Laborator 9
MPI – Evaluarea performanțelor / determinarea punctului de optim

Să se implementeze un algoritm paralel MPI care să afle câte numere prime sunt mai mici decât un număr dat (LIMIT) și care este numărul prim cel mai mare. Toate firele de execuție (inclusiv firul de execuție 0 – FIRST) vor testa un număr egal de numere. Dacă numărul LIMIT este mai mic de 100 fiecare fir de execuție va afișa numerele prime găsite, în caz contrar se vor afișa doar câte numere prime au fost găsite și care este cel mai mare număr prim. Se va afișa și timpul total de execuție.

Pentru rularea programului nu uitați să efectuați setările necesare pentru mediul MPI: Project -> properties -> c/c++-> All options -> Additional Include Directories : $(MSMPI_INC); $(MSMPI_INC)\x86 ; Project -> Properties -> Linker -> All options -> Additional Dependencies : msmpi.lib; Project -> Properties -> Linker -> All options -> Additional Library Directories : $(MSMPI_LIB32)

```
#include "stdafx.h"
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define LIMIT     16000000
#define FIRST     0

int isprime(int n) {
      int i, squareroot;
      if (n>10) {
            squareroot = (int)sqrt(n);
            for (i = 3; i <= squareroot; i = i + 2)
                  if ((n%i) == 0)
                        return 0;
            return 1;
      }
      else
            return 0;
}

int main(int argc, char *argv[])
{
      clock_t start, stop;
```

```
double durata;
int  ntasks,
     rank,
     n,
     pc,
     pcsum,
     foundone,
     maxprime,
     mystart,
     stride;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
if (((ntasks % 2) != 0) || ((LIMIT%ntasks) != 0)) {
     printf("Sorry - this exercise requires an even number of
                                                          tasks.\n")
     printf("evenly divisible into %d.  Try 4 or 8.\n", LIMIT);
     MPI_Finalize();
     exit(0);
}
start = clock();
mystart = (rank * 2) + 1;
stride = ntasks * 2;
pc = 0;
foundone = 0;
if (rank == FIRST) {
     printf("Using %d tasks to scan %d numbers\n", ntasks,
                                                          LIMIT);
     pc = 4;
     for (n = mystart; n <= LIMIT; n = n + stride) {
          if (isprime(n)) {
               pc++;
               foundone = n;
               if (LIMIT<100) printf("%d\n", foundone);
          }
     }
     MPI_Reduce(&pc, &pcsum, 1, MPI_INT, MPI_SUM, FIRST,
                                        MPI_COMM_WORLD);
     MPI_Reduce(&foundone, &maxprime, 1, MPI_INT, MPI_MAX,
                                        FIRST, MPI_COMM_WORLD);
     end_time = MPI_Wtime();
     printf("Done. Largest prime is %d Total primes %d\n",
                                        maxprime, pcsum);
     stop = clock();
     durata = (double)(stop - start) / CLOCKS_PER_SEC;
```

```
            printf("Time elapsed: %2.10f seconds\n", durata);
      }
      if (rank > FIRST) {
            for (n = mystart; n <= LIMIT; n = n + stride) {
                  if (isprime(n)) {
                        pc++;
                        foundone = n;
                        if (LIMIT<100) printf("%d\n",foundone);
                  }
            }
            MPI_Reduce(&pc, &pcsum, 1, MPI_INT, MPI_SUM, FIRST,
                                          MPI_COMM_WORLD);
            MPI_Reduce(&foundone, &maxprime, 1, MPI_INT, MPI_MAX,
                                          FIRST, MPI_COMM_WORLD);
      }

      MPI_Finalize();
}
```

Sarcini de lucru:

Să se implementeze și varianta serială a algoritmului pentru a putea compara timpii de execuție și pentru a calcula factorul de accelerare a algoritmului paralel.

Să se înlocuiască metoda de calcul al timpului de execuție utilizând funcția *MPI_Wtime()*.

Să se varieze numărul de fire de execuție și dimensiunea intervalului de evaluat pentru a determina raportul optim între cei doi parametrii.