Prova Scritta di Algoritmi Paralleli e Sistemi Distribuiti

Appello del 13 luglio 2022

Durata della Prova: 2 ore

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

```
int a;
int b;

/* The two following functions are running concurrently: */
int threadA(void)
{
    int s = 0;
    int i;
    for (i = 0; i < 1000000; ++i)
        s += a;
    return s;
}
void threadB(void)
{
    int i;
    for (i = 0; i < 1000000; ++i)
        b = b + 1;
}
```

Presenterà problematiche di false sharing:

a. Sempre
b. Mai
c. In base alla posizione in memoria di a e b
d. In base all'ordine di esecuzione

2. (3 *punti*) Ricordando la formula dell'efficienza di un programma parallelo in termini di overhead (To) e tempo sequenziale (Ts), e fissato le dimensioni di un problema, all'aumentare del numero di processi l'**efficienza**:

   a. Aumenta sempre
   b. Diminuisce sempre
   c. Dipende dalla frazione seriale del problema
   d. Dipende dal problema specifico

3. (fino a *4 punti*) Si scriva un programma in OpenMP che faccia il prodotto scalare di due vettori in modo ottimale (master slave, con padding, etc), evitando di utilizzare operazioni di riduzione

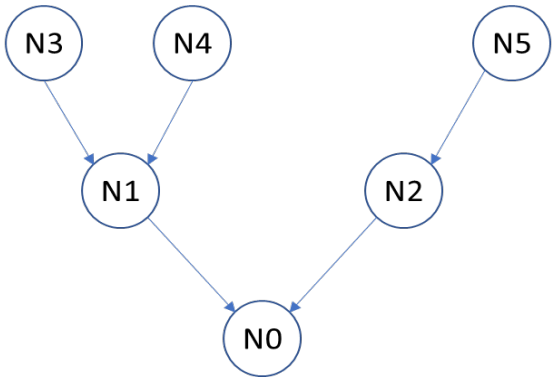
4. (4 *punti*) L'esecuzione del seguente programma:

```
void* run(void* arg) {
      int* p = (int*)arg;
      sleep(*p);

}

int main(int argc, char* argv[]) {
      pthread_t thid[3];
      int s[3];
      for(int i=0;i<3;i++){
            s[i]=i+1;
            pthread_create(&thid[i], NULL, &run, &s[i]);
      }
      sleep(2);
      for(int i=0;i<3;i++)
            pthread_join(thid[i], NULL);

   return 0;
}
```

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

   a. Poco più di 8 secondi
   b. Poco più di 5 secondi
   c. Poco più di 3 secondi
   d. Poco più di 2 secondi

5. (fino a 6 *punti*) Si vuole realizzare un sistema multithread che esegua in parallelo dei job il cui ordine di esecuzione dipenda da un albero di precedenza. Si consideri, come **esempio**, l'albero mostrato nella seguente figura:



Ogni nodo dell'albero corrisponde ad un job da eseguire. L'esecuzione parallela, **nel caso dell'albero in figura**, prevederebbe quindi i seguenti vincoli: (i) il job del nodo N1 può eseguire solo quando sono terminati i job dei nodi N3 ed N4, (ii) Il job del nodo N2 può eseguire solo quando è terminato quello di N5, (iii) il job di N0 può eseguire solo quando sono finiti i job dei nodi N1 ed N2.

Si consideri il seguente codice:

```
struct Node{
      int id;
      Node* left;
      Node* right;
      void job()
};

Node* tree;

void* executeTree(void* arg) {
      ...
}

int main(int argc, char* argv[]) {
    tree=buildTree();
    pthread_t thid;
    pthread_create(&thid, NULL, &executeTree, tree);
    pthread_join(thid, NULL);
    return 0;
}
```

Si consideri la funzione `buildTree()` come già realizzata. In particolare, tale funzione costruisce un albero di oggetti `Node` dove il metodo `job()` è già definito.

Si implementi la funzione `executeTree(void* arg)` in modo da eseguire in parallelo i metodi `job()` dei nodi dell'albero costruito da `buildTree()`, rispettando i vincoli di precedenza definiti dall'albero stesso. E' lasciato allo studente la possibilità di definire ulteriori variabili e/o funzioni ausiliarie.

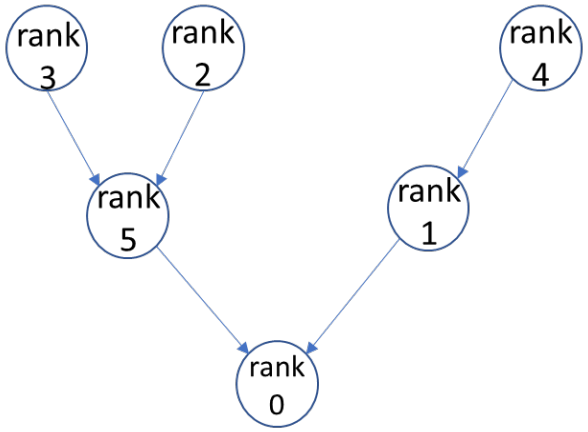
6. (4 *punti*) Data la seguente porzione di programma MPI:

```
for(int size=MINSIZE;size<MAXSIZE;size*=10)
      if (rank == 0) {
            MPI_Request request;
            data[0] = 2;
            MPI_Isend(&data[0], size, MPI_INT, 1, 17, MPI_COMM_WORLD, &request);
                  data[0] = 6;
                  MPI_Status status;
                  MPI_Wait(&request, &status);
                  data[0] = 8;

            } else if (rank==1) {
                  MPI_Status status;
                  MPI_Recv(data, size, MPI_INT, 0, 17, MPI_COMM_WORLD, &status);
                  printf("%d", data[0]);
            }
```

Quale dei seguenti output non può essere mai stampato?

a. 22222666
b. 66666666
c. 22222222
d. 66622222

7. (fino a 6 *punti*) Si vuole realizzare un programma MPI che esegua in multi-processo dei job il cui ordine di esecuzione dipenda da un albero di precedenza. Ogni nodo è associato ad uno specifico 'rank' di processo MPI. Si consideri, come **esempio**, l'albero mostrato nella seguente figura:



Ad ogni nodo dell'albero corrisponde un job da eseguire. L'esecuzione MPI, **nel caso dell'albero in figura**, prevederebbe quindi i seguenti vincoli: (i) il job del nodo associato al processo di rank 5 può eseguire solo quando sono terminati i job dei

nodi associati ai processi di rank 3 e 2, (ii) Il job del nodo associato al rank 1 può eseguire quando è terminato quello di rank 4, (iii) il job del nodo associato al rank 0 può eseguire solo quando sono finiti i job dei rank 1 e 2.

Si consideri il seguente codice:

```
int myrank;

struct Node{
      int rank;
      Node* left;
      Node* right;
      void job()
};

Node* tree;

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

      tree=buildTree();
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

      ...

      MPI_Finalize();
      return 0;
}
```

Si consideri la funzione `buildTree()` come già realizzata. In particolare, tale funzione costruisce un albero di oggetti `Node` dove il metodo `job()` è definito, e dove ogni nodo è associato ad un rank diverso. Si supponga, altresì, che il numero di processi MPI sia pari, almeno, al numero di nodi.

Si implementi la parte mancante della funzione `main` in modo che i metodi `job()` dei nodi dell'albero costruito da `buildTree()` siano eseguiti rispettando i vincoli di precedenza definiti dall'albero stesso. E' lasciato allo studente la possibilità di definire ulteriori variabili e/o funzioni ausiliarie.

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);
```