Parallel Programming

Laboratory 4 – Problem 2

~ 2022 ~

Balázs Benedek

Group 30444/1

Problem 2:

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	17.672621	1	0.1282051282
2	9.06657	1.949206922	0.2498983233
3	6.414338	2.755174579	0.3532275102
4	5.396834	3.274627495	0.4198240378
5	4.354255	4.058701431	0.5203463373
6	3.766142	4.692499911	0.6016025527
7	3.347946	5.278645773	0.676749458
8	3.116926	5.669887896	0.7269087046
9	2.87878	6.138927254	0.7870419556
10	2.584043	6.839135804	0.8768122825
11	2.367289	7.465341578	0.9570950742
12	2.265901	7.799379143	0.999920403
…			
16	2.529094	6.987728016	0.8958625662

TABLE 1. Performance parameters for Problem 1

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include <sys/wait.h>
```

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

#define N 1000000000
#define MICRO_SEC_TO_SEC (float)1000000.0f

void do_work(int i, int PROCESSES);
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*/

    FILE *fp;
    char filename[20];
    int M, tM;
    M = 0;

    for (i = 0; i < PROCESSES; i++)
    {
        sprintf(filename, "process%d.txt", i);
        fp = fopen(filename, "r");

        fscanf(fp, "%d", &tM);
        M += tM;
        fclose(fp);
    }

    clock_gettime(CLOCK_MONOTONIC, &stop);

    printf("\n> PI = %lf\n", 4.0 * M / N);
    printf("$> execution time (s): %lf\n", (difftimespec_us(stop, start) /
MICRO_SEC_TO_SEC));
}

int isInsideCircle(double x, double y)
{
    return 1.0 >= ((x * x) + (y * y));
}

/* 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 */
    unsigned int M = 0;
    unsigned int seed = time(NULL);

    for (unsigned int i = 0; i < N / PROCESSES; i++)
    {
        double x = rand_r(&seed) * 1.0 / RAND_MAX;
        double y = rand_r(&seed) * 1.0 / RAND_MAX;
```

```
        if (isInsideCircle(x, y))
        {
                ++ M;
        }
        else
        {
            /* Do nothing */
        }
    }

    FILE *fp;
    char filename[20];
    sprintf(filename, "process%d.txt", i);
    fp = fopen(filename, "w");

    fprintf(fp, "%d", M);
    fclose(fp);
}

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;
}
```