Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 16 gennaio 2024

Durata della Prova: 2,5 ore

1. (*2 punti*) Sia P(p, n) un problema parallelo di **p** processori e di dimensione **n**, caratterizzato da un overhead $\mathbf{T_o = 4\ p \log p}$ e tempo seriale $\mathbf{T_s = n}$. Affinché l'efficienza venga mantenuta al **60% (0.6)** per P, quale deve essere la relazione tra n e p ?
    a. n = 0.6
    b. n = 4 * p * log p
    c. n = 6 * p * log p
    d. n = p * log p

2. *(2 punti*) Il seguente codice eseguito in un contesto a memoria condivisa:

```
int x;  // x and y shared variables
...
int y;

/* Le due funzioni seguenti sono lanciate in parallelo */
}
void thread1(void)
{
    int prod;
    for (int i = 0; i < 1000; ++i)
        prod = prod * x;
}

void thread2(void)
{

    for (int i = 0; i < 1000; ++i)
        y++;
```

Presenterà problematiche di false sharing:

a. Sempre
b. Mai
c. In base alla posizione in memoria di x e y
d. Se la distanza in memoria in byte tra x e y è maggiore o uguale a 1000 * sizeof(int)

3. (*2 punti*) Dato un ipercubo di dimensione 3, sia il tempo di comunicazione punto-punto tra due nodi adiacenti uguale a **Tw**. Trascurando i tempi di computazione, il tempo impiegato da un messaggio inviato dal nodo 0 al nodo 5 è uguale **in media** a:
    a. Tw
    b. 5 * Tw
    c. 2 * Tw
    d. Tw * log (8)

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

```
void* run(void * arg){
    int* p = (int*)arg;
    sleep(1);
    sleep(3-*p);
    return NULL;
}
int main(int arg, char* argv[])
{
    pthread_t thid[3];
    int i=0;
    while(i<3){
        sleep(1);
        pthread_create(&thid[2-i], NULL, &run, &i);
        i++;
    }
    for(int i = 0; i< 3; i++)
        pthread_join(thid[i], NULL);
}
```

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

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

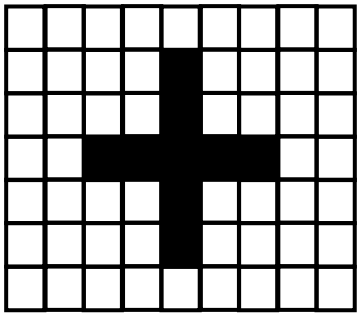
5. (*fino a 8 punti*) Si vuole convertire una esecuzione sincrona di una funzione in modalità asincrona utilizzando la libreria Posix threads. Si consideri il seguente codice:

```
#define timeout XXX

....

void doSomething(){

}

void startAsynch(){
    ....
}

void waitEnd(int timeout){
    ....
}

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

    startAsynch();

    doInTheMeanTime();

    waitEnd(timeout);

    return 0;
}
```

La funzione che deve essere eseguita è `doSomething()` (si consideri tale funzione già implementata). Nel `main` non si chiama esplicitamente `doSomething()` ma viene chiamata la funzione `startAsynch()` che deve far partire l'esecuzione di `doSomething()` in modo asyncrono, ovvero in contemporanea all'esecuzione del main della funzione `doInTheMeanTime()` (si consideri anche tale funzione come già implementata). La funzione `waitEnd(int timeout)` dovrà quindi attendere fino a che l'esecuzione di `doSomething()` non sia terminata o non sia passato un tempo pari a `timeout` secondi dall'invocazione della `waitEnd()` stessa. Si fornisca una implementazione delle funzioni `startAsynch()` e `waitEnd(int timeout)` unitamente alla dichiarazione delle variabili e/o funzioni eventualmente necessarie.

6. (*fino a 4 punti*)

Si vuole realizzare un codice per l'esecuzione di un automa cellulare di dimensione NROWSxNCOLS in parallelo con MPI, nel quale il dominio dell'automa è partizionato sia sulle righe che sulle colonne (R_PARTITIONS sulla righe e C_PARTITIONS sulle colonne) e il vicinato da considerare è quello di Von-Neumann-di raggio 2, ovvero quello in figura:

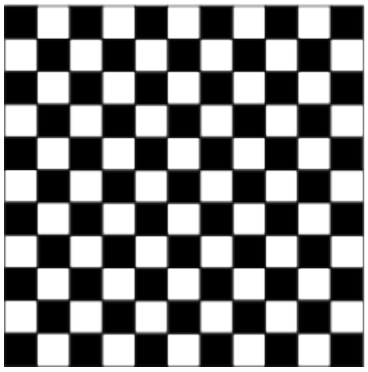


Ovvero, la cella centrale è quella sulla quale si calcola la funzione di transizione, il vicinato, in questo caso, consiste quindi delle 2 celle in alto, 2 celle in basso, 2 celle a sinistra e 2 a destra.
Volendo scambiare ogni bordo con **una sola** operazione send/receive, quale delle seguenti dichiarazioni MPI_Type_vector risulta corretta?

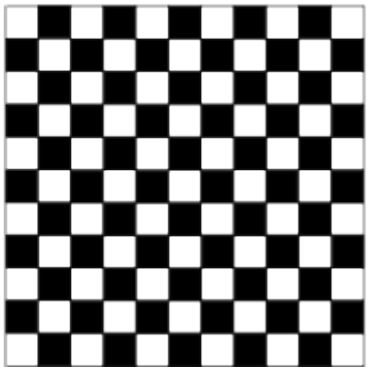
a.
```
MPI_Type_vector(2, NCOLS/C_PARTITIONS,
  NCOLS/C_PARTITIONS+2, MPI_INT, &UpDownDataType);
  MPI_Type_vector(NROWS/R_PARTITIONS, 2,
  NCOLS/C_PARTITIONS+2, MPI_INT, &LeftRightDataType);
```
b.
```
MPI_Type_vector(2, NCOLS/C_PARTITIONS,
  NCOLS/C_PARTITIONS+4, MPI_INT, &UpDownDataType);
  MPI_Type_vector(NROWS/R_PARTITIONS, 2,
  NCOLS/C_PARTITIONS+4, MPI_INT, &LeftRightDataType);
```
c.
```
MPI_Type_vector(4, NCOLS/C_PARTITIONS,
  NCOLS/C_PARTITIONS+4, MPI_INT, &UpDownDataType);
  MPI_Type_vector(NROWS/R_PARTITIONS, 4,
  NCOLS/C_PARTITIONS+4, MPI_INT, &LeftRightDataType);
```
d.
```
MPI_Type_vector(2, NCOLS/C_PARTITIONS,
  NCOLS/C_PARTITIONS+4, MPI_INT, &UpDownDataType);
  MPI_Type_vector(NROWS/R_PARTITIONS, 2,
  NROWS/R_PARTITIONS+4, MPI_INT, &LeftRightDataType);
```

7. (*fino a 7 punti*)

Si vuole realizzare uno scambio di dati tramite send/receive MPI tra 2 nodi entrambi ospitanti una matrice quadrata di interi, in modo da copiare, con riferimento alla figura di esempio fornita in seguito, il **contenuto delle celle indicate in nero** del nodo 0 nelle celle indicate in nero nel nodo 1:



Nodo 0



Nodo 1

Si consideri il seguente codice:

```
#define N 111
int rank;
int mat[N][N];

void exchChess(){
      ...
}

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

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

      exchChess();

      MPI_Finalize();

      return 0;
}
```

Si implementi la funzione `exchChess()` in modo da realizzare lo scambio di dati tramite **una sola** operazione di send/receive utilizzando opportunamente le funzionalità di MPI_Type_vector.
N.B.: non sarà considerata valida una implementazione nella quale viene trasferita tutta la matrice.

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);
```