Parallel Programming

Laboratory 4 – Problem 1

~ 2022 ~

Balázs Benedek

Group 30444/1

Problem 1:

Range = [11, 100.000.000,0] (100 millions – 8 zeroes)

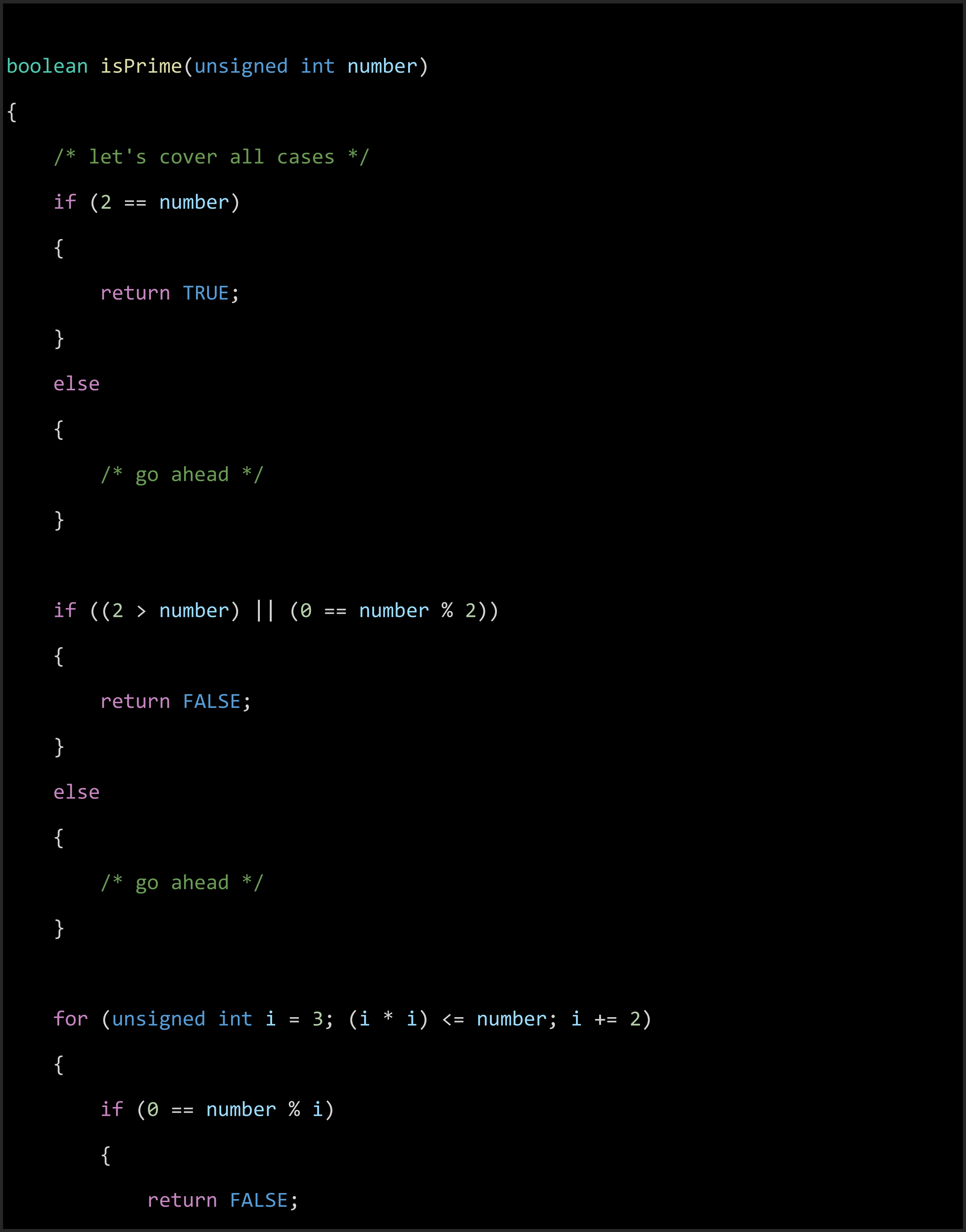
M = 6 * 1.3 = 7.8

PROCESSES	Execution time [seconds]	[Relative] Speedup S(n) = T(1)/T(n)	[Relative] Efficiency E(n) = S(n) / M
1	66.386711	1	0.1282051282
2	38.835976	1.709412711	0.2191554757
3	37.010376	1.793732412	0.2299656938
4	22.118071	3.001469296	0.3848037559
5	23.297922	2.849469193	0.3653165632
6	19.688889	3.371785528	0.4322801959
7	14.937309	4.444355473	0.5697891632
8	12.235679	5.425666283	0.6955982414
9	14.902102	4.454855496	0.57113532
10	13.063422	5.081877551	0.6515227629
11	9.402281	7.060702717	0.905218297
12	10.50884	6.317225403	0.8099006926
…			
16	10.386058	6.391906438	0.8194751844
110	66.386711	1	0.1282051282

TABLE 1. Performance parameters for Problem 1

```
#include <stdio.h>

#include <unistd.h>

#include <time.h>

#include <stdlib.h>
```

```
#include <math.h>
#include <sys/wait.h>

#define TRUE  1u
#define FALSE 0u
#define LOW 11
#define HIGH 1e+8
#define MICRO_SEC_TO_SEC (float)1000000.0f

typedef unsigned char boolean;

/* for simplicity the no. of processes is defined here,
 * But a better idea is to read it from the command line */

void do_work(int i, int PROCESSSES);
int64_t difftimespec_us(const struct timespec after, const struct timespec before);

int main(int argc, char** argv)
{
    int i, pid;
    /* Take initial time here */
    /* Use clock_gettime(). Do NOT use clock() */
    struct timespec start;
    struct timespec stop;

    clock_gettime(CLOCK_MONOTONIC, &start);
    int PROCESSES = atoi(argv[1]);

    for(i = 0; i < PROCESSES; i++)

    {

        pid = fork();

        if(pid < 0) /* some error occurred – fork failed */

        {

            printf("Error");

            exit(-1);

        }

        if(pid == 0) /* child process code */

        {

            do_work(i, PROCESSES);

            exit(0);

        }

        /* do not place any wait() call here */

    }


    /* wait for all processes to finish their execution */

    for(i = 0; i < PROCESSES; i++)

        wait(NULL);


    /* Take final time here */

    /* Use clock_gettime(). Do NOT use clock() */

    /* Compute the execution time*/

    clock_gettime(CLOCK_MONOTONIC, &stop);

    printf("\n$> execution time (s): %lf\n", (difftimespec_us(stop, start) /
MICRO_SEC_TO_SEC));

}
```



```
boolean isPrime(unsigned int number)
{
    /* let's cover all cases */
    if (2 == number)
    {
        return TRUE;
    }
    else
    {
        /* go ahead */
    }

    if ((2 > number) || (0 == number % 2))
    {
        return FALSE;
    }
    else
    {
        /* go ahead */
    }

    for (unsigned int i = 3; (i * i) <= number; i += 2)
    {
        if (0 == number % i)
        {
            return FALSE;
        }
        else
        {
            /* continue execution */
        }
    }
    return TRUE;
}

boolean checkCircular(unsigned int number)
{
    unsigned int cnt;
    unsigned int temp;

    cnt = 0;
    temp = number;
    while (0 != temp) {
        ++ cnt;
        temp /= 10;
    }

    temp = number;
    while (TRUE == isPrime(temp)) {
        unsigned int rem = temp % 10;
        unsigned int div = temp / 10;
        temp = (pow(10, cnt - 1)) * rem + div;
        if (temp == number)
            return TRUE;
    }
    return FALSE;
}

void circularPrimesInRange(unsigned int A, int P, unsigned int B)
{
    unsigned int val = A;
    while (val < B)
    {
        if (checkCircular(val))
        {
            printf("%d ", val);
        }
        else
        {
            /* Do nothing */
        }
        val += P;
    }
}

/* this function is executed by each process */
void do_work(int i, int PROCESSES)
{
    //printf("Hello there, from process %d! \n", i);
    /* rest of the code goes here */

    circularPrimesInRange(LOW + 2*i, 2*PROCESSES, HIGH);

}


int64_t difftimespec_us(const struct timespec after, const struct timespec
before)

{

        return ((int64_t)after.tv_sec - (int64_t)before.tv_sec) *
(int64_t)1000000

                     + ((int64_t)after.tv_nsec - (int64_t)before.tv_nsec) / 1000;

}
```