Assignment 1

Marco Franzon

August 2020

1 Introduction

This analysis aims to implement an algorithm that computes the approximation of π in both a serial and parallel manner. 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.

2 Exercise 1

The first task is the approximation of π using numerical integration. In particular, π can be approximated using the mid-point quadrature rule.

pi.c implements both a serial and parallel version of the approximation. The provided **makefile** compiles the code into an executable which prints the approximation of π and the execution time for both the serial and parallel computation.

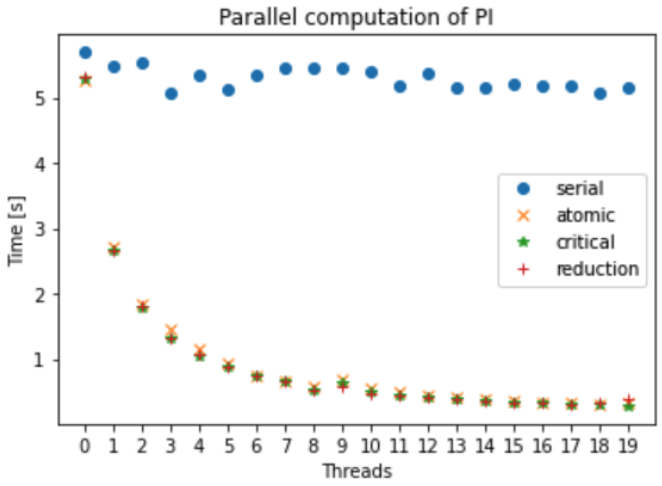

Figure 1: Strong scaling, $N = 10^9$

3 Exercise 2

The second exercise deals with OpenMP loop schedules. The aim is to visualise how different schedules distribute work among the available processors. **loop_schedules.c** implements a solution for the exercise. The included makefile can be used to compile the code. The implementation allocates an array of dimension N which is subsequently filled by availability : each thread will fill part of the array, following the adopted schedule.

The function **print_usage** (already provided) has been used to visualize the result. The comparison of the behaviour of static and dynamic schedules with different chunk sizes is performed using only one omp parallel region

Consider only the case N = 40 and 10 threads:

- static equally distributes the work among the threads. In particular, each thread will be assigned a chunk of N/nthreads, following the order of the threads;
- static, chunk size = 1 equally distributes the work among the threads. In particular, each thread will be assigned a chunk of size 1, following the order of the threads;
- static, chunk size = 10 equally distributes the work among the threads. In particular, each thread will be assigned a chunk of size 10, following the order of the threads;
- dynamic does not necessarily distributes the work equally among the threads. The work distribution is determined at run-time, using no constraints on chunk size;
- dynamic, chunk size = 1 does not necessarily distributes the work equally among the threads. The work distribution is determined at run-time, using chunk of size 1;
- dynamic, chunk size = 10 does not necessarily distributes the work equally among the threads. The work distribution is determined at run-time, using chunk of size 10.

In this particular case using a static schedule guarantees a better workload balancing. It should be underlined that this is not always the case: a static schedule with a chunk size of 20 would result in a highly unbalanced workload distribution. Moreover, if each iteration takes different time to be completed, work unbalance may arise as well. Consider the case where the iteration time (in our case, the time to write an element of the array) increases with the iteration number: static will equally distribute iterations, but actual workload will be unequally distributed, the last threads taking much longer to execute their iterations. If this is the case, then a dynamic scheduling would perform better. Another reason to consider when choosing between static and dynamic scheduling is overhead: sharing workload is done at compile time with static

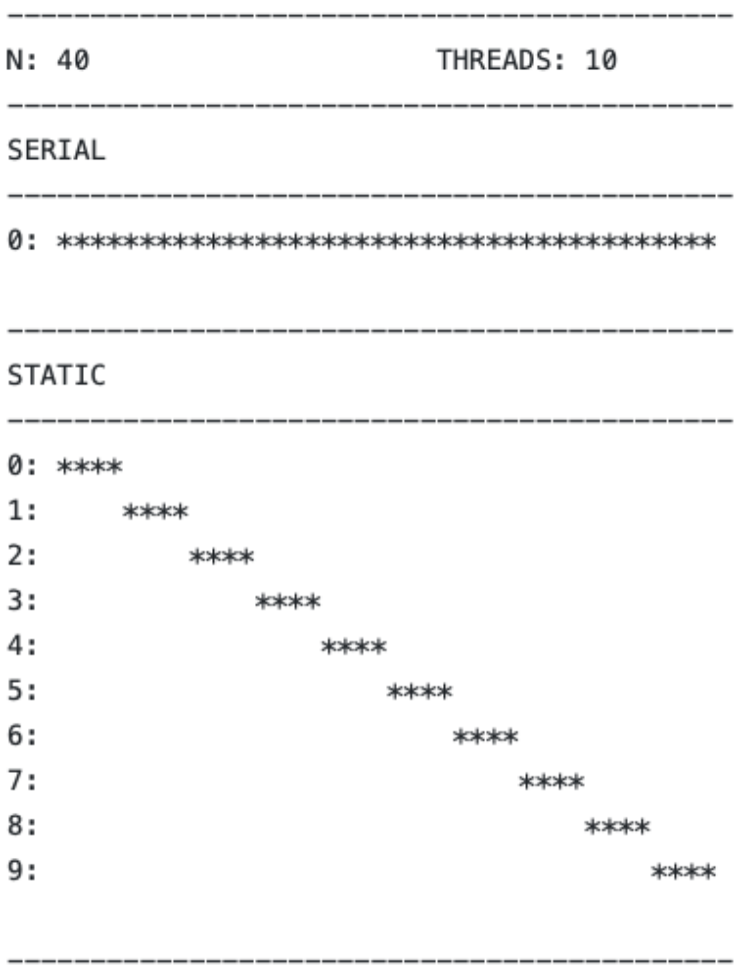

Figure 2: Loop schedules examples

scheduling and at run time with dynamic scheduling (which means additional overhead for communication).