# Lab 6d - factorize con MPI

https://elly2023.smfi.unipr.it/mod/page/view.php?id=5404

**OBIETTIVO**
Parallelizzazione di mpi_factorize.c tramite scomposizione del dominio 1D in sottodomini.

## Attività svolte

Dopo aver creato la directory di lavoro "*mkdir ~/HPC2324/mpi/factorize/*", ho copiato al suo interno i file da "*cp /hpc/home/roberto.alfieri/SHARE/mpi/factorize/* .*"

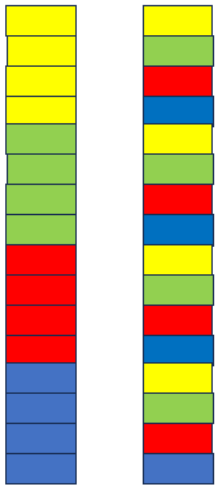
**Nota**: il dominio dei possibili numeri primi da testare viene suddiviso in blocchi di dimensione fissa; ogni blocco è identificato da un indice ( block_idx ). I numeri di un blocco vengono testati dalla funzione block_factorize (block_idx), che ritorna 1 se trova il numero, altrimenti torna 0.

*Ripartizione a blocchi contigui*

```
for (block_idx=lastBlock; block_idx >= firstBlock ; block_idx--)
{
    if ( block_factorize (block_idx) )
    {
      // stampa il numero trovato
      break;
    }
}
```

*Ripartizione interleaved*

```
for (block_idx=block_addr_size*MPIsize-MPIrank-1; block_idx >= 0 ; block_idx=block_idx-MPIsize)
{
    if ( block_factorize (block_idx) )
    {
      // stampa il numero trovato
      break;
    }
}
```



Per prima cosa ho modificato lo script mpi_factorize.slurm inserendo il modulo da me generato, dopodiche ho lanciato il job in modo da compilare ed eseguire il programma mpi_factorize.c

Dopodiché ho creato una copia, rinominandola in mpi_factorize_term.c, dove il programma viene ottimizzato terminando subito quando un task trova il numero primo (nella versione precedente, mpi_factorize.c, il task che trova il numero interrompe la ricerca ma gli altri task completano il processamento di tutti i blocchi assegnati).

**mpi_factorize_term.c**

```
/*
mpi_factorize_term.c
OTTIMIZZAZIONE: Il programma termina quando un task trova il numero primo
*/

/*
module load intel impi
mpicc mpi_factorize.c   -lcrypto -lm

openssl genrsa -out rsa_key.pem 68
openssl rsa -in rsa_key.pem -modulus -noout

mpirun -n 1  a.out  -m B81915BC0A2222F4B   -a 4    # modulus 68 bit - prime 34 bit (4 addr + 28 block)
*/

#include <stdio.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/bn.h>
#include <math.h>
#include <mpi.h>


//void status() {}
void options(int argc, char * argv[]) ;
int block_factorize (unsigned long int);


BIGNUM *P, *Q, *M, *F, *ZERO,  *ONE, *TWO, *BLOCK_DIM_SIZE, *BLOCK_DIM_BIT ;
BN_CTX *ctx;
int block_idx=0, block_found;
unsigned long int  modulus_bit;      // numero di bit del modulo
unsigned long int  prime_bit;        // numedo di bit del numero primo ( modulus_bit/2)
unsigned long int  block_addr_bit;   // numero di bit dell'indirizzo dei blocchi
unsigned long int  block_dim_bit;    // numero di bit di un blocco
unsigned long int  block_addr_size;  // numero di blocchi (2^ block_addr_bit)
unsigned long int  block_dim_size;   // dimensione di un blocco (2^block_dim_bit)
unsigned long int  block_addr_start; // indirizzo del primo blocco da processare
unsigned long int  lastBlock, firstBlock;  // indirizzo del primo e ultimo blocco
char m[256];                         // modulus



// variables required by MPI
int MPIrank=0, MPIsize=1;
int block_addr_size_global, flag, received;



int main(int argc, char *argv[])
{
        int i;
        float t1, t2, t3, t4;  // timer

// BN setting
        P = BN_new();     // prime number
        Q = BN_new();     // prime number
        M = BN_new();     // modulw = p x q
        F = BN_new();     // found number
        ZERO           = BN_new();  //  0
        ONE            = BN_new();  //  1
        TWO            = BN_new();  //  2
        BLOCK_DIM_BIT  = BN_new();  // quanti bit per blocco
        BLOCK_DIM_SIZE = BN_new();  // dimensione blocco
        ctx = BN_CTX_new();
        BN_set_word(ZERO,0);
        BN_set_word(ONE,1);
        BN_set_word(TWO,2);



// Default values
        block_addr_bit = 4;

// Options management
        options(argc, argv);

// MPI init
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &MPIrank);
        MPI_Comm_size(MPI_COMM_WORLD, &MPIsize);
        MPI_Request request;
        MPI_Status status;

// Domain decomposition
        block_addr_size_global = pow(2,block_addr_bit);
        block_addr_size = block_addr_size_global/MPIsize;


        BN_hex2bn(&M, m);
        modulus_bit     = strlen(m)*4;
        prime_bit       = modulus_bit/2;
        block_dim_bit   = prime_bit - block_addr_bit;
        block_dim_size  = pow(2,block_dim_bit);

        BN_set_word(BLOCK_DIM_BIT,block_dim_bit);                 // bits per block
        BN_exp(BLOCK_DIM_SIZE, TWO , BLOCK_DIM_BIT,ctx);
```

```
        block_addr_start= block_addr_size*MPIsize-1;

        if (MPIrank==0)
        {
            printf("\n# Modulus: ");
            BN_print_fp(stdout,M);
            printf(" %d bits,  prime %d bit, address %d bit, block %d bit, start %d \n\n", modulus_bit, prime_bit,
block_addr_bit, block_dim_bit,  block_addr_start );
        }

     lastBlock=block_addr_size*(MPIrank+1)-1;
     firstBlock=block_addr_size*MPIrank;
     printf("Task %d/%d - block %d - range %d-%d: ", MPIrank, MPIsize, block_addr_size, firstBlock, lastBlock);
        MPI_Barrier(MPI_COMM_WORLD);

     //Attiva la ricezione asincorna non bloccante della comunicazione da altri task
     MPI_Irecv(&received, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &request);

        t1=MPI_Wtime();

        // Ripartizione a blocchi contigui
        for (block_idx=lastBlock; block_idx >= firstBlock ; block_idx--)
     {
          // ogni task prima di analizzare un nuovo blocco testa l'eventuale ricezione del messaggio
            MPI_Test (&request, &flag, MPI_STATUS_IGNORE);
            if (flag) break;  //messaggio arrivato

     t2=MPI_Wtime();
            printf ("Task %d/%d - block %d processing  \n", MPIrank, MPIsize, block_idx);
            fflush(stdout);

            if ( block_factorize (block_idx) )
            {
     // Invia un messaggio a tutti i task
     for(i=0; i<MPIsize; i++)
     {
        if (MPIrank == i) continue;
                    MPI_Isend(&flag, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
     }

     t3=MPI_Wtime();

                // stampa in numero trovato
     printf("Task %d/%d - block %d - FOUND: ", MPIrank, MPIsize, block_idx);
     BN_print_fp(stdout,F);
                printf(" %.1f sec. \n", t3-t2);
                break;
            }
     else
             {
                t3=MPI_Wtime();
                printf("Task %d/%d - block %d - NOT found - %.1f sec.\n",  MPIrank, MPIsize, block_idx, t3-t2);
             }
     }

        // Ripartizione interleaved
        /*for (block_idx=block_addr_start-MPIrank; block_idx >= 0 ; block_idx=block_idx-MPIsize)
        {
             t2=MPI_Wtime();
             printf ("Task %d/%d - block %d processing  \n", MPIrank, MPIsize, block_idx);
             fflush(stdout);
             if ( block_factorize (block_idx) )
             {
                 t3=MPI_Wtime();
                 printf("Task %d/%d - block %d - FOUND: ", MPIrank, MPIsize, block_idx);
                 BN_print_fp(stdout,F);
                 printf(" %.1f sec. \n", t3-t2);
                 break;
             }
             else
             {
                t3=MPI_Wtime();
                printf("Task %d/%d - block %d - NOT found - %.1f sec.\n",  MPIrank, MPIsize, block_idx, t3-t2);
             }
        }*/

        t4=MPI_Wtime();
```

```
        printf("Task %d/%d  exiting,  time %.1f sec.  \n",  MPIrank, MPIsize, t4-t1);

        MPI_Finalize();
        return (0);
}



int block_factorize (unsigned long int block_addr)
{

    BIGNUM *R, *X, *Y, *BLOCK_IDX, *BLOCK_ADDR;
    BN_CTX *ctx2;
    ctx2 = BN_CTX_new();
    R       = BN_new();  // resto della divisione
    X       = BN_new();  // indice del  blocco
    Y       = BN_new();  // ultimo numero del  blocco
    BLOCK_IDX       = BN_new();  // ultimo numero del  blocco
    BLOCK_ADDR      = BN_new();  // Block Address

    BN_set_word(R,0);
    BN_set_word(X,1);
    BN_set_word(Y,1);
    BN_set_word(BLOCK_ADDR, block_addr);
    BN_mul(X, BLOCK_ADDR, BLOCK_DIM_SIZE,ctx2);  // x = i  block_size
    BN_add(X,X,ONE);                             // x = x+1
    BN_add(Y,X,BLOCK_DIM_SIZE);                  // y = x + block_size
    BN_sub(Y,Y,TWO);                             // y = y - 2
    BLOCK_IDX=BN_dup(X);

    while ( BN_cmp(BLOCK_IDX,Y) )
    {
        BN_add(BLOCK_IDX,BLOCK_IDX,TWO);
        BN_mod(R,M,BLOCK_IDX, ctx2);
        if ( BN_is_zero(R) )
        {
            F=BN_dup(BLOCK_IDX);
            return (1); // FOUND
        }
    }

    return (0);  // NOT FOUND
}

/*********************************************/

void options(int argc, char * argv[])
{
    int i;
    while ( (i = getopt(argc, argv, "a:b:m:h")) != -1)
    {
        switch (i)
        {
            case 'a':  block_addr_bit   = strtol(optarg, NULL, 10);  break;
            case 'b':  modulus_bit      = strtol(optarg, NULL, 10);  break;
            case 'm':  strcpy(m,optarg);  break;
            case 'h':  printf ("\n%s [-b modulus_bit] [-m modulus] [-a block_addr_bit] [-h]",argv[0]);  exit(1);
            default:   printf ("\n%s [-b modulus_bit] [-m modulus] [-a block_addr_bit] [-h]",argv[0]);  exit(1);
        }
    }
}
```

## mpi_factorize_term.txt

```
[martina.genovese@ui01 factorize]$ more mpi_factorize_term.txt
Task 1/4 - block 64 - range 64-127: Task 2/4 - block 64 - range 128-191:
# Modulus: B3D5FA9A09A7D7F39Task 3/4 - block 64 - range 192-255: 5 72 bits,  prime 36 bit, address 8 bit, block
28 bit, start 255

Task 0/4 - block 64 - range 0-63: Task 2/4 - block 191 processing
Task 0/4 - block 63 processing
Task 1/4 - block 127 processing
Task 3/4 - block 255 processing
Task 1/4 - block 127 - NOT found - 14.9 sec.
Task 1/4 - block 126 processing
Task 2/4 - block 191 - NOT found - 15.1 sec.
Task 2/4 - block 190 processing
```

```
Task 0/4 - block 63 - NOT found - 16.1 sec.
Task 0/4 - block 62 processing
Task 3/4 - block 255 - NOT found - 16.1 sec.
Task 3/4 - block 254 processing
Task 1/4 - block 126 - NOT found - 14.8 sec.
Task 1/4 - block 125 processing
Task 2/4 - block 190 - NOT found - 15.0 sec.
Task 2/4 - block 189 processing
Task 3/4 - block 254 - NOT found - 15.8 sec.
Task 3/4 - block 253 processing
Task 0/4 - block 62 - NOT found - 15.9 sec.
Task 0/4 - block 61 processing
Task 1/4 - block 125 - NOT found - 14.9 sec.
Task 1/4 - block 124 processing
Task 2/4 - block 189 - NOT found - 14.8 sec.
Task 2/4 - block 188 processing
Task 3/4 - block 253 - NOT found - 15.9 sec.
Task 3/4 - block 252 processing
Task 0/4 - block 61 - NOT found - 15.9 sec.
Task 0/4 - block 60 processing
Task 1/4 - block 124 - NOT found - 14.8 sec.
Task 1/4 - block 123 processing
Task 2/4 - block 188 - NOT found - 15.4 sec.
Task 2/4 - block 187 processing
Task 3/4 - block 252 - NOT found - 15.6 sec.
Task 3/4 - block 251 processing
Task 0/4 - block 60 - NOT found - 15.9 sec.
Task 0/4 - block 59 processing
Task 1/4 - block 123 - NOT found - 14.9 sec.
Task 1/4 - block 122 processing
Task 2/4 - block 187 - NOT found - 14.8 sec.
Task 2/4 - block 186 processing
Task 3/4 - block 251 - NOT found - 15.8 sec.
Task 3/4 - block 250 processing
Task 0/4 - block 59 - NOT found - 16.1 sec.
Task 0/4 - block 58 processing
Task 1/4 - block 122 - NOT found - 14.8 sec.
Task 1/4 - block 121 processing
Task 2/4 - block 186 - NOT found - 15.0 sec.
Task 2/4 - block 185 processing
Task 3/4 - block 250 - NOT found - 15.6 sec.
Task 3/4 - block 249 processing
Task 0/4 - block 58 - NOT found - 15.8 sec.
Task 0/4 - block 57 processing
Task 1/4 - block 121 - NOT found - 14.8 sec.
Task 1/4 - block 120 processing
Task 2/4 - block 185 - NOT found - 14.9 sec.
Task 2/4 - block 184 processing
Task 3/4 - block 249 - NOT found - 15.9 sec.
Task 3/4 - block 248 processing
Task 0/4 - block 57 - NOT found - 16.0 sec.
Task 0/4 - block 56 processing
Task 1/4 - block 120 - NOT found - 15.0 sec.
Task 1/4 - block 119 processing
Task 2/4 - block 184 - NOT found - 15.0 sec.
Task 2/4 - block 183 processing
Task 3/4 - block 248 - NOT found - 15.4 sec.
Task 3/4 - block 247 processing
Task 0/4 - block 56 - NOT found - 15.6 sec.
Task 0/4 - block 55 processing
Task 1/4 - block 119 - NOT found - 14.8 sec.
Task 1/4 - block 118 processing
Task 2/4 - block 183 - NOT found - 14.9 sec.
Task 2/4 - block 182 processing
Task 3/4 - block 247 - NOT found - 15.8 sec.
Task 3/4 - block 246 processing
Task 0/4 - block 55 - NOT found - 16.0 sec.
Task 0/4 - block 54 processing
Task 1/4 - block 118 - NOT found - 14.8 sec.
Task 1/4 - block 117 processing
Task 2/4 - block 182 - NOT found - 15.1 sec.
Task 2/4 - block 181 processing
Task 3/4 - block 246 - NOT found - 15.4 sec.
Task 3/4 - block 245 processing
Task 0/4 - block 54 - NOT found - 15.5 sec.
Task 0/4 - block 53 processing
Task 1/4 - block 117 - NOT found - 14.8 sec.
Task 1/4 - block 116 processing
Task 2/4 - block 181 - NOT found - 15.0 sec.
Task 2/4 - block 180 processing
Task 3/4 - block 245 - NOT found - 15.9 sec.
Task 3/4 - block 244 processing
Task 0/4 - block 53 - NOT found - 15.8 sec.
Task 0/4 - block 52 processing
```

```
Task 1/4 - block 116 - NOT found - 14.8 sec.
Task 1/4 - block 115 processing
Task 2/4 - block 180 - NOT found - 15.2 sec.
Task 2/4 - block 179 processing
Task 3/4 - block 244 - NOT found - 15.4 sec.
Task 3/4 - block 243 processing
Task 0/4 - block 52 - NOT found - 15.5 sec.
Task 0/4 - block 51 processing
Task 1/4 - block 115 - NOT found - 14.8 sec.
Task 1/4 - block 114 processing
Task 2/4 - block 179 - NOT found - 15.0 sec.
Task 2/4 - block 178 processing
Task 3/4 - block 243 - NOT found - 15.5 sec.
Task 3/4 - block 242 processing
Task 0/4 - block 51 - NOT found - 15.8 sec.
Task 0/4 - block 50 processing
Task 1/4 - block 114 - NOT found - 14.9 sec.
Task 1/4 - block 113 processing
Task 2/4 - block 178 - NOT found - 15.4 sec.
Task 2/4 - block 177 processing
Task 3/4 - block 242 - NOT found - 15.8 sec.
Task 3/4 - block 241 processing
Task 0/4 - block 50 - NOT found - 16.2 sec.
Task 0/4 - block 49 processing
Task 1/4 - block 113 - NOT found - 14.8 sec.
Task 1/4 - block 112 processing
Task 2/4 - block 177 - NOT found - 15.2 sec.
Task 2/4 - block 176 processing
Task 3/4 - block 241 - NOT found - 15.9 sec.
Task 3/4 - block 240 processing
Task 1/4 - block 112 - NOT found - 14.9 sec.
Task 1/4 - block 111 processing
Task 0/4 - block 49 - NOT found - 15.8 sec.
Task 0/4 - block 48 processing
Task 2/4 - block 176 - NOT found - 15.2 sec.
Task 2/4 - block 175 processing
Task 3/4 - block 240 - NOT found - 15.4 sec.
Task 3/4 - block 239 processing
Task 1/4 - block 111 - NOT found - 14.8 sec.
Task 1/4 - block 110 processing
Task 0/4 - block 48 - NOT found - 16.1 sec.
Task 0/4 - block 47 processing
Task 2/4 - block 175 - NOT found - 15.2 sec.
Task 2/4 - block 174 processing
Task 3/4 - block 239 - FOUND: EF7F8032D 7.9 sec.
Task 3/4  exiting,  time 258.8 sec.
Task 1/4 - block 110 - NOT found - 14.6 sec.
Task 1/4  exiting,  time 266.2 sec.
Task 0/4 - block 47 - NOT found - 16.0 sec.
Task 0/4  exiting,  time 269.9 sec.
Task 2/4 - block 174 - NOT found - 15.0 sec.
Task 2/4  exiting,  time 271.2 sec.
```

Infine ho creato una copia rinominata mpi_factorize_chkpt.c, dove il programma viene ottimizzato implementando la gestione dei check-point, ovvero la possibilità di interrompere l'esecuzione e riprenderla in un momento successivo partendo dalla stato dell'esecuzione al momento dell'interruzione.

**mpi_factorize_chkpt.c**

```c
/*
mpi_factorize_chkpt.c
OTTIMIZZAZIONE: evoluzione del programma in cui viene gestito il checkpointing consentendo di specificare (opzione -s) il blocco da cui iniziare il processamento ( block_addr_start )
*/

/*
module load intel impi
mpicc mpi_factorize.c   -lcrypto -lm

openssl genrsa -out rsa_key.pem 68
openssl rsa -in rsa_key.pem -modulus -noout

mpirun -n 1  a.out  -m B81915BC0A2222F4B   -a 4    # modulus 68 bit - prime 34 bit (4 addr + 28 block)
*/

#include <stdio.h>
#include <unistd.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <openssl/bn.h>
#include <math.h>
#include <mpi.h>


//void status() {}
void options(int argc, char * argv[]) ;
int block_factorize (unsigned long int);

BIGNUM *P, *Q, *M, *F, *ZERO,  *ONE, *TWO, *BLOCK_DIM_SIZE, *BLOCK_DIM_BIT ;
BN_CTX *ctx;
int block_idx=0, block_found;
unsigned long int  modulus_bit;      // numero di bit del modulo
unsigned long int  prime_bit;        // numedo di bit del numero primo ( modulus_bit/2)
unsigned long int  block_addr_bit;   // numero di bit dell'indirizzo dei blocchi
unsigned long int  block_dim_bit;    // numero di bit di un blocco
unsigned long int  block_addr_size;  // numero di blocchi (2^ block_addr_bit)
unsigned long int  block_dim_size;   // dimensione di un blocco (2^block_dim_bit)
unsigned long int  block_addr_start=0; // indirizzo del primo blocco da processare
unsigned long int  lastBlock, firstBlock;  // indirizzo del primo e ultimo blocco
char m[256];                         // modulus



// variables required by MPI
int MPIrank=0, MPIsize=1;
int block_addr_size_global, flag, received;



int main(int argc, char *argv[])
{
        int i;
        float t1, t2, t3, t4;  // timer

// BN setting
        P = BN_new();     // prime number
        Q = BN_new();     // prime number
        M = BN_new();     // modulw = p x q
        F = BN_new();     // found number
        ZERO           = BN_new();  //  0
        ONE            = BN_new();  //  1
        TWO            = BN_new();  //  2
        BLOCK_DIM_BIT  = BN_new();  // quanti bit per blocco
        BLOCK_DIM_SIZE = BN_new();  // dimensione blocco
        ctx = BN_CTX_new();
        BN_set_word(ZERO,0);
        BN_set_word(ONE,1);
        BN_set_word(TWO,2);



// Default values
        block_addr_bit = 4;

// Options management
        options(argc, argv);

// MPI init
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &MPIrank);
        MPI_Comm_size(MPI_COMM_WORLD, &MPIsize);
        MPI_Request request;
        MPI_Status status;

// Domain decomposition
        block_addr_size_global = pow(2,block_addr_bit);
        block_addr_size = block_addr_size_global/MPIsize;


        BN_hex2bn(&M, m);
        modulus_bit     = strlen(m)*4;
        prime_bit       = modulus_bit/2;
        block_dim_bit   = prime_bit - block_addr_bit;
        block_dim_size  = pow(2,block_dim_bit);


        BN_set_word(BLOCK_DIM_BIT,block_dim_bit);                 // bits per block
        BN_exp(BLOCK_DIM_SIZE, TWO , BLOCK_DIM_BIT,ctx);
```

```
        if(!block_addr_start)
    block_addr_start= block_addr_size*MPIsize-1;

        if (MPIrank==0)
        {
            printf("\n# Modulus: ");
            BN_print_fp(stdout,M);
            printf(" %d bits,  prime %d bit, address %d bit, block %d bit, start %d \n\n", modulus_bit, prime_bit,
block_addr_bit, block_dim_bit,  block_addr_start );
        }

        lastBlock=block_addr_size*(MPIrank+1)-1;
    firstBlock=block_addr_size*MPIrank;
    printf("Task %d/%d - block %d - range %d-%d: ", MPIrank, MPIsize, block_addr_size, firstBlock, lastBlock);
        MPI_Barrier(MPI_COMM_WORLD);

    //Attiva la ricezione asincorna non bloccante della comunicazione da altri task
    MPI_Irecv(&received, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &request);

        t1=MPI_Wtime();

        // Ripartizione a blocchi contigui
        for (block_idx=lastBlock; block_idx >= firstBlock ; block_idx--)
    {
         // ogni task prima di analizzare un nuovo blocco testa l'eventuale ricezione del messaggio
            MPI_Test (&request, &flag, MPI_STATUS_IGNORE);
            if (flag) break;  //messaggio arrivato

    t2=MPI_Wtime();
            printf ("Task %d/%d - block %d processing  \n", MPIrank, MPIsize, block_idx);
            fflush(stdout);

            if ( block_factorize (block_idx) )
            {
    for(i=0; i<MPIsize; i++)
    {
           // Invia un messaggio a tutti i task
            if (MPIrank == i) continue;
                  MPI_Isend(&flag, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
    }

               t3=MPI_Wtime();

               // stampa in numero trovato
    printf("Task %d/%d - block %d - FOUND: ", MPIrank, MPIsize, block_idx);
    BN_print_fp(stdout,F);
               printf(" %.1f sec. \n", t3-t2);
               break;
            }
    else
            {
               t3=MPI_Wtime();
               printf("Task %d/%d - block %d - NOT found - %.1f sec.\n",  MPIrank, MPIsize, block_idx, t3-t2);
            }
    }

        // Ripartizione interleaved
        /*for (block_idx=block_addr_start-MPIrank; block_idx >= 0 ; block_idx=block_idx-MPIsize)
        {
             t2=MPI_Wtime();
             printf ("Task %d/%d - block %d processing  \n", MPIrank, MPIsize, block_idx);
             fflush(stdout);
             if ( block_factorize (block_idx) )
             {
                 t3=MPI_Wtime();
                 printf("Task %d/%d - block %d - FOUND: ", MPIrank, MPIsize, block_idx);
                 BN_print_fp(stdout,F);
                 printf(" %.1f sec. \n", t3-t2);
                 break;
             }
             else
             {
                t3=MPI_Wtime();
                printf("Task %d/%d - block %d - NOT found - %.1f sec.\n",  MPIrank, MPIsize, block_idx, t3-t2);
             }
        }*/

        t4=MPI_Wtime();
```

```
        printf("Task %d/%d  exiting,  time %.1f sec.  \n",  MPIrank, MPIsize, t4-t1);

        MPI_Finalize();
        return (0);
}



int block_factorize (unsigned long int block_addr)
{

    BIGNUM *R, *X, *Y, *BLOCK_IDX, *BLOCK_ADDR;
    BN_CTX *ctx2;
    ctx2 = BN_CTX_new();
    R       = BN_new();  // resto della divisione
    X       = BN_new();  // indice del  blocco
    Y       = BN_new();  // ultimo numero del  blocco
    BLOCK_IDX       = BN_new();  // ultimo numero del  blocco
    BLOCK_ADDR      = BN_new();  // Block Address

    BN_set_word(R,0);
    BN_set_word(X,1);
    BN_set_word(Y,1);
    BN_set_word(BLOCK_ADDR, block_addr);
    BN_mul(X, BLOCK_ADDR, BLOCK_DIM_SIZE,ctx2);  // x = i  block_size
    BN_add(X,X,ONE);                             // x = x+1
    BN_add(Y,X,BLOCK_DIM_SIZE);                  // y = x + block_size
    BN_sub(Y,Y,TWO);                             // y = y - 2
    BLOCK_IDX=BN_dup(X);

    while ( BN_cmp(BLOCK_IDX,Y) )
    {
        BN_add(BLOCK_IDX,BLOCK_IDX,TWO);
        BN_mod(R,M,BLOCK_IDX, ctx2);
        if ( BN_is_zero(R) )
        {
            F=BN_dup(BLOCK_IDX);
            return (1); // FOUND
        }
    }

    return (0);  // NOT FOUND
}

/*********************************************/

void options(int argc, char * argv[])
{
    int i;
    while ( (i = getopt(argc, argv, "a:b:m:h")) != -1)
    {
        switch (i)
        {
            case 'a':  block_addr_bit   = strtol(optarg, NULL, 10);  break;
            case 'b':  modulus_bit      = strtol(optarg, NULL, 10);  break;
            case 'm':  strcpy(m,optarg);  break;
            case 'h':  printf ("\n%s [-b modulus_bit] [-m modulus] [-a block_addr_bit] [-h]",argv[0]);  exit(1);
     case 's':  block_addr_start = strtol(optarg, NULL, 10);  break;
            default:   printf ("\n%s [-b modulus_bit] [-m modulus] [-a block_addr_bit] [-h]",argv[0]);  exit(1);
        }
    }
}
```