Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 22 giugno 2022

Durata della Prova: 2 ore

1. (*3 punti*) In un sistema multi-processore, NON si hanno problemi di cache coherence se:
    a. Se processi diversi si riferiscono a locazione di memoria diverse
    b. Se tutti i processi eseguono solo operazioni di lettura
    c. Se i processi scambiano dati tra loro
    d. Se si è in un contesto di memoria condivisa

2. (*3 punti*) In un ipercubo di dimensione D con p nodi, la distanza tra due nodi qualsiasi è uguale:
    a. Prodotto del risultato dell'operazione OR effettuata tra i bit corrispondenti della loro rappresentazione binaria
    b. log p
    c. Somma del risultato dell'operazione XOR effettuata tra i bit corrispondenti della loro rappresentazione binaria
    d. p

3. (*fino a 3 punti*) Si supponga che il tempo di comunicazione di una rete ad ipercubo composta da **N** nodi, per una comunicazione punto-punto tra due nodi adiacenti, sia uguale a **Tw**. Trascurando i tempi di computazione, si indichi quale è il tempo necessario per ordinare una sequenza bitonica.

4. (*fino a 5 punti*) Il seguente codice:

```
int numThreads=10;

void* run(void * arg){
      sleep(1);
      int i = (int)arg;
      printf("i=%d\n",i);
      return NULL;
}


int main(int arg, char* argv[])
{
      pthread_t thid[numThreads];

      for(int i=0; i<numThreads;i++){

            int ris = pthread_create(&thid[i], NULL, &run, &i);
            if (ris){
                  printf("errore creazione thread\n");
                  exit(-1);
            }
      }
      for(int i=0; i<numThreads;i++)
            pthread_join(thid[i], NULL);
}
```

Stamperà come primo numero:

a. 0
b. 10
c. Dà errore
d. 1

5. (*fino a 6 punti*) Si vuole realizzare una funzionalità “barrier” utlizzando la libreria posix thread (senza utilizzare la specifica funzionalità “barrier” offerta della libreria) che permetta ai thread di sincronizzarsi in uno specifico punto. In particolare, si consideri il seguente codice:

```
int nThread = 5;

.... dichiarazione variabili


void initBarrier(){
    ....
}

void barrier(){
    ....
}

void destroyBarrier(){
    ....
}

void* threadFunc(void* arg) {
    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);

    }

    for (int i = 0; i < nThread; i++) {
        pthread_join(th[i], NULL);
    }
    destroyBarrier();
    return 0;
}
```

Implementare le funzioni `initBarrier()`,`barrier()` e `destroyBarrier()` (ed eventuali dichiarazioni di variabili) in modo tale che i threads si sincronizzino alla chiamata di barrier. In particolare, l’output atteso del codice di esempio prevederebbe quindi prima 5 stampe “inizio” seguite da 5 stampe “fine”.

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

```
if (rank == 0) {
    MPI_Send(data, MAXSIZE, MPI_INT, 1, 17, MPI_COMM_WORLD);
    MPI_Recv(data, MAXSIZE, MPI_INT, 1, 23, MPI_COMM_WORLD, &status);
} else if (rank==1) {
    MPI_Send(data, MAXSIZE, MPI_INT, 0, 23, MPI_COMM_WORLD);
    MPI_Recv(data, MAXSIZE, MPI_INT, 0, 17, MPI_COMM_WORLD, &status);
}
```

Possono esservi problemi di deadlock:

a. Quando la dimensione del dato trasferito è grande
b. Quando la dimensione del dato trasferito è piccolo
c. In base all'ordine di esecuzione dei processi
d. Se il contesto di esecuzione è a memoria condivisa o distribuita

7. (*fino a 6 punti*) Si vuole realizzare una funzionalità "barrier" utlizzando la libreria MPI (senza utilizzare la specifica funzionalità "barrier" offerta della libreria) che permetta ai processi di sincronizzarsi in uno specifico punto. In particolare, si consideri il seguente codice:

```
.... dichiarazione variabili


void barrier(int rank, int size){
      …

}

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

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

  barrier(rank,size);

  MPI_Finalize();

  return 0;

}
```

Implementare la funzione `barrier()` in modo tale che i processi MPI si sincronizzino alla chiamata di tale funzione.

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);
```