# HPC - Exercise 1

Eduardo Gonnelli

01 July 2019

## 1 Introduction

This report aims to provide the $\pi$ implementation employing the midpoint numerical method. The numerical method was written in C++ programming language and executed in two versions: in serial and in parallel. The execution time for both codes was measured, considering the same problem size, and the scalability and speedup graphs were generated. The parallel part of the problem took into account the variation of the number of threads, from 1 up to 20, and three synchronization pragma OpemMP derivatives: Critical, Atomic and Reduction. All the codes were executed on Ulysses Cluster.

### 1.1 OpenMP

The OpenMP is an Application Programming Interface (API) that supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

As long as different threads write to a different memory location, for example, different elements of the same vector, there is no reason to worry. Problems arise if they simultaneously write to the same address in memory. Then, threads may step on each other and generate incorrect results. This is a bug in the code and is called a "data race."

Synchronizing, or coordinating the actions of, threads is sometimes necessary in order to ensure the proper ordering of their accesses to shared data and to prevent data corruption. A thread is not allowed to enter a critical region, as long as another thread executes it. As a result, a thread cannot perform the update, while another thread is inside the critical region. For this reason, the mechanisms have been proposed to support the synchronization needs of a variety of applications. For our exercise, the Atomic and Critical Construct and Reduction Clause were implemented.

#### 1.1.1 Critical

The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously. The associated code is referred to as a critical region, or a critical section. The Figure 1 refers to the critical section of code.

```
#pragma omp parallel
{
double local = 0.0;
#pragma omp for
        for (int i = 0; i <= (int) n-1; ++i){
                double x_i = h*(i+0.5);
                local += function(x_i);
                }
#pragma omp critical
integral += local;
}
```

**Figure 1:** Syntax of the critical construct in C/C++ – The structured block is executed by all threads, but only one at a time executes the block. Optionally, the construct can have a name.

#### 1.1.2 Atomic

The atomic construct, which also enables multiple threads to update shared data without interference, can be an efficient alternative to the critical region. it is applied only to the (single) assignment statement that immediately follows it. The Figure 2 refers to the atomic section of code.

```
#pragma omp parallel
{
double local = 0.0;
#pragma omp for
        for (int i = 0; i <= (int) n-1; ++i) {
                double x_i = h*(i+0.5);
                local += function(x_i);
                }
#pragma omp atomic
integral += local;
}
```

**Figure 2:** Syntax of the atomic construct in C/C++ – The statement is executed by all threads, but only one thread at a time executes the statement.

### 1.1.3 Reduction

The reduction operator(s) and variable(s) are specified in the reduction clause. By definition, the result variable, like *sum* in this case, is shared in the enclosing OpenMP region. The command *#pragma omp parallel for reduction(+:sum)* is commonly used to parallelize the loop. The Figure 3 refers to the reduction section of code.

```
#pragma omp parallel for reduction(+:integral)
        for (int i = 0; i <= (int) n-1; ++i) {
            double x_i = h*(i+0.5);

            integral += function(x_i);
        }
```

**Figure 3:** Reduction Example of the reduction clause – This clause gets the OpenMP compiler to generate code that performs the summation in parallel

### 1.1.4 Execution time measurement

The execution time of the parallel part of the program was measured employing the function name *omp_get_wtime()*. This function provides the absolute wall-clock time in seconds. It must to be highlighted that the *omp_get_wtime()* returns the number of wall-clock or "elapsed" seconds. In other words, it returns an absolute value. A meaningful timing value is therefore obtained by taking the difference between two calls. Due to this fact, the *omp_get_wtime()* was called before and after the specific portion of the program in which execution time must be measured. The Figure 4 shows the piece of code that contains the application of *omp_get_wtime()*.

```
double tstart = omp_get_wtime();

/* code block to be timed */

double t_wall_clock = omp_get_wtime() - tstart;
```

**Figure 4:** An example how to use function *omp_get_wtime()*

## 2 Scalability and Speedup

The speedup was calculated based on the measured execution time. A simple approach was employed for the calculation, i.e., the same program was executed on a single processor (N° threads equal 1), and on a parallel machine with $p$ processors, and the runtimes were compared. The speedup can be defined as:

$$S_p = T_1/T_p \tag{1}$$

where $S_p$ is the speedup, $T_1$ is the execution time on a single processor and $T_p$ is the time on $p$ processors. Furthermore, to measure how far the results are from the ideal speedup, the efficiency was calculated. It can be defined as:

$$E_p = S_p/p \tag{2}$$

# 3 Midpoint Rule

The rectangular rule (also called the midpoint rule) is a numerical method in which is possible to estimate an integral value with finite sums of rectangles. Using more rectangles can increase the accuracy of the approximation. As can be seen on Figure 5, the midpoint rule uses rectangles whose heights are the values of $f$ at the midpoints of their bases. The interval integration is defined as $a \leq x \leq b$ and divided up into $n$ equal subintervals of length $h = (b - a)/n$.

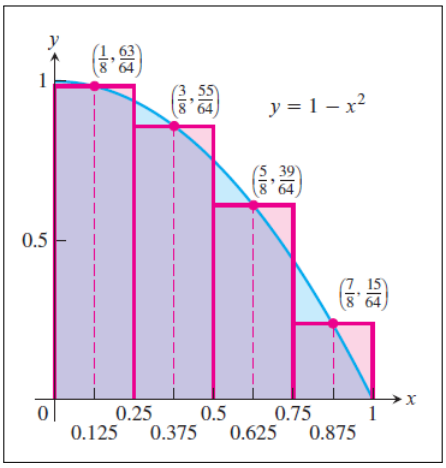


**Figure 5:** The midpoint rule uses rectangles whose height is the value of f(xi) at the midpoints of their bases.

For $\pi$ approximation, the integral of the function $F(x) = 1/(1 + x^2)$ was numerically obtained applying the midpoint procedure.

# 4 Results

The Figure 6 shows the results for critical, atomic and reduction OpenMP derivatives. This figure shows the behavior of the execution time when the number of threads increases.

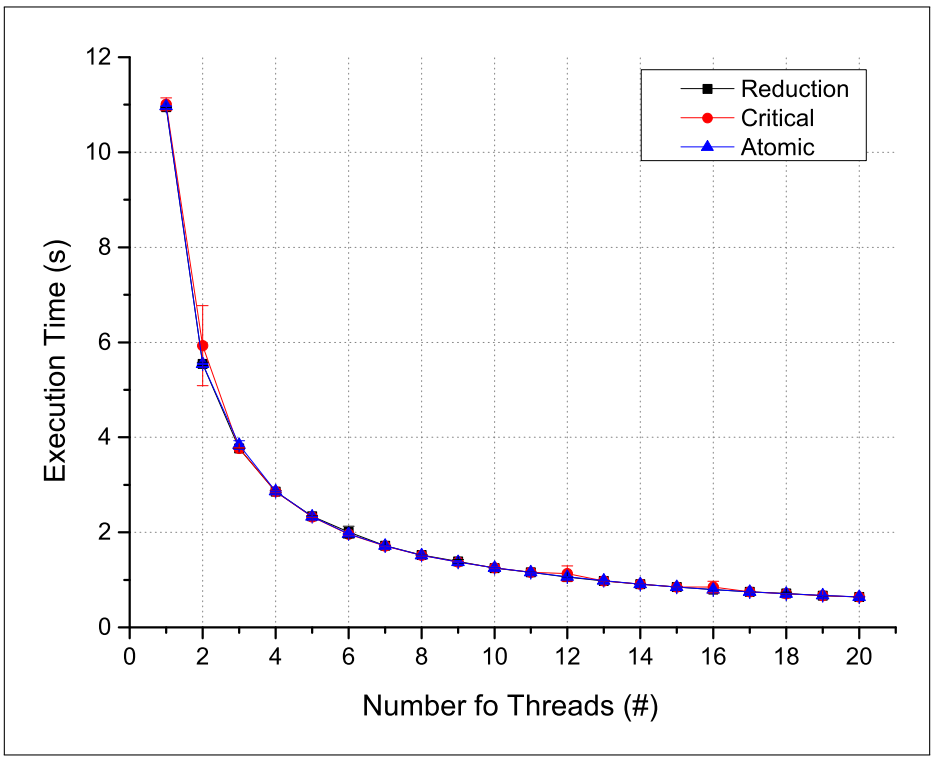


**Figure 6:** Time measurements for different numbers of threads.

The implementation of the serial code for $\pi$ approximation achieved the execution time of 42.10 $\pm$ 0.02 s. The code was executed on the node cn04-33 and the integral approximation of $\pi$ was equal to: 3.14159. For the serial code the execution time was measured employing the Chrono library.

The measurement of the scaling was done by testing how the overall computational time of the job scales with the number of processing elements. The behavior can be seen in Figure 7. The magenta line represents the ideal speedup.

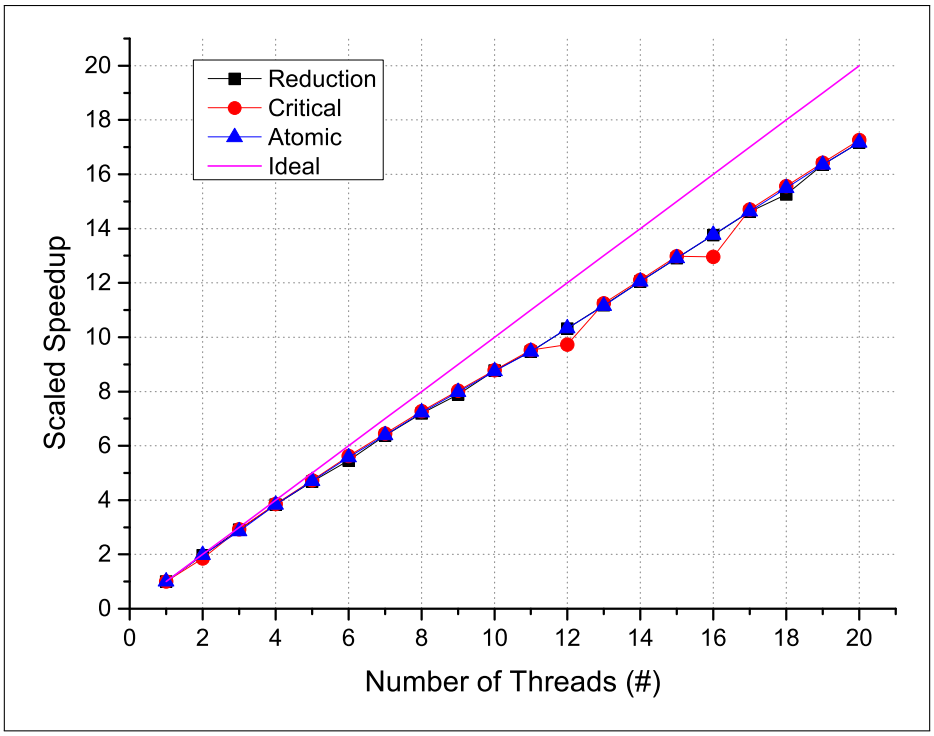


**Figure 7:** Speedup

The Table 1 contains the values for efficiency.

**Table 1:** Efficiency Values.

| | Efficiency (%) | | |
|---|---|---|---|
| N. of Threads (#) | Reduction | Critical | Atomic |
| 1 | 100.00 | 100.00 | 100.00 |
| 2 | 98.74 | 92.84 | 98.86 |
| 3 | 96.94 | 97.43 | 95.29 |
| 4 | 95.75 | 96.43 | 95.87 |
| 5 | 93.70 | 94.63 | 94.10 |
| 6 | 90.98 | 93.67 | 93.05 |
| 7 | 91.14 | 92.07 | 91.34 |
| 8 | 89.92 | 90.91 | 90.36 |
| 9 | 87.58 | 89.16 | 88.69 |
| 10 | 87.76 | 87.79 | 87.53 |
| 11 | 86.10 | 86.60 | 86.07 |
| 12 | 86.02 | 81.02 | 86.07 |
| 13 | 86.02 | 86.53 | 85.80 |
| 14 | 86.07 | 86.53 | 86.11 |
| 15 | 86.06 | 86.52 | 86.07 |
| 16 | 86.03 | 80.97 | 86.04 |
| 17 | 86.01 | 86.49 | 86.07 |
| 18 | 84.79 | 86.48 | 86.05 |
| 19 | 86.08 | 86.48 | 86.09 |
| 20 | 85.85 | 86.29 | 85.84 |

# References


[1] The OpenMP: Architecture Review Boards (ARB)
`https://www.openmp.org/`

[2] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP-Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, 2008.

[3] Ruud van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD*. The MIT Press, Cambridge, 2017.

[4] George B. Thomas, Jr. *Thomas' CALCULUS*. Pearson, Boston, 2016

[5] Victor Eijkhout, Edmond Chow, and Robert van de Geijn. *Introduction to High-Performance Scientific Computing*. Texas Advanced Computing Center, The University of Texas at Austin. Public Draft, 2011

# Appendices

This section contains the codes developed for this work. The compile command for OpenMP was *$ g++ -fopenmp -std=c++11 atomic_pi_chrono.cc -o atomic_pi_chrono.o* and for the serial code was *$ g++ -std=c++11 serial_pi_chrono.cc -o serial_pi_chrono.o*

## A Job Script

**Listing 1:** Job Script

```
#!/bin/bash
#PBS -l nodes=1:ppn=20
#PBS -l walltime=02:30:00
#PBS -q regular

cd /home/egonnell/igirotto/ex01

#load the modules

module load openmpi/1.8.3/intel/14.0

echo "REDUCTION"

for j in {1..5..1};
do for i in {1..20..1};
do export OMP_NUM_THREADS=$i; ./reduction_pi_chrono.o;
sleep 3;
done
done

echo "CRITICAL"

for j in {1..5..1};
do for i in {1..20..1};
do export OMP_NUM_THREADS=$i; ./critical_pi_chrono.o;
sleep 3;
done
done

echo "ATOMIC"

for j in {1..5..1};
do for i in {1..20..1};
do export OMP_NUM_THREADS=$i; ./atomic_pi_chrono.o;
sleep 3;
done
done
```

# B C++ codes

**Listing 2:** Serial Code

```
#include <iostream>
#include <chrono>
#include <vector>

// SERIAL

double function(double x);

int main() {

        double a = 0.0;
        double b = 1.0;
        int n = 2147483647;
        double integral = 0.0;
        double h = (b - a) / n;

auto start = std::chrono::high_resolution_clock::now();

for (int i = 0; i <= (int) n-1; ++i) {
      double x_i = h*(i+0.5);
      integral += function(x_i);
}

auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_vector = finish - start;

double final= 4*h*integral;

        std::cout << "Integral is equal to: " << final << "\n";
        std::cout << "Chrono (std::vector) time: " << elapsed_vector.count()
<<  " s\n";
              return 0;
    }
    double function(double x) {
      return 1.0/( 1.0 + x * x);
    }
```

**Listing 3:** Critical construct

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

// CRITICAL

double function(double x);

int main() {

        double a = 0.0;
        double b = 1.0;
        int n = 2147483647;
        double integral = 0.0;
        double h = (b - a) / n;

auto start = std::chrono::high_resolution_clock::now();
```

```
double tstart = omp_get_wtime();

#pragma omp parallel
{
double local = 0.0;
#pragma omp for
        for (int i = 0; i <= (int) n-1; ++i){
                double x_i = h*(i+0.5);
                local += function(x_i);
                }
#pragma omp critical
integral += local;
}

double duration = omp_get_wtime() - tstart;
auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_vector = finish - start;

        double final= 4*h*integral;
        std::cout << "Number of threads: " << omp_get_max_threads()
<< "\n";
        std::cout << "Integral is equal to: " << final << "\n";
        std::cout << "Time for loop: " << duration << " s\n";
        std::cout << "Chrono (std::vector) time: " << elapsed_vector.count()
<< " s\n";
                return 0;
    }
    double function(double x) {
      return 1.0/( 1.0 + x * x);
    }
```

**Listing 4:** Atomic construct

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

//ATOMIC

double function(double x);

int main() {

        double a = 0.0;
        double b = 1.0;
        int n = 2147483647;
        double integral = 0.0;
        double h = (b - a) / n;

auto start = std::chrono::high_resolution_clock::now();
double tstart = omp_get_wtime();

#pragma omp parallel
{
double local = 0.0;
#pragma omp for
        for (int i = 0; i <= (int) n-1; ++i) {
                double x_i = h*(i+0.5);
                local += function(x_i);
                }
#pragma omp atomic
```

```
integral += local;
}

double duration = omp_get_wtime() - tstart;
auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_vector = finish - start;

        double final= 4*h*integral;
        std::cout << "Number of threads: " << omp_get_max_threads()
        << "\n";
        std::cout << "Integral is equal to: " << final << "\n";
        std::cout << "Time for loop: " << duration << " s\n";
        std::cout << "Chrono (std::vector) time: " << elapsed_vector.count()
        << " s\n";
              return 0;
    }
    double function(double x) {
      return 1.0/( 1.0 + x * x);
    }
```

**Listing 5:** Reduction Clause

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

//REDUCTION

double function(double x);

int main() {

        double a = 0.0;
        double b = 1.0;
        int n = 2147483647;
        double integral = 0.0;
        double h = (b - a) / n;

auto start = std::chrono::high_resolution_clock::now();
double tstart = omp_get_wtime();

#pragma omp parallel for reduction(+:integral)
        for (int i = 0; i <= (int) n-1; ++i) {
            double x_i = h*(i+0.5);

            integral += function(x_i);
        }

double duration = omp_get_wtime() - tstart;
auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_vector = finish - start;

        double final= 4*h*integral;
        std::cout << "Number of threads: " << omp_get_max_threads()
        << "\n";
        std::cout << "Integral is equal to: " << final << "\n";
        std::cout << "Time for loop: " << duration << " s\n";
        std::cout << "Chrono (std::vector) time: " << elapsed_vector.count()
        << " s\n";
              return 0;
```

```
}
double function(double x) {
  return 1.0/( 1.0 + x * x);
}
```