Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 26 giugno 2023

Durata della Prova: 2,5 ore

1. (3 punti) Il tempo impiegato per ordinare un vettore di lunghezza **n** su **p** nodi utilizzando l'algoritmo bitonic sort è proporzionale a:

   a. $n/p + \log^2 p$
   b. $n/p \log (n) + \log^2 p$
   c. $n/p \log (n/p) + n/p \log^2 p$
   d. $n \log n$

2. (*3 punti*). Si supponga di avere un algoritmo di routing dimension-ordered di tipo two-step su mesh 2D non toroidale di dimensione N_X x N_Y, con tempo di comunicazione punto-punto tra due nodi adiacenti uguale a **Tw.** Trascurando i tempi di start-up, si indichi qual è il tempo massimo che impiega un messaggio che viene spedito da un nodo generico S ad un nodo generico D (con S!=D).
   a) Tw*Nx*Ny
   b) Tw*(Nx+Ny)
   c) 2*Tw*(Nx+Ny)-Tw
   d) Tw*log(Nx+Ny)

3. (5 *punti*) Il seguente codice:

```
pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel = 0;
int tot = XXXXX;


void* fuel_filling(void* arg) {
    int* p = (int*)arg;
    for (int i = 0; i < *p; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += tot/(*p);
        pthread_mutex_unlock(&mutexFuel);
        pthread_cond_signal(&condFuel);
    }
    return NULL;
}

void* car(void* arg) {
    int* p = (int*)arg;
    pthread_mutex_lock(&mutexFuel);
    while (fuel < tot/(*p))
       pthread_cond_wait(&condFuel, &mutexFuel);
```

```
    fuel -= tot/(*p);
    pthread_mutex_unlock(&mutexFuel);
    return NULL;
}

int main(int argc, char* argv[]) {
    ...
    int numFill = XXX;
    int numCar = XXX;
    pthread_create(&thfill, NULL, &fuel_filling, &numFill);
    for (int i = 0; i< numCar ; i++)
        pthread_create(&thcar, NULL, &car, &numCar);
    for (int i = 0; i< numCar ; i++)
        pthread_join(thcar, NULL);
    pthread_join(thcar, NULL);
    ...
}
```

Assumendo che il valore della variabile `tot` sia divisibile sia per `numFill` che per `numCar`, può risultare in una condizione di deadlock :

a. mai
b. se numFill<numCar
c. se numFill=numCar
d. se numFill>numCar

4. (*fino a 7 punti*) Si vuole implementare un meccanismo di timeout nella invocazione di funzioni utilizzando la libreria posix thread.
   In particolare, si considerino le seguenti funzioni:

```
void doTaskForSure(){
    ....
}

void doTaskOptionally(){
    ....
}
```

Come **già** implementate. In particolare, la funzione `doTaskForSure()` realizza un task che NON deve essere interrotto dal meccanismo di timeout, mentre `doTaskOptionally()` realizza un task che PUO' essere interrotto dal timeout.

Si implementi la funzione:

```
void doTasksWithTimeout(int timeoutms){
    ....
}
```

Che invochi le precedenti funzioni realizzando il meccanismo di timeout sopradescritto. La variabile `timeoutms` contiene il numero di millisecondi di durata del timeout. Il timeout si riferisce all'invocazione di entrambe le funzioni una dopo l'altra (prima `doTaskForSure()` e poi `doTaskOptionally()`).

P.s.: per realizzare il timeout è possibile avvalersi della funzione C `usleep(long time)` che arresta l'esecuzione del thread per `time` microsecondi.

5. (*fino a 5 punti*) Si consideri il seguente codice MPI:

```
int rank;
MPI_Datatype sendDT;
MPI_Datatype recDT;

#define dim XXX

int v[dim];
int m[dim/3][3];


void exchInfo(){
      MPI_Status status;
      if (rank ==0)
            MPI_Send(v, 1, sendDT, 1, 17, MPI_COMM_WORLD);
      else
            MPI_Recv(m, 1, recDT, 0, 17, MPI_COMM_WORLD, &status);
}

void prepareFibonacci(){
}
```

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


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

      if (rank == 0){
            prepareFibonacci();
      }

      Int a=XXX, b=XXX, c=XXX

      MPI_Type_vector(a, 3, b, MPI_INT, &sendDT);
      MPI_Type_commit(&sendDT);
      MPI_Type_vector(a, 3, c, MPI_INT, &recDT);
      MPI_Type_commit(&recDT);

      exchInfo();

      if (rank == 1)
            for(int i = 0; i < dim-2; i++)
                  printf("%d+%d=%d\n",m[i][0],m[i][1],m[i][2]);

      MPI_Finalize();

      return 0;
}
```

Il vettore v è riempito con la sequenza di fibonacci:
v={1,1,2,3,5,8,13,21,34….}

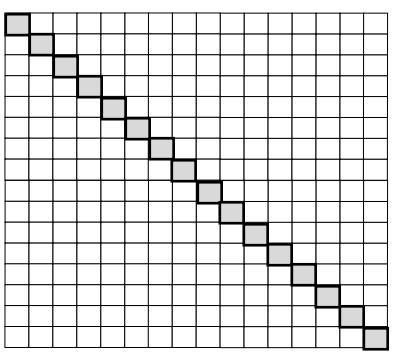
Tale codice, con i corretti valori per a, b e c, trasferisce il vettore v del nodo 0 nella matrice m del nodo 1 in modo tale che ogni riga di m contenga una tripletta nella quale la somma dei primi 2 elementi dia il terzo elemento. Ovvero, l'output del programma risulterebbe:
1+1=2
1+2=3
2+3=5
3+5=8
…

Quali sono i valori per a, b e c che garantiscono tale risultato?

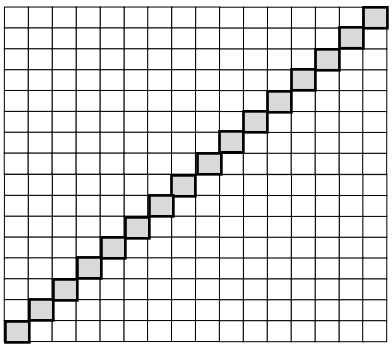
a. a=dim, b=3, c=3
b. a=dim-2, b=3, c=1
c. a=dim, b=1, c=1
d. a=dim-2, b=1, c=3

6. (*fino a 7 punti*)

Si realizzi uno scambio di dati tramite send/receive MPI tra 2 nodi entrambi ospitanti una matrice di interi. In particolare, utilizzando opportunamente `MPI_Type_vector`, si vuole trasferire il contenuto della diagonale principale della matrice del nodo 0 nella diagonale secondaria della matrice del nodo 1, con una sola operazione di send/receive (si veda la seguente figura)



Nodo 0



Nodo 1

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 il seguente codice:

```
#define NCOLS 200
#define NROWS 200
int rank;
int mat[NROWS][NCOLS];

...

void exchDiagonal(){
 ...
}

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

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

      exchDiagonal();

      MPI_Finalize();

      return 0;
}
```

Si fornisca una implementazione della funzione: `exchDiagonal()` 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);
```