



Introduction

We would like to optimize the computation of pi by numerical integration method.

A serial version of the algorithm is available on GitHub, alongside some useful material:



https://github.com/gabrielefronze/iCSC/tree/master

This version of the code is the sequential baseline to be used to measure speedup of parallel versions.

```
#include "StopWatch.h"
#include <iostream>
const long num_steps = 500000000; //number of x bins
int main(){
      StopWatch stopWatch; //this object will be destroyed when out of
      scope
     double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps; //x-step
      int n_threads=1;
     for (long i=1; i<=num_steps; i++) {</pre>
           x = (i - 0.5) * step; //computing the x value
            sum += 4.0 / (1.0 + x * x); //adding to the cumulus
     pi = step * sum;
     printf("Pi value: %f\n
      Number of steps: %d\n
      Number of threads: %d\n",
      pi,num_steps,n_threads);
      return 0;
```

StopWatch

This class is useful for timing purposes. It is an interface around:

omp_get_wtime()

This function returns a time measure which is safe in a multithread environment.

The class is implemented as a default constructor and a destructor: it is enough to create an object inside the main function and to let it "die" outside of the scope. The destructor prints on screen the measured elapsed time.

```
#include <omp.h>
#include <iostream>

class StopWatch{

public:
        StopWatch(){
            start = omp_get_wtime();
        }
        ~StopWatch(){
            printf{"Elapsed time: %f\n", omp_get_wtime()-start};
        }

private:
        double start;
}
```

Compilation

Compiling OpenMP code requires a compatible compiler, such as gcc, clang-omp or icc.

Each compiler has his own flag to enable OpenMP support. Tipically the flag is -fonpenmp. For icc the flag is -openmp.

To compile a C/C++ file use the following command:

<compiler> -o <executable_name> -fopenmp <inputfile>

Ingredients

We want to use the pragma:

#pragma omp parallel

This pragma does a fork right after, running an exact copy of the <u>following scope</u> in each spawned thread.

The number of threads is defined and controlled by the environment variable:

OMP_NUM_THREADS

Accessible via the functions:

omp_set_num_threads()
omp_get_num_threads()
omp_get_thread_num()

```
#include "StopWatch.h"
#include <omp.h>
#include <iostream>
const long num_steps = 500000000; //number of x bins
int main()
      StopWatch stopWatch;
      double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps; //x-step
      int n threads=1;
      #pragma omp parallel
            n_threads = omp_get_num_threads();
           // Some changes have to be made:
            // at the moment each thread performs the same operations
            // TIP: work on the for loop ranges
            for (long i=1; i<=num steps; i++) {</pre>
                  x = (i - 0.5) * step; //computing the x value
                  sum += 4.0 / (1.0 + x * x); //adding to the cumulus
      pi = step * sum;
      printf("Pi value: %f\n
      Number of steps: %d\n
      Number of threads: %d\n",
      pi,num_steps,n_threads;
     return 0;
```

Ingredients

We want to use the pragma:

#pragma omp parallel

This pragma does a fork right after, running an exact copy of the <u>following scope</u> in each spawned thread.

The number of threads is defined and controlled by the environment variable:

OMP_NUM_THREADS

Accessible via the functions:

omp_set_num_threads()
omp_get_num_threads()
omp_get_thread_num()

```
#include "StopWatch.h"
#include <omp.h>
#include <iostream>
const long num steps = 500000000; //number of x bins
int main()
      StopWatch stopWatch;
      double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps; //x-step
      int n threads=1;
      #pragma omp parallel
             n threads = omp get num threads();
             /// Some changes have to be made:
/// at the moment each thread performs the same operations
/// TIP: work on the for loop ranges
             for (long i=1; i<=num_steps; i++) {</pre>
                    x = (i - 0.5) * step; //computing the x value
                    sum += 4.0 / (1.0 + x * x); //adding to the cumulus
      pi = step * sum;
      printf("Pi value: %f\n
      Number of steps: %d\n
      Number of threads: %d\n",
      pi,num_steps,n_threads;
      return 0;
```

And now... Make the code rain!

Try to complete the exercise using #pragma omp parallel

