Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 16 settembre 2022

Durata della Prova: 2 ore

1. (*3 punti*) Il seguente codice eseguito in un contesto a memoria condivisa:

```
struct foo {
    int a;
    int b;
};

static struct foo f;

/* The two following functions are running concurrently: */
int threadA(void)
{
    int s = 0;
    for (int i = 0; i < 6666666; ++i)
        s += f.a;
    return s;
}
void threadB(void)
{
    for (int i = 0; i < 6666666; ++i)
        f.b++;
}
```

Presenterà problematiche di false sharing:

a. Molto probabilmente
b. Poco probabilmente
c. In base allo scheduling dei threads per l'esecuzione
d. Solo quando viene schedulato il thread A prima del thread B

2. (3 *punti*) Ricordando la formula dell'efficienza di un programma parallelo in termini di overhead (To) e tempo sequenziale (Ts), e fissato il numero di processori/threads di un problema, all'aumentare delle dimensioni del problema l'**efficienza**:

   a. Diminuisce sempre
   b. Aumenta sempre
   c. Dipende dalla frazione seriale del problema
   d. Dipende dal problema specifico

3. (fino a *4 punti*) Nel seguente spezzone di programma OpenMP viene calcolata la somma degli elementi di un array. Si verifichi la possibilità di migliorarne l'efficienza, proponendo una soluzione.

```
…
#pragma omp parallel shared(a)
{
   #pragma omp for
      for (i=0; i < N; i++)
        #pragma omp critical
            sum = sum + a[i];
…
```

4. (*fino a 4 punti*) Dato la seguente porzione di codice MPI:

```
if (rank == 0) {
    MPI_Send(data, MAXSIZE, MPI_INT, 1, TAG, MPI_COMM_WORLD);
    MPI_Recv(data, MAXSIZE, MPI_INT, 1, TAG, MPI_COMM_WORLD, &status);
    MPI_Send(data, MAXSIZE, MPI_INT, 2, TAG, MPI_COMM_WORLD);
    MPI_Recv(data, MAXSIZE, MPI_INT, 2, TAG, MPI_COMM_WORLD, &status);

} else if (rank==1) {
    MPI_Recv(data, MAXSIZE, MPI_INT, 0, TAG, MPI_COMM_WORLD, &status);
    MPI_Send(data, MAXSIZE, MPI_INT, 0, TAG, MPI_COMM_WORLD);
    MPI_Send(data, MAXSIZE, MPI_INT, 2, TAG, MPI_COMM_WORLD);
    MPI_Recv(data, MAXSIZE, MPI_INT, 2, TAG, MPI_COMM_WORLD, &status);
} else if (rank==2) {
...
}
```

Chiamando S0 l'operazione di MPI_Send indirizzata al nodo di rank 0, S1 la MPI_Send verso il nodo di rank 1 e R0/R1 le operazioni di ricezione (MPI_Recv) dal nodo di rank 0/1, rispettivamente, indicare quale ordine di esecuzione per il rank 2 NON esclude la possibilità di deadlock:

a. R0-S1-R1-S0
b. R1-R0-S0-S1
c. R1-S1-R0-S0
d. R0-R1-S1-S0

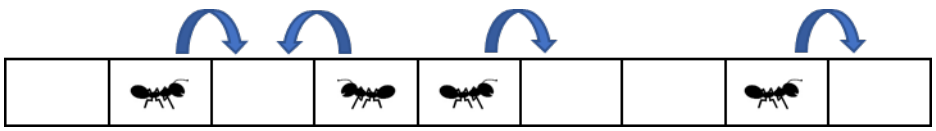
5. (4 *punti*) L'esecuzione del seguente programma:

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

      sleep(*p);

}

int main(int argc, char* argv[]) {
      pthread_t thid[3];
      int s[3];
      for(int i=3;i>=0;i--){
            pthread_create(&thid[i], NULL, &run, &i);
      }
      sleep(2);
      for(int i=0;i<3;i++)
            pthread_join(thid[i], NULL);

   return 0;
}
```

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

a. Poco più di 4 secondi
b. Poco più di 6 secondi
c. Poco più di 3 secondi
d. Poco più di 2 secondi

6. (*fino a* 6 *punti*) Si vuole simulare il movimento di formiche lungo un percorso unidimensionale. Si immagini che le formiche siano all'interno di celle di un array (**ogni cella può contenere al massimo 1 formica**), come nella seguente figura:



Ogni formica ha una posizione ed una direzione iniziale (destra o sinistra) e procede lungo tale direzione una cella per volta. Ogni volta che una formica si

“scontra” con l’estremità sinistra o destra o con un’altra formica cambia la sua direzione.

Si consideri il seguente codice nel quale ogni formica è associata ad un diverso thread:

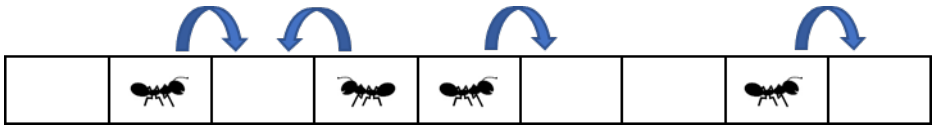
```
struct ant{
	int pos;
	int dir;
	pthread_t thid;
};

ant* ants;
int nAnts;
int nCells;
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);
	setVariables(nAnts,nCells);
	ants = new ant[nAnts];

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

	init();

	start=true;
```

```
        pthread_cond_broadcast(&cond);

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

Le funzioni `setVariables()` e `setPosDir()` servono per il setup iniziale, ovvero stabilire quante formiche e quante celle ci sono, 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 le informazioni di tutte le formiche (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.
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.

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

7. (*fino a 6 punti*) Si vuole simulare il movimento di formiche lungo un percorso unidimensionale. Si immagini che le formiche siano all'interno di celle di un array (**ogni cella può contenere al massimo 1 formica**), come nella seguente figura:



   Ogni formica ha una posizione ed una direzione iniziale (destra o sinistra) e procede lungo tale direzione una cella per volta. Ogni volta che una formica si "scontra" con l'estremità sinistra o destra dell'array, o si scontra con un'altra formica, cambia la sua direzione. Di seguito una possibile implementazione non parallela che riproduce il comportamento sopradescritto:

```
struct ant{
      int pos;
      int dir;
};

ant* ants;
int nAnts;
```

```
int nCells;

ant** readV;
ant** writeV;

void init(){
	readV = new ant*[nCells];
	writeV = new ant*[nCells];

	for(int i=0; i<nAnts; i++){
		readV[ants[i].pos]=&ants[i];
	}

}

void move(){

	for(int i=0;i<nCells;i++){
		writeV[i]=NULL;
		if (i>0 && readV[i-1]!=NULL && readV[i-1]->dir==1 &&
			readV[i]==NULL){
			//arriva il vicino da sinistra
			writeV[i]=new ant;
			*(writeV[i])=*(readV[i-1]);
			writeV[i]->pos++;

		} else if (i<nCells-1 && readV[i+1]!=NULL && readV[i+1]->dir==-1 &&
			readV[i]==NULL){
			//arriva il vicino da destra
			writeV[i]=new ant;
			*(writeV[i])=*(readV[i+1]);
			writeV[i]->pos--;
		}else if (readV[i]!=NULL && (i+readV[i]->dir<0 || i+readV[i]-
			>dir>nCells-1 || readV[i+readV[i]->dir]!=0)){
			//caso in cui cambio direzione
			writeV[i]=new ant;
			*(writeV[i])=*(readV[i]);
			writeV[i]->dir*=-1;
		}

	}

	ant** p=readV;
	readV=writeV;
	writeV=p;


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

	setVariables(nAnts,nCells);
	ants = new ant[nAnts];
	for(int i=0; i<nAnts; i++){
		setPosDir(i, ants[i].pos, ants[i].dir);
	}

	init();

	while(true){
		move();
	}
	return 0;
}
```

Le funzioni `setVariables()` e `setPosDir()` (omesse per brevità) servono per il setup iniziale, ovvero stabilire quante formiche e quante celle ci sono, e la posizione e la direzione iniziale di ciascuna formica. In pratica, prima dell'invocazione della funzione `init()`, nel `main()`, l'array `ants` conterrà le informazioni di tutte le formiche (ovvero la posizione e la direzione iniziale).

Modificare il codice riportato in modo da realizzare una versione parallela in MPI, nella quale l'array contenente le formiche è suddiviso tra i vari processi MPI che dovranno gestire la loro porzione dell'array e le formiche ivi contenute (le formiche muovendosi possono passare da un processo all'altro).
In particolare si richiede l'implementazione delle funzioni `init()` e `move()` (ed eventuali dichiarazioni di variabili). Si aggiungano, inoltre, le inizializzazioni MPI necessarie all'inizio del `main()` (ovvero prima dell'invocazione della funzione `setVariables()`).

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 OpenMP

```
#pragma omp parallel private(tid) shared (sum)

#pragma omp parallel for
```

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);
```