HPC OpenMP - Hands on

Marcello Capasso

November 2018

Abstract

This analysis aims to implement an algorithm that computes the approximation of π in both a serial and parallel fashion. Moreover the scalability is computed for an increasing number of threads. Secondly, a visualization of the loop scheduling types for a toy code is made and the result is discussed.

1 Exercise 1

The algorithm for the numerical integration of π is here found:

Listing 1: Serial_pi.c

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

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

        const long unsigned int N = 1000000000;
        double h = 1./N;
        double pi = 0;
        double h2 = h/2;
        double mid;
        double result;

        for(int i = 0; i < N; i++)
        {
                mid = (2*i+1)*h2;
                result = 1/(1+mid*mid);
                pi += result;

        }
        pi = 4*pi*h;
        printf("%f", pi);
        return 0;
}
```

The algorithm runs in 4.479 seconds for a N value of one billion.

Three versions of the parallel algorithm were implemented. First making use of the atomic directive, and then the critical and reduce ones. This is done in order to protect the summation variable from race conditions. They all share the same common part:

Listing 2: common parts

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

int main( int argc, char * argv[] ) {
    double start = omp_get_wtime();
    const unsigned long int N = 1000000000;
    double h = 1./N;
    double pi = 0;

    double h2 = h/2;
    double result = 0;
    double mid;

    [...]

    pi = 4*pi*h;
    printf("%f", pi);
    printf("%f", (double)omp_get_wtime()-start);

    return 0;
}
```

The part included in [...] is specific for each scheduler type and is found in the following sections.

1.1 Atomic & Critical

Listing 3: atomic and critical

```
#pragma omp parallel private(mid, result) shared(pi)
    {
        #pragma omp for schedule(static)
        for(int i = 0; i < N; ++i) {
            mid = (2*i+1)*h2;
            result += 1/(1+mid*mid);
        }
    #pragma omp atomic OR #pragma omp critical
        pi += result;
    }
```

1.2 Reduction

Listing 4: reduction

```
#pragma omp parallel
    {
        #pragma omp for schedule(static) reduction(+:pi)
        for(int i = 0; i < N; ++i) {
            mid = (2*i+1)*h2;
            result = 1/(1+mid*mid);
            pi += result;
        }
    }
```

2 Results

The algorithms were run for 1, 2, 4, 8, 16, 20, 30 and 40 threads. The running time is measured and plotted in figure 5. As it can be seen, increasing the number of threads increases the speed up only to a certain extent. A number greater than 10 does not considerably decrease the running time, which is stuck approximately to 0.3 seconds. Am almost perfectly equal behaviour can be noted for all the three protection types.

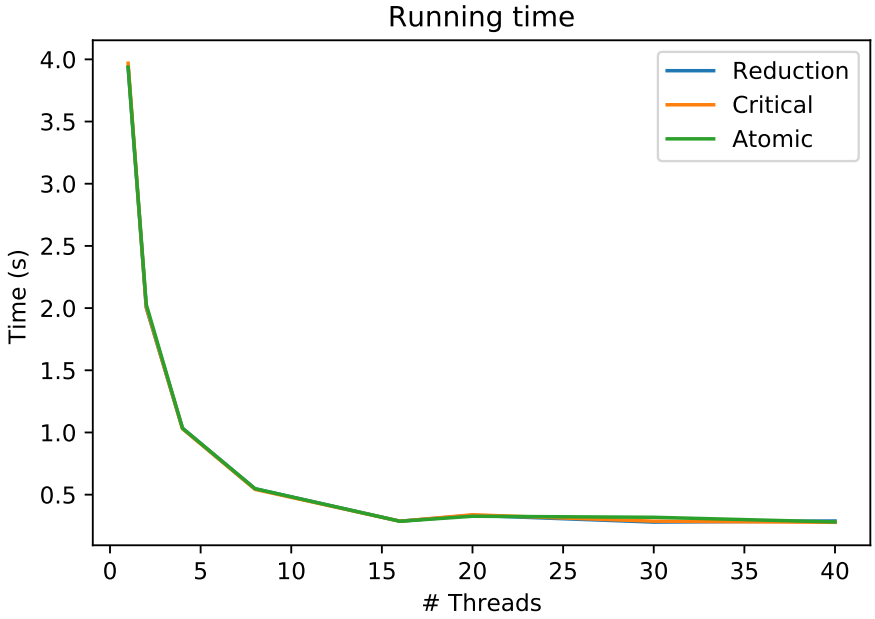

Figure 1: Running time as a function of number of threads

3 Exercise 2

Here the results obtained for the second assignment are shown. Ten threads and two chunk sizes were used. The code was also run for one single thread, which of courses yielded a straight line of asterisks. The results obtained for “static” and “dynamic” alone, without specifying the chunk sizes have been omitted as the size is set to 1 by default. All the results are expected.

Figure 2: Static scheduler, chunk of size 1

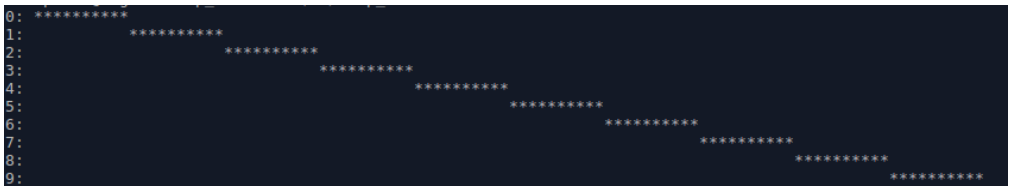

Figure 3: Static scheduler, chunk of size 10

Figure 4: Dynamic scheduler, chunk of size 1

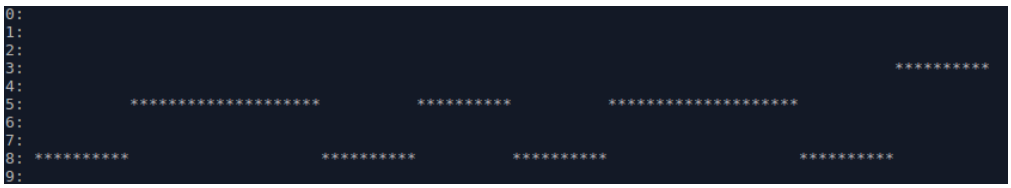

Figure 5: dynamic scheduler, chunk of size 10