Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 13 gennaio 2023

Durata della Prova: 2 ore e 30 minuti

1. (4 *punti*) Si consideri un database di prodotti alimentari. Ogni prodotto ha un identificativo, una descrizione e un fornitore. Questo database puo' essere memorizzato tramite un array di struct, dove ogni struct contiene {id, descrizione, fornitore} (**caso A**) oppure tramite 3 array, uno contenente tutti gli id, uno contenente tutte le descrizioni e uno contenente tutti i fornitori (**caso B**). Naturalmente, in questo ultimo caso, le informazioni di ogni prodotto si trovano nella medesima posizione nei tre array. Quali delle seguenti affermazioni risulta vera riguardo la ricerca di un prodotto tramite il suo id dal punto di vista della gestione ottimale della cache?

   a. Se gli id non sono ordinati (ricerca lineare), il caso B risulta più efficiente
   b. Se gli id non sono ordinati (ricerca lineare), il caso A risulta più efficiente
   c. Se gli id sono ordinati (ricerca binaria), il caso B risulta più efficiente
   d. Se gli id sono ordinati (ricerca binaria), il caso A risulta più efficiente

2. (3 *punti*) In una implementazione parallela di un automa cellulare tramite partizionamento del dominio spaziale, non è necessario lo scambio dei bordi (halo cells):

   a. In un contesto a memoria distribuita
   b. Quando l'automa cellulare non è toroidale
   c. mai
   d. In un contesto a memoria condivisa

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

```
…
MPI_Comm_size(MPI_COMM_WORLD, &nsize);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (vet[rank]==0) {
      MPI_Send(data, MAXSIZE, MPI_INT, (rank+1)%nsize, TAG, MPI_COMM_WORLD);
      MPI_Recv(data, MAXSIZE, MPI_INT, (rank-1+nsize)%nsize, TAG,
            MPI_COMM_WORLD, &status);
} else {
      MPI_Recv(data, MAXSIZE, MPI_INT, (rank-1+nsize)%nsize, TAG,
            MPI_COMM_WORLD, &status);
      MPI_Send(data, MAXSIZE, MPI_INT, (rank+1)%nsize, TAG, MPI_COMM_WORLD);
}
```

Affinchè non si possa avere una situazione di deadlock, il vettore `vet` deve avere come <u>minimo</u> un numero di valori diversi da 0 pari a:

a. 0
b. 1
c. nsize/2
d. non è possibile evitare il deadlock

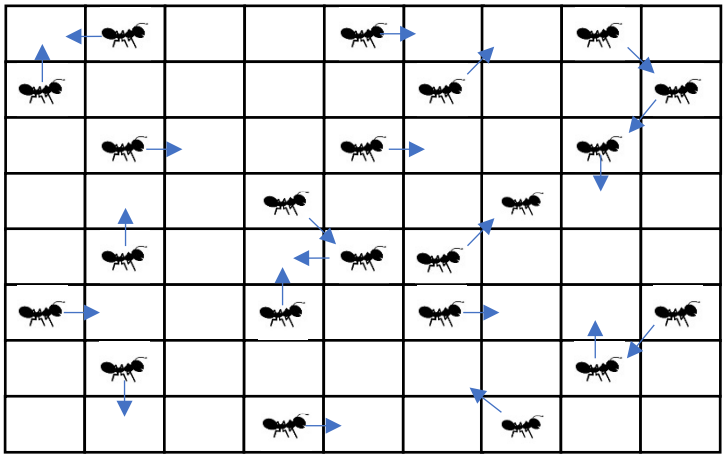
4. (5 *punti*) L'esecuzione del seguente programma:

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

      sleep(4-(*p));

}

int main(int argc, char* argv[]) {
      pthread_t thid;
      int i = 1;
      pthread_create(&thid, NULL, &run, &i);
      sleep(1);
      i++;
      sleep(i);
      pthread_join(thid, NULL);

   return 0;
}
```

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

a. Poco più di 3 secondi
b. Poco più di 4 secondi
c. La durata può cambiare ad ogni run
d. Poco più di 7 secondi

5. (*fino a 8 punti*) Si vuole simulare il movimento di formiche lungo un territorio bidimensionale. Si immagini che le formiche siano all'interno di celle di una matrice (**ogni cella può contenere al massimo 1 formica**), come nella seguente figura:



Ogni formica ha una posizione ed una direzione iniziale (tra le 8 direzioni cardinali) e procede lungo tale direzione una cella per volta. Ogni volta che una formica non può proseguire lungo la sua direzione, perché raggiunge una estremità del territorio o perché la cella di arrivo contiene già un'altra formica, cambia la sua direzione in modo causale tra quelle percorribili.
Si consideri il seguente codice nel quale ogni formica è associata ad un diverso thread:

```
struct ant{
      int posX,posY;
      int dirX, dirY; //es.: NORD (dirX=0, dirY=-1), SUDEAST (dirX=1,dirY=1)…
      pthread_t thid;
};

ant* ants;
int nAnts;
const int nCellX=10;
const int nCellY=10;
pthread_cond_t cond;
pthread_mutex_t mutex;
bool start=false;

...



void init(){
      ...
}
```

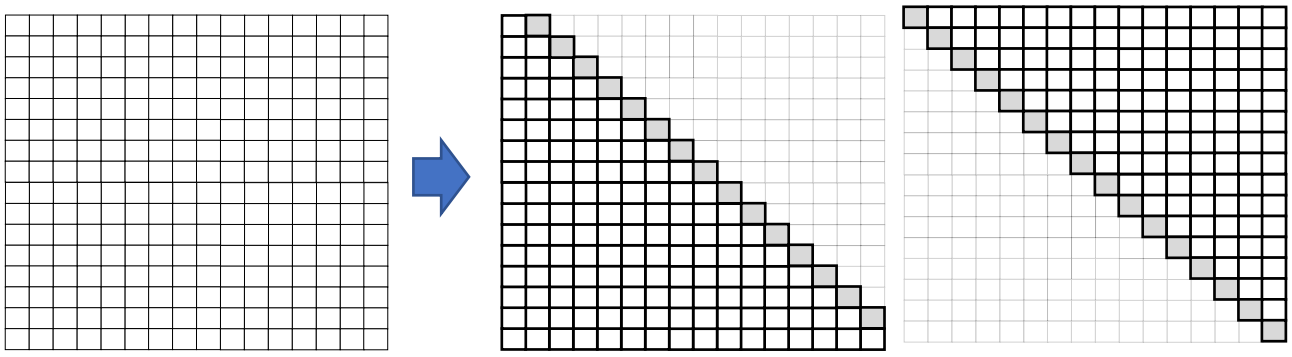
```
void move(ant& a){
      ...
}

void* antRun(void* arg) {
      ant* me = (ant*)arg;
      pthread_mutex_lock(&mutex);
      while(!start){
            pthread_cond_wait(&cond, &mutex);
      }
      pthread_mutex_unlock(&mutex);

      while(true){
            move(*me);
      }

}

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

      pthread_mutex_init(&mutex, NULL);
      pthread_cond_init(&cond, NULL);
      setNAnts(nAnts);
      ants = new ant[nAnts];

      for(int i=0; i<nAnts; i++){
            pthread_create(&ants[i].thid, NULL, &antRun, &ants[i]);
            setPosDir(i, ants[i].posX, ants[i].posY, ants[i].dirX,ants[i].dirY);
      }

      init();

      start=true;
      pthread_cond_broadcast(&cond);

      for(int i=0; i<nAnts; i++){
            pthread_join(ants[i].thid, NULL);
      }
    return 0;
}
```

Le funzioni `setNAnts()` e `setPosDir()` servono per il setup iniziale, ovvero stabilire quante formiche ci sono, le dimensioni in celle della matrice, e la posizione e la direzione iniziale di ciascuna formica. Si consideri tali funzioni come già implementate. In pratica, prima dell'invocazione della funzione `init()` nel `main()`, l'array `ants` conterrà la lista di tutte gli oggetti formica (ogni oggetto formica contiene: la posizione , la direzione iniziale e l'id del thread associato).
Implementare le funzioni `init()` e `move()` (ed eventuali dichiarazioni di variabili) in modo da riprodurre il movimento delle formiche sopradescritto.
N.B. La condition `cond` e la mutex `mutex` (insieme alla variabile booleana `start`), presenti nel codice, servono per garantire che i thread partano tutti solo dopo che l'inizializzazione è terminata.

(S*uggerimento*: prestare particolare attenzione al caso in cui 2 formiche "competano" per occupare la stessa cella)

6. (*fino a 6 punti*)

Si vuole parallelizzare un automa cellulare (AC) con MPI considerando di partizionare il dominio spaziale su 2 nodi come mostrato nella seguente figura:



Si noti che, in questo caso, lo scambio bordi riguarda la diagonale e la sopradiagonale. In particolare, il nodo 0 amministra la diagonale e riceve dal nodo 1 la replica della sopradiagonale (in grigio nella figura). Viceversa, il nodo 1 riceve la replica della diagonale (in grigio nella figura) e amministra la sopradiagonale.

Si consideri la seguente implementazione parallela di un automa cellulare di dimensione `NROWS X NCOLS` per un numero di step pari a `nsteps`:

```
#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();
```

```
	initAutoma();

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

	MPI_Finalize();

	return 0;
}
```

Si considerino le funzioni `initAutoma()` e `transFunc()`, che implementano rispettivamente l'inizializzazione dell'automa cellulare e l'applicazione della funzione di transizione ad ogni della del dominio, come già realizzate e si fornisca una implementazione delle funzioni: `init()`, `exchBord()`, `swap()` e della dichiarazione delle variabili necessarie.

N.B. lo scambio della diagonale/sopradiagonale **deve** essere eseguito tramite una sola send/receive MPI per diagonale/sopradiagonale utilizzando `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);
```