Parallel computing, Programming assignment

Angelos Anagnostopoulos
up1066593@upnet.gr

University of Patras — January 1, 2023

Introduction

The point of this excercise was to use parallelization libraries in the C programming language in order to calculate various numbers with use of different algorithms and mathematical formulae. Code was written in both OpenMP and the MPI libraries, both wrappers that deal with threads and inter-process messages. They are very easy to use and replace the mind-bending complexity of Posix threads, and thus were embraced by the C community.

For this assignment, we have to calculate the constants e and π, as well as the natural logarithm ln(x) of a small number x between 0 and 2.

1 Approximating pi

Throughout the years, multiple methods of approximating π have been proposed, each with its own pros and cons. For this assignment we were left free to choose the implementation details by ourselves, which gives us room to navigate into different mathematical formulae and write some interesting code.

1.1 OpenMP implementation

By integrating $\sqrt{1-x^2}$ from -1 to 1 we can calculate the value of pi wiht great accuracy. This stems from the fact that the integral converges to $\frac{\pi}{2}$. We therefore need to arithmetically calculate the integral:

$$I = \int_{-1}^{1} \sqrt{1-x^2}\, \mathrm{d}x = \frac{\pi}{2}. \tag{1}$$

Arithmetic integration is done by using a tiny step dx in order to divide the integral up into very small areas that can be calculated and summed up. We divide our function f(x) into multiple dx segments and multiply the function value with our current dx segment, adding the result to a "master" sum. This is essentialy the definition of a definite integral. The smaller we make dx, the better the accuracy of the method. As it turns out, even relatively small values of 10.000, are effective in calculating multiple decimal points. As we will see later on, this is not the case with our other method of choice. The complete source code can be seen on the next page.

In each itteration of the program, for each small dx we add a partial area value to the result. At the end we display the final result to the console. With a standard integer we can reach about 2 million itterations, which is more than enough to calculate π with great accuracy.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main(int argc, char **argv){

    long double x, y, dx, area = 0.0;
    float time;
    int i, steps, threads;

    steps = atoi(argv[1]) * 10000;
    threads = atoi(argv[2]);
    time = omp_get_wtime();
    dx = 1.0 / (long double)steps;
    x = dx;

    omp_set_num_threads(threads);
    #pragma omp parallel for schedule(dynamic) reduction(+: area) private(x, y, i, dx)
    for (i = 1; i < steps; i++)
    {
        dx = 1.0 / (long double)steps;
        x = (long double)i * dx;
        y = sqrt((long double)1.0 - (pow(x, 2)));
        area += dx * y;
    }
    printf("Constant pi value: %1.10Lf\n", area * 4.0);
    printf("Time taken = %f\n",omp_get_wtime() - time);
}
```

"/home/angelos/Desktop/parallel_projects/pi_aprox/pi_integral_openmp.c"

1.2 MPI implementation

The MPI library is a bit more complicated syntactically, but ultimatelly achieves the same results by allowing different threads and processes to communicate with each other and automatically dividing the workload to them. For this method, we chose to implement a Monte Carlo method. By using randomness over many itterations, we can reach an estimation about the value of π which is close to the real thing!

This of course requires creation of a random number in each itteration and is very time consuming but yeilds promising results for large numbers of itterations. For our needs, we went with 10.000.000. Unfortunately this number does not give us accurate results, so a workaround has been implemented. By storing calculated values in a file and taking their median, we can come a lot closer to the true value of π than with a single use of the method.

Our method is based on the unit circle, and the fact that the area under one quarter of the unit circle is equal to $\frac{\pi}{4}$. By taking a lot of random points and seeing if they belong inside the unit circle or not, we can come up with a clever trick to calculate an approximation of pi as the number of "hits" divided by the total number of points. Let the total number of points be N and the points inside the unit circle M. Then:

$$\frac{M}{N} = \frac{\pi}{4}, \tag{2}$$

$$\pi = \frac{4 * M}{N} \tag{3}$$

After writting the results to the file, a small python script is called to calculate the median and print the output to the screen. Below is the source code for the MPI implementation as well as the python script:

Approximating π with MPI source code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <unistd.h>

#define N 1E8
#define d 1E-8

int main (int argc, char **argv){

    double pi=0.0, x, y;
    int rank, size, error, i, start;
    int result=0, sum=0;
    FILE *fp;

    error=MPI_Init (&argc, &argv);

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    //Synchronize all processes and get the begin time
    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();

    srand((int)time(0));

    //Parallelized part for partial results from different processes
    for (i=rank; i<N; i+=size)
    {
        x=rand()/(RAND_MAX+1.0);
        y=rand()/(RAND_MAX+1.0);
        if(x*x+y*y<1.0)
            result++;
    }

    //Sum up all results
    MPI_Reduce(&result, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    //Synchronize all processes and get the end time
    MPI_Barrier(MPI_COMM_WORLD);

    //Caculate and write PI values from each itteration to a file
    fp = fopen("./pi_aprox/pi_sums.txt", "a");
    if (fp == NULL){
        printf("File cannot be openned, exiting...");
        exit(1);
    }
    if (rank==0){
        pi=4*d*sum;
        printf("np=%2d;    Time=%fs;    PI=%0.8f\n", size, MPI_Wtime() - start, pi);
        fprintf(fp, "%0.8f\n", pi);
    }
    fclose(fp);

    error=MPI_Finalize();

    return 0;
}
```

"/home/angelos/Desktop/parallel_projects/pi_aprox/pi_montecarlo_MPI.c"

Python script to find median of multiple approximations

```
import os

line_sum = 0
filepath = os.getcwd() + "/pi_aprox/pi_sums.txt"

with open(filepath, "r") as file:
    lines = file.readlines()
    for line in lines:
        line = line.strip()
        line_sum += float(line)
    median = line_sum / len(lines)
    print(f"\nThe median value for pi is: {median}")
```

"/home/angelos/Desktop/parallel_projects/pi_aprox/sumfile.py"

In order to automate the whole process, a shell script was created. To run it, simply execute:

```
$ ./pi_aprox/run.sh
```

2 Approximating ln(x)

In order to approximate the value of ln(x), we can use a mathematical formula derived by the Taylor series. That will give us a sum that we shall divide upon our threads to calculate partially, and then add their results to a master-sum. The formula in question is:

$$ln(x) = \sum_{i=1}^{n} (-1)^{i+1} [\frac{(x-1)^i}{i}] \tag{4}$$

Natural logarithm ln(x) approximation source code:

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

int main(int argc, char **argv){

    long double ln = 0.0, part_sol, x;
    float time;
    int i, steps, threads;

    x = atof(argv[1]);
    steps = atoi(argv[2]) * 10000;
    threads = atoi(argv[3]);

    time = omp_get_wtime();
    //Number of threads to use for parallel region
    omp_set_num_threads(threads);
    #pragma omp parallel for shared(ln, part_sol) reduction(+: ln)
    for (i = 1; i <= steps; i++)
    {
        part_sol = pow(-1.0, (i+1)) * (pow((x-1),i) / i);
        ln += part_sol;
    }

    printf("Natural log value: %1.10Lf\n", ln);
    printf("Time taken = %f\n", omp_get_wtime() - time);

    return 0;
}
```

"/home/angelos/Desktop/parallel_projects/ln_aprox/ln_openmp.c"

3 Approximating e

Euler's constant can be calculated from the McLauren series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \tag{5}$$

We shall create an algorithm that parallelizes this, exactly as we did with ln(x) in the above section. This algorithm will need to calculate the factorial of each itteration and add the partial sum to our approximation of e.

Euler's constant (e) approximation source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main(int argc, char **argv){

    long double e = 1.0, fact = 1.0, part_sol; //To skip the first itteration for e = 0
    and 0! = 1
    float time;
    int i, steps, threads;

    steps = atoi(argv[1]) * 10000;
    threads = atoi(argv[2]);

    time = omp_get_wtime();

    //Number of threads to use for parallel region
    omp_set_num_threads(threads);
    #pragma omp parallel for shared(e) reduction(+: e)
    for (i = 1; i <= steps; i++)
    {
        fact *= i;
        part_sol = 1.0 / fact;
        e += part_sol;
    }

    printf("Euler's constant value: %1.11Lf\n", e); //We are correct up to 11 decimal
    points. Anything less and C rounds up the number w/ a small error.
    printf("Time taken = %f\n", omp_get_wtime() - time);

    return 0;
}
```

"/home/angelos/Desktop/parallel_projects/euler_aprox/euler_openmp.c"

Running the code

Shell scripts were created in order to automate file compilation and execution. In order to compile our .c files,simply run:

```
$ ./compile
```

The binary files created, are placed in a new directory called /bin and can be executed with the appropriate command. Please note that all code is written with argv in mind for the required steps and the threads to be used. Steps are multiplied by 10.000 for user ease, therefore execute the output with:

```
./bin/<binary> <steps> <threads_number>
```

with the exception of the natural logarithm approximation, in which we need to specify a value for x between [0,2]

```
./bin/ln_openmp <x> <steps> <threads_number>
```

Results

It is interesting to note that although all threads on my computer were being utilized, the results were only minimally sped up for different numbers of threads. Unfortunately, i can only go up to 4 threads so all tests and comparisons were done using 1,2 and 4 of those threads respectively.

Timing code is included in the source code instead of being used with a unix command, so that users can run their own tests and draw their own conclusions on the matter. Alas, sceencaps of timings with different threads/steps combinations are shown bellow.