Parallel Programming

Laboratory 8

~ 2022 ~

Balázs Benedek

Group 30444/1

Problem 1:

Range: 10.000.000 – 20.000.000

M = 6 * 1.3 = 7.8

PROCESSES	Execution time [milliseconds]	[Relative] Speedup S(n) = T(1)/T(n)	[Relative] Efficiency E(n) = S(n) / M
1	273892	1	0.1282051282
6	70302	3.895934682	0.4994788054
12	48099	5.69433876	0.7300434307
24	49415	5.542689467	0.7106012137

TABLE 1. Performance parameters for Problem 1

1 Thread:

```
blasio99@DESKTOP-BB:/mnt/e/TNT/University/__4YEAR__/AN4_SEM2/PP/Lab08$ ./lab08
> Thread 0 found 1066 numbers
> Sum == 1066
> ellapsed time: 273.8920984000 sec
```

6 Threads:

```
blasio99@DESKTOP-BB:/mnt/e/TNT/University/__4YEAR__/AN4_SEM2/PP/Lab08$ ./lab08
> Thread 0 found 172 numbers
> Thread 1 found 274 numbers
> Thread 2 found 181 numbers
> Thread 3 found 176 numbers
> Thread 4 found 146 numbers
> Thread 5 found 117 numbers
> Sum == 1066
> ellapsed time: 70.3020974000 sec
```

12 Threads:

```
blasio99@DESKTOP-BB:/mnt/e/TNT/University/__4YEAR__/AN4_SEM2/PP/Lab08$ ./lab08
> Thread 0 found 96 numbers
> Thread 1 found 76 numbers
> Thread 2 found 152 numbers
> Thread 3 found 122 numbers
> Thread 4 found 93 numbers
> Thread 5 found 88 numbers
> Thread 6 found 97 numbers
> Thread 7 found 79 numbers
> Thread 8 found 78 numbers
> Thread 9 found 68 numbers
> Thread 10 found 61 numbers
> Thread 11 found 56 numbers
> Sum == 1066
> ellapsed time: 48.0998170000 sec
blasio99@DESKTOP-BB:/mnt/e/TNT/University/__4YEAR__/AN4_SEM2/PP/Lab08$ ./lab08
```

24 Threads:

```
 > Thread 12 found 52 numbers
 > Thread 13 found 45 numbers
 > Thread 14 found 33 numbers
 > Thread 15 found 46 numbers
 > Thread 16 found 39 numbers
 > Thread 17 found 39 numbers
 > Thread 18 found 32 numbers
 > Thread 19 found 36 numbers
 > Thread 20 found 36 numbers
 > Thread 21 found 25 numbers
 > Thread 22 found 36 numbers
 > Thread 23 found 20 numbers
 > Sum == 1066
 > ellapsed time: 49.4152659000 sec
 blasio99@DESKTOP-BB:/mnt/e/TNT/University/__4YEAR__/AN4_SEM2/PP/Lab08$
```

Code:

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

```
/* defines */
#define TRUE  (1u)              /* boolean true */
#define FALSE (0u)              /* boolean false */
#define NUMBER_LENGTH (8)       /* the length of a number */
#define FACTOR_LENGTH (4)       /* the length of a divisor */
#define RANGE_MIN (10000000)    /* the minimum value of the range */
#define RANGE_MAX (20000000)    /* the maximum value of the range */
#define DIV_MIN (1000)          /* the smallest divisor */
#define DIV_MAX (9999)          /* the greatest divisor */
#define NR_OF_THREADS (24)      /* the number of threads to make it parallel */
#define UNAVAILABLE (99)        /* the define for a used digit - at verification
of the presence of a divisor digit in the number */

/* typedefs */
typedef unsigned char boolean;
typedef double float64;

/* global variables */
int sum = 0;
int partialSum[NR_OF_THREADS] = {0};
boolean vampireNumbers[RANGE_MAX - RANGE_MIN + 1] = {FALSE};

/* function headers */
boolean checkDigits(int* digitsNr, int divisor1, int divisor2);
boolean checkVampire(int nr);
void vampire(void);
void printResults(void);

/* main function */
void main(void)
{
    omp_set_nested(1);
    omp_set_dynamic(0);

    float64 start = omp_get_wtime();    /* get the start time */
    omp_set_num_threads(NR_OF_THREADS); /* set the number of threads */
    vampire();                          /* solution */
    float64 end = omp_get_wtime();      /* get the end time */

    printResults();                     /* print the results, and the vampire
numbers into file */
```

```
    printf("> ellapsed time: %.10lf sec\n", end - start);
}

void vampire(void)
{
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = RANGE_MIN; i <= RANGE_MAX; ++i)
        {
            if(TRUE == checkVampire(i))
            {
                partialSum[omp_get_thread_num()] ++;    /* sum of each thread */
                vampireNumbers[i - RANGE_MIN] = TRUE;   /* bitmap for vampire
numbers */
                                                        /* i - RANGE_MIN to start
from position 0 */
            }
        }
    }
}

boolean checkVampire(int nr)
{
    int divisors[10000] = {0};
    int nrOfDiv = 0;
    int digits[NUMBER_LENGTH] = {0};
    int auxNr = nr;

    for (int i = 0; i < NUMBER_LENGTH; ++i)
    {
        digits[i] = auxNr % 10;
        auxNr /= 10;
    }

    for (int i = DIV_MIN; i <= DIV_MAX; ++i)
    {
        if (0 == nr % i)
        {
            divisors[nrOfDiv++] = i;
        }
    }
```

```
    for(int i = 0; i < nrOfDiv - 1; ++i)
    {
        for (int j = i; j < nrOfDiv; ++j)
        {
            if (divisors[i] * divisors[j] == nr)
            {
                if (checkDigits(digits, divisors[i], divisors[j]))
                {
                    return TRUE;
                }
            }
        }
    }
    return FALSE;
}

boolean checkDigits(int* digitsNr, int divisor1, int divisor2)
{
    int auxDigits[NUMBER_LENGTH];

    /* 2 divisors cannot terminate with 0 at the same time */
    if((0 == divisor1 % 10) && (0 == divisor2 % 10))
    {
        return FALSE;
    }

    /* create an auxiliar digit vector, to not change the digitsNr */
    for (int i = 0; i < NUMBER_LENGTH; auxDigits[i] = digitsNr[i], i++);

    /* checking the presence of all divisor digits in the original number */
    for (int i = 0; i < FACTOR_LENGTH; ++i)
    {
        int digit1 = (int)(divisor1 % 10);
        int digit2 = (int)(divisor2 % 10);

        divisor1 /= 10;
        divisor2 /= 10;

        boolean foundDigit1 = FALSE;
        boolean foundDigit2 = FALSE;
```

```
        for (int j = 0; j < NUMBER_LENGTH; ++j)
        {
            if ((auxDigits[j] == digit1) && (FALSE == foundDigit1))
            {
                foundDigit1 = TRUE;
                auxDigits[j] = 99;  /* set to 99, as we do not want to repeat the
digits */
            }

            if ((auxDigits[j] == digit2) && (FALSE == foundDigit2))
            {
                foundDigit2 = TRUE;
                auxDigits[j] = 99;  /* set to 99, as we do not want to repeat the
digits */
            }
        }

        if ((FALSE == foundDigit1) || (FALSE == foundDigit2))
        {
            /* if one of the digits is not found then it is not vampire */
            return FALSE;
        }
    }
    return TRUE;
}

void printResults(void)
{
    for(int i = 0; i < NR_OF_THREADS; ++i)
    {
        printf("> Thread %d found %d numbers\n", i, partialSum[i]);
        sum += partialSum[i];
    }
    printf("> Sum == %d \n", sum);

    FILE *fp = fopen("results.txt", "w");

    for(int i = 0; i < RANGE_MAX - RANGE_MIN; ++i)
    {
        if(TRUE == vampireNumbers[i])           /* if the number is vampire */
        {
```

```
            fprintf(fp, "%d\n", i + RANGE_MIN); /* i + RANGE_MIN because we have
started from position 0 in the bitmap */
        }
    }
    fclose(fp);
}
```