCOLS; i++){
            int * p = new int;
            (*p)=i;
            int ris = pthread_create(&thid[i], NULL, &run, p);
            if (ris){
                  printf("errore creazione thread\n");
                  exit(-1);
            }
      }
      for(int i = 0; i< NROWS*NCOLS; i++)
            pthread_join(thid[i], NULL);
}
```

Il dominio dell'automa cellulare è memorizzato nell'array `domain`, in particolare in ogni elemento di tale array è composto da 2 matrici 3X3, una di lettura (`readM`) ed una di scrittura (`writeM`)), che contengono nell'elemento centrale lo stato della cella corrispondente dell'automa (es: lo stato della cella (3,4) dell'automa è memorizzato in `domain[3*4].readM(1,1)`) e negli elementi restanti la replica dello stato delle celle vicine (es: lo stato della cella dell'automa (3,4) è memorizzato come replica nella cella (2,5) in `domain[2*5].readM(2,0)`). In questo modo, l'esecuzione dell'automa consisterà in ogni thread che esegue il tipico ciclo di controllo, dove la funzione di transizione (`transFunc`) e lo swap delle matrici (`swap`), utilizzeranno come ingresso/uscita le 2 matrici 3X3 della cella corrispondente. Tra le chiamate di queste 2 funzioni è necessario, quindi, sincronizzare i thread (`synch`) in modo che lo swap inizi solo dopo

che le celle di replica dei vicini siano state aggiornate. Tale sincronizzazione potrebbe essere realizzata in modo *globale* implementando una barriera tra tutti i thread, ma viene richiesto, nello specifico, che questa avvenga in modo ***locale***, ovvero che ogni thread si sincronizzi solo coi thread "vicini".
Si consideri le funzioni: `transFunc`, `swap` e `initAutomata` come già implementate e si implementi la sincronizzazione locale dei thread fornendo una implementazione del metodo `synch`, con l'aggiunta di eventuali variabili a supporto, e della funzione `init` dove tali variabili vengono correttamente inizializzate.

6. (*fino a 7 punti*) Si vuole parallelizzare un automa cellulare con MPI considerando il caso in cui ogni cella è assegnata ad un diverso processo MPI (il numero di processi MPI è quindi pari al numero di celle del dominio dell'automa).
   Si consideri la seguente implementazione:

```
#define NCOLS 200
#define NROWS 200
int nsteps=120;
int rank;

...

void init(){
...
}


void swap(){
...
}


void exchBord(){
 ...
}


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

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

      init(rank);

      initAutoma();

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

      MPI_Finalize();

}
```

Si considerino le funzioni `initAutoma()` e `transFunc()`, che implementano rispettivamente l'inizializzazione dell'automa cellulare e l'applicazione della funzione di transizione per la cella amministrata dal processo MPI, come già realizzate e si fornisca una implementazione delle funzioni: `init()`, `exchBord()`, `swap()` e della dichiarazione delle variabili necessarie.
N.B. come semplificazione, si consideri solo il vicinato di Von Neumann (ovvero composto da: Nord, Sud, Est e Ovest).

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);
```