REPORT FOR THE PROJECT
for the exam of
BIG DATA SCIENCE AND MACHINE LEARNING

Computation of Napier's constant

Academic Year 2021/2022

Professors: GABRIELE GAETANO FRONZÉ
FEDERICA LEGGER
SARA VALLERO

Candidate: RACCA ELEONORA

1 Introduction

Sequential programming can be very time consuming, even with the fastest CPUs available. For this and for many other reasons in the last two decades it is becoming more and more common the parallel computing. This way of programming exploits many CPUs simultaneously, reducing the time needed for repetitive computation tasks.

The aim of this project is to compare the performances of different implementations of parallel computation while calculating the Napier's constant. The computing model used to develop the algorithms in this project is OpenMP.

1.1 Napier's constant

The Napier's constant e is a mathematical constant used for many applications across all sciences. It is included in the Euler's identity with the other four most important mathematical constants:

$$e^{i\pi} + 1 = 0$$

e is an irrational and a transcendental number, and its first 50 decimal places are:

$$2.71828182845904523536028747135266249775724709369995$$

Its name is taken after John Napier, a Scottish mathematician, physicist and astronomer. Sometimes it is also called Euler's number, after the Swiss mathematician Leonhard Euler, even though it was discovered by the Swiss mathematician Jacob Bernoulli.

e can be defined in many different ways, for example it can be defined as the base of the natural logarithm or the limit

$$e = \lim_{n \to 0} \left(1 + \frac{1}{n}\right)^n$$

A convenient way to compute it while taking advantage of the parallel computation is its infinite series:

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

In this project the Napier's number e is calculated with Equation 1. As it is impossible to do the calculation until ∞, the maximum number considered for n is 100000.

1.2 The Code

The code used to compute the calculation can be found in the GitHub repository:

`https://github.com/eleoracca/MLCourse-2122`

in the subfolder `OpenMP-Exam`.

After cloning the GitHub repository to a local folder, it is possible to compile and execute the code locally without adjustments. To compile the code, the following command is needed:

```
g++ -o nepero -fopenmp Nepero.c
```

To execute the code, it is sufficient to launch the newly created executable, which will perform all the functions needed for the comparison.

The code is made of seven functions:

- `main`: which defines the maximum value of n and calls for the functions that perform each parallel computation;
- `factorial`: which is called by all the functions performing the parallel computations and calculates the factorial of a given number n;

- `range`: which performs the parallel computation assigning a user-defined portion of the `for` loop to each thread;
- `forsimple`: which performs the parallel computation using the `for` loop refactored by OpenMP;
- `forcritical`: which performs the parallel computation combining the `for` loop with the `critical` statement of OpenMP to avoid the race conditions;
- `foratomic`: which performs the parallel computation combining the `for` loop with the `atomic` statement of OpenMP to avoid the race conditions;
- `forreduction`: which performs the parallel computation combining the `for` loop with the `reduction` statement of OpenMP to avoid the race conditions;

The structure of the functions, which perform the parallel computations, is always the same.
At the beginning the global variables are defined, which are: the file on which export the results; the variable `e` with the sum at each step of the loop; the time variables, to compute the elapsed time.
In the functions `range` and `forsimple`, the variable `e` is defined as a vector and it is also necessary to have an additional variable `thread_id` which takes into account which thread is working.
Following the declarations, is the `for` loop, which loops on the number of threads from 1 to the maximum available on the specific computer used. In this `for` loop, a nested `for` loop computes the parallelised calculation of the Napier's constant and the time variables record the passing of time.
At the end of the computation, the elapsed time and the value of e are printed.
At the end of the `for` loop on the number of available threads, the execution of the function stops and the `main` function calls the next computation function.

2 Parallel execution in for loops

To parallelize the algorithms with OpenMP, the `pragma` directives are needed. In particular it is necessary to activate the environment

```
#pragma omp parallel
```

Inside this scope, there are two main ways to parallelize the `for` loop in OpenMP:

1. code the `for` loop in the usual way, but defining appropriate ranges for each thread;
2. use the `pragma` directive

   ```
   #pragma omp for
   ```

In the project, both the possibilities were developed in order to compare the performances of the two. The comparison was done using the resources available in the yoga platform of INFN, with 64 threads.
The code uses two different functions:

1. `range`, which implements the computation of the `for` loops using user-defined ranges;
2. `forsimple`, which implements the `pragma` directives.

In Figure 1 is represented the elapsed time as a function of the number of threads used for the computation. In violet is represented the elapsed time using the function `range`, while in green is the elapsed time of the `forsimple` function.
As can be seen, the `range` function is slightly better in terms of time performances with respect to the `forsimple` function. This is a bit strange, because the `pragma` directives are usually more optimised and thus using them improves the elapsed time.
For both the functions, it can be seen that the time is greatly reduced when using more threads, but this behaviour saturated at about 1 second. This is due to the fact that parts of the codes are sequential. Furthermore, over 45 threads, great variations can be observed in the elapsed time: this can be due to the fluctuations of the usage of the threads.

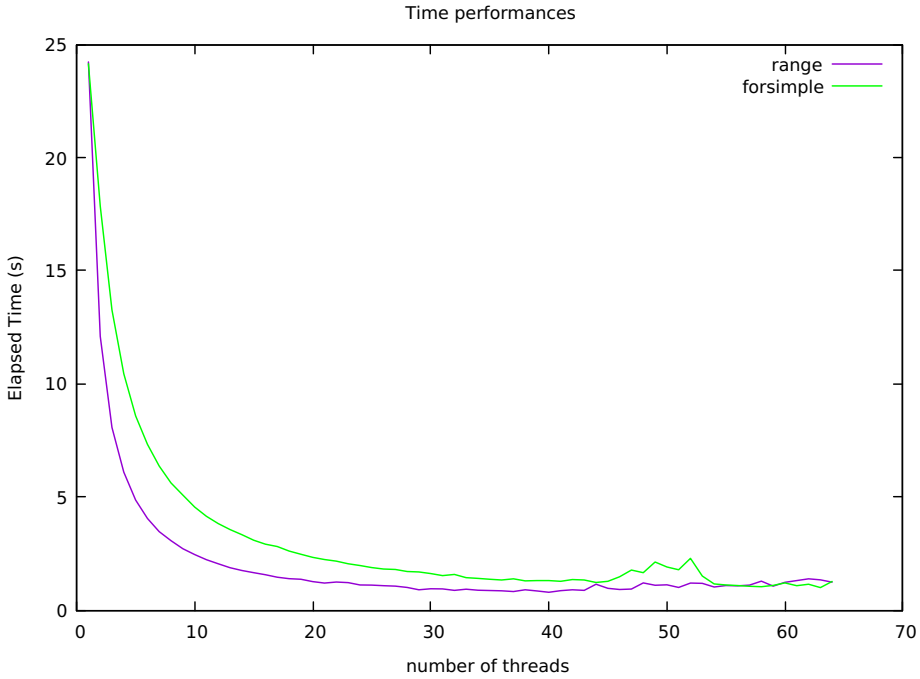

Figure 1: *Elapsed time as a function of the number of threads for the functions* `range` *(violet, no specific* `pragma` *directive) and* `forsimple` *(green, with* `pragma for` *directive).*

3 Avoiding race conditions

In the `range` and `forsimple` the variable `e`, with the sum at each step of the `for` loop, is defined as a vector, as long as the number of threads. At the end of the computation, the elements of the vector are summed to evaluate the number of e. This in principle leads to longer elapsed times, as the sequential part of the code increases.

A way to improve this situation is to use the same variable for all the loops of the different threads. In this case race conditions may occur: these are due to the fact that many threads try to access the same variable at the same time and the result of the computation may be overwritten in the wrong way.
To avoid this problem, there are three possible `pragma` directives:

1. `critical`, which allows one thread access the memory space of the variable and makes the others wait for the first to finish;

2. `atomic`, which protects a specific operation for that variable, for example only one thread can update the variable, but all of them can read its value at the same time;

3. `for reduction (e:sum)`, which is a directive of the `for` loop that creates private copies of the variables for all the threads and at the end performs the desired operation.

In Figure 2 is again represented the elapsed time as a function of the number of threads used for the computation. In blue is represented the elapsed time using the function `forreduction`, which implements the `for reduction (e:sum)` directive, while in green is the elapsed time of the `forsimple` function, in which the variable `e` is defined as a vector.
As can be seen and can be expected, the performances are very similar. In fact in both cases the threads have their own variable in which the partial computation is stored and at the end all of them are summed.

In Figure 3 is, for the third time, represented the elapsed time as a function of the number of threads used for the computation. In this case the curves represented are the elapsed times measured by using the functions `forcritical` in violet, `foratomic` in green and `forreduction` in blue. They implement the directives `critical`, `atomic` and `for reduction (e:sum)` respectively.

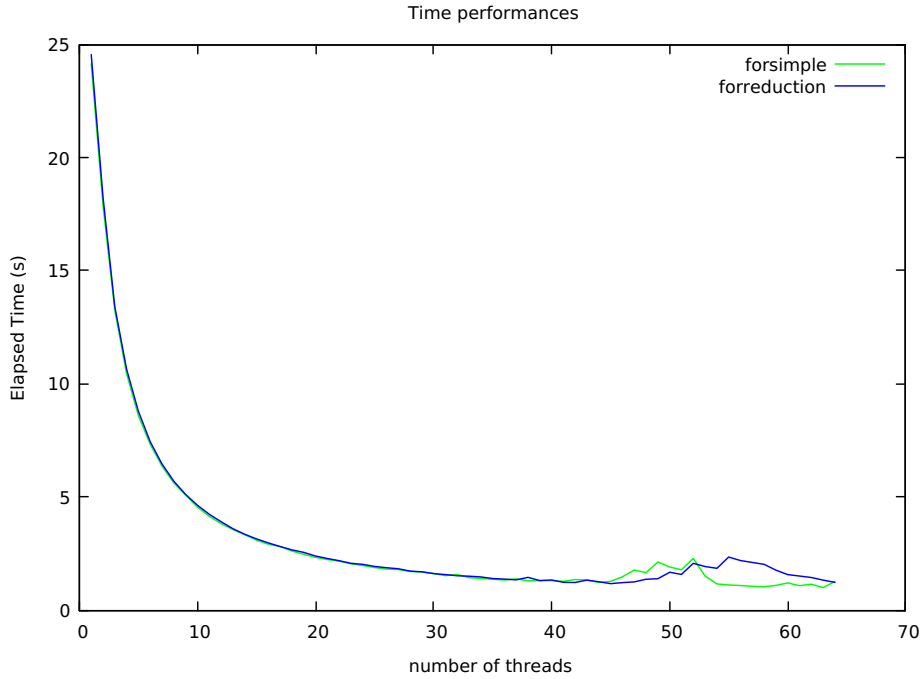

Figure 2: *Elapsed time as a function of the number of threads for the functions* `forsimple` *(green, with* `pragma for` *directive) and* `forreduction` *(blue, with* `pragma for reduction` *directive).*

The time performances are very similar, even though in principle the `critical` and `atomic` should be slightly slower as only one thread can access the variable dedicated for the sum. In this case it can also be seen that when having a lot of threads, there are fluctuation in the elapsed time, but it seems that in this case the `forcritical` and the `foratomic` functions are slightly more stable.

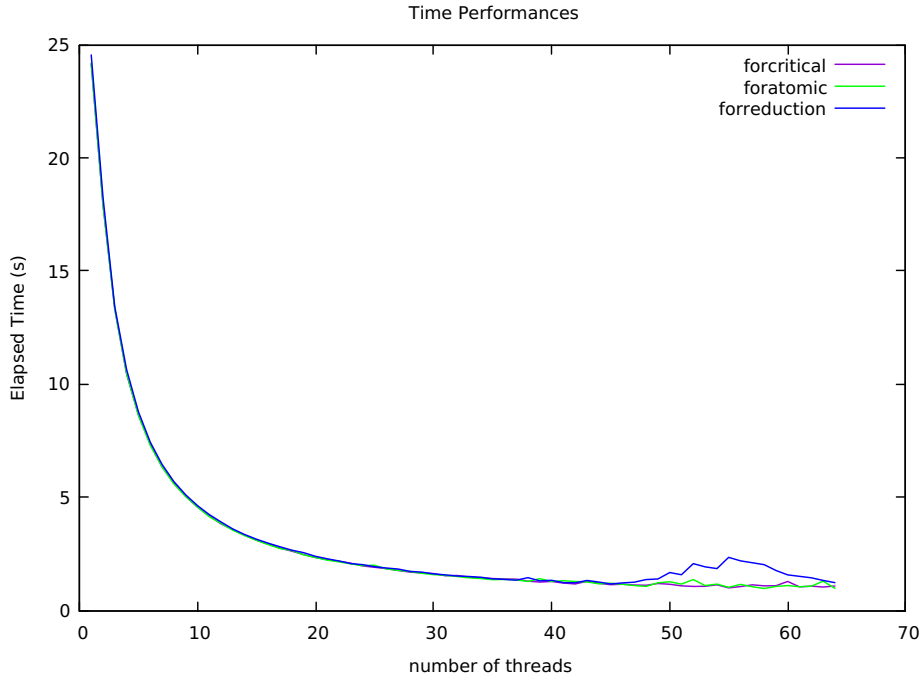

Figure 3: *Elapsed time as a function of the number of threads for the functions* `forcritical`, `foratomic` *and* `forreduction` *(blue, with* `pragma for reduction` *directive).*

4 Conclusions

The project aimed to calculate the value of Napier's constant e using its definition in terms of a series:

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

In the project many different techniques to parallelize the computation.
Each technique provides great time reductions as a function of the number of the threads: the computation lasts more or less 24 seconds when using one thread, and almost 1 second when using 64 threads. In all the cases, some variations in time can be noticed when using more than 40 threads, but the usage of the `pragma critical` and `atomic` seem to reduce this fluctuation.

In conclusion, with the algorithm implemented and contrary to the expectations, below 40 threads it is quicker the implementation without the specific `pragma` directive for the `for` loop. However, when using more than 40 threads it is better in terms of time performances to use the `pragma` directive for the `for` loop. It may also be safer to use the `pragma critical` and `atomic` to experience less variations in terms of computing time.