# Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

# Appello del 16 settembre 2022

# Durata della Prova: 2,30 ore

**(Traccia per A.A. antecedente 2021/22)**

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

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

static struct foo f; //f is shared among threads

/* 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];
…
```

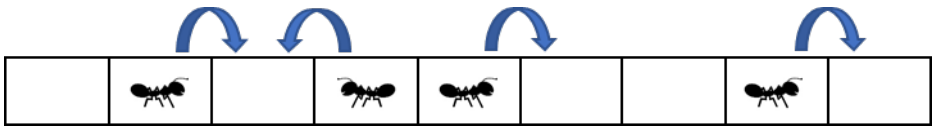
4. (*4 punti*) Data la seguente porzione di codice MPI:

```
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. (*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]);
```

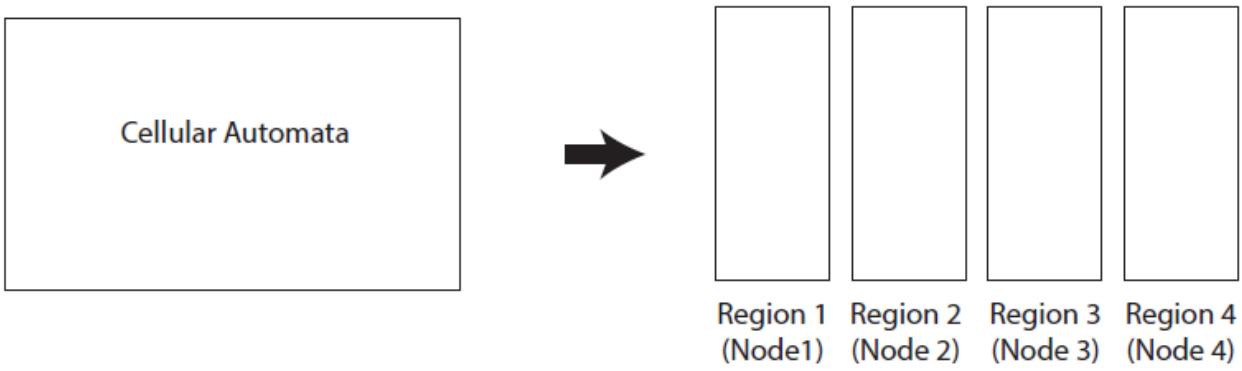
```
                writeV[i]->pos++;

            } 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).

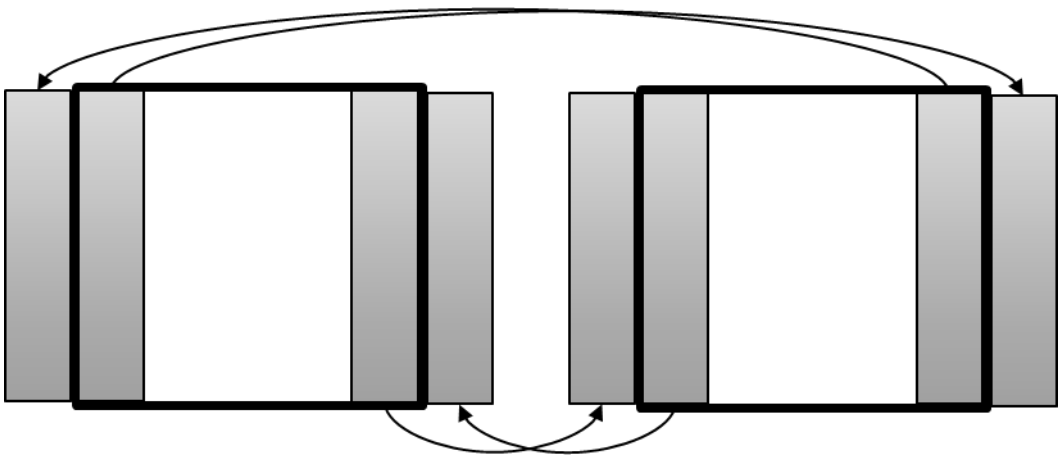
Modificare il codice riportato in modo da realizzare una versione parallela in MPI, nella quale l'array contenente le formiche è suddiviso tra i vari processi MPI che dovranno gestire la loro porzione dell'array e le formiche ivi contenute (le formiche muovendosi possono passare da un processo all'altro).
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()`).

6. (*fino a 10 punti*)

Un automa cellulare (AC) è definito da un dominio, ovvero una griglia di celle (es. una matrice bidimensionale), e da una funzione di transizione (es: regole del Gioco della Vita). L'esecuzione di un automa cellulare avviene step-by-step: ad ogni step la funzione di transizione viene applicata ad **ogni** cella del dominio, prendendo in ingresso lo stato della cella stessa e quello delle sue vicine (ovvero le celle contigue ad essa definite dal ***vicinato***, uguale per ogni cella dell AC), e producendo in uscita il nuovo stato della cella. L'implementazione di un automa cellulare prevede l'utilizzo di una matrice di lettura e di una di scrittura. La funzione di transizione legge dalla matrice di lettura e scrive in quella di scrittura. Alla fine del generico step le matrici vengono scambiate tra di loro in modo da poter eseguire il nuovo step di transizione. Si vuole implementare una versione parallela di automa cellulare con MPI che preveda di suddividere il dominio lungo la 'x' in regioni come nella seguente figura:




Per poter eseguire la funzione di transizione delle celle ai bordi delle regioni (ovvero quelle celle tale per cui le celle contigue sono situate in una regione differente) è necessario che i nodi (o processi) scambino le celle di bordo delle relative regioni come mostrato nella seguente figura:



I bordi scambiati sono colonne lunghe quando l'altezza del dominio.

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.
Per semplificare l'implementazione, si consideri solo il caso di 2 processi.

## 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);
```