Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 13 febbraio 2024

Durata della Prova: 2,5 ore

1. (*2 punti*) Sia **8 * p * ln p** l'overhead T_o generato da un problema parallelo che gira su p processori e **2 * n** il suo tempo seriale T_s. Affinché l'efficienza venga mantenuta al 80% (cioè 0.8), la relazione tra n e p deve essere :

    a. n = 8 * ln p
    b. n = 16 * p * ln p
    c. n = 2 * ln p
    d. n = 8 * p * ln p

2. (2 *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)
{
    int r;
    for (int i = 0; i < 6666666; ++i)
        r *= f.b;
}
```

Presenterà problematiche di false sharing:

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

3. (2 punti) Si supponga che il tempo di comunicazione in una **rete Omega**, che connetta **16** nodi di input a **16** nodi di output, per una comunicazione punto-punto **tra due stadi adiacenti**, sia uguale a **Tw**. Trascurando i tempi di computazione e startup del messaggio, il tempo di comunicazione tra l'input 0 e l'output 7, sarà in media:

    a. 16 * Tw
    b. 4 * Tw
    c. 2 * Tw
    d. 3 * Tw

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

```
void* run(void * arg){
      int* p = (int*)arg;
      sleep(3-*p);
      sleep(3-*p);
      return NULL;
}


int main(int arg, char* argv[])
{
      pthread_t thid[3];
      int i=0;
      while(i<3){
            pthread_create(&thid[i], NULL, &run, &i);
            sleep(1);
            i++;
      }

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

}
```

su una architettura quad-core, durerà:

a. Al massimo poco più di 3 secondi
b. Al massimo poco più di 4 secondi
c. Al massimo poco più di 5 secondi
d. Al massimo poco più di 6 secondi

5. (*fino a 7 punti*) Si vuole realizzare una funzionalità “barrier con timeout” utlizzando la libreria posix thread (senza utilizzare la specifica funzionalità “barrier” offerta della libreria) che permetta ai thread di sincronizzarsi in uno specifico punto. Inoltre allo scadere di un timeout il programma deve terminare indipendentemente dall’esecuzione dei thread. In particolare, si consideri il seguente codice:

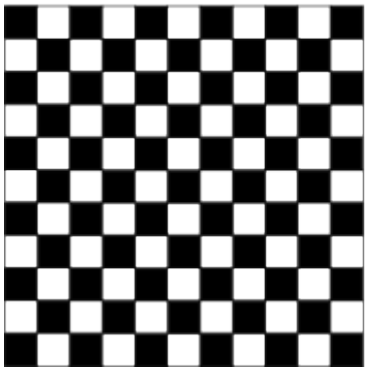
```
int nThread = XX;
int timeout = XX;

....


void initBarrier(){
    ....
}

void barrier(){
    ....
}

void waitToTerminate(){
    ....
}

void* threadFunc(void* arg) {
    doSomething();
    printf("inizio\n");
    barrier();
    printf("fine\n");
    return NULL;
}



int main(int argc, char* argv[]) {
    pthread_t th[nThread];
    initBarrier();
    for (int i = 0; i < nThread; i++) {
        pthread_create(&th[i], NULL, &threadFunc, NULL);
    }
    waitToTerminate();
    printf(“fine del main\n”);
    return 0;
}
```

Si consideri la funzione `doSomething()` come già implementata e si fornisca una implementazione delle funzioni `initBarrier()`, `barrier()` e `waitToTerminate()` (ed eventuali dichiarazioni di variabili) in modo tale che i threads si sincronizzino alla chiamata di `barrier()` e il programma termini in ogni caso se sono trascorsi `timeout` secondi dalla chiamata di `initBarrier()`. Ad esempio, se `nThread` fosse 5 l’output atteso prevedrebbe prima 5 stampe “inizio”

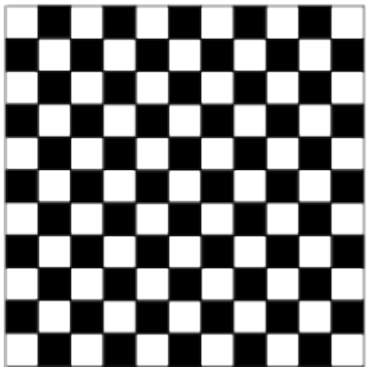
seguite da 5 stampe “fine” ed infine “fine del main”. Se però nel frattempo il timeout scatta (ad esempio nel mentre che i thread stanno eseguendo la funzione `doSomething()`) l’esecuzione dovrà terminare immediatamente e l’unica stampa sarà solo “fine del main”.

6. (5 *punti*)

Si vuole realizzare uno scambio di dati tramite send/receive MPI tra 2 nodi entrambi ospitanti una matrice quadrata (NxN) di interi, in modo da copiare, con riferimento alla figura di esempio fornita in seguito, il **contenuto di <u>tutte e sole</u> le celle indicate in nero** dal nodo 0 al nodo 1:



Nodo 0



Nodo 1

Si vuole realizzare lo scambio di dati tramite **<u>una sola</u>** operazione di send/receive utilizzando opportunamente le funzionalità di MPI_Type_vector.
Quale delle seguenti affermazioni risulta corretta?

a. Se N è pari è necessario definire solo 1 datatype
b. Se N è pari è necessario definire solo 2 datatype
c. Se N è dispari è necessario definire solo 1 datatype
d. Se N è dispari è necessario definire solo 2 datatype

7. (*fino a 7 punti*) Si realizzi uno scambio di dati tramite send/receive MPI tra 2 nodi entrambi ospitanti una matrice quadrata di interi. In particolare, utilizzando opportunamente `MPI_Type_vector`, si deve spedire il contenuto della intera matrice del nodo di rank 0 e riceverlo nella matrice del nodo di rank 1 in forma trasposta, ovvero con le righe e le colonne scambiate (in altri termini: le righe diventano le colonne e le colonne diventano le righe).

Considerando il seguente codice:

```
#define DIM XXX

int rank;
int mat[DIM][DIM];

...

void exchTransposed(){
 ...
}

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

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

      exchTransposed();

      MPI_Finalize();

      return 0;
}
```

Si fornisca una implementazione della funzione: `exchTransposed()`, e della dichiarazione delle variabili necessarie, usando il **minor numero possibile** di operazioni di send/receive.
N.B. la matrice deve risultare trasposta al termine delle operazioni di receive senza la necessità di ulteriori operazioni

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)

void sleep(int seconds)
```

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);
```