High Performance Computing

CS-301
Prof. Bhaskar Chaudhry
Due Date: 24/02/2021

Pratvi Shah, 201801407
Arkaprabha Banerjee,201801408
Lab Date: 24/02/2021

Lab 4

Hardware Details:

- CPU = 4
- Socket = 1
- Cores per Socket = 4
- Size of L1d cache = 32K
- Size of L1i cache = 32K
- Size of L2 cache = 256K
- Size of L3 cache = 6144K

1 Introduction

In this report we shall be comparing serial and parallel implementations for the Monte Carlo Simulation to find the value of pi. The primary focus would be on various random generators and how they function in the parallel paradigm :

- Implement the serial version of the algorithm and analyze the relevant metrics.
- In order to make the code parallel, we shall try to divide the computations equally between the specified number of threads. The primary focus would be on various random generators. We shall be implementing via 3 main categories :
  1. No seed value
  2. One global seed value
  3. Thread specific seed value

Question 1: Calculate value of π using Monte Carlo Simulation

Implementation Details:

1. Description about the Serial implementation

In serial implementation a single loop runs for all the N(problem size) steps. We shall normalize the random values of x and y coordinates between -1 to +1 and find out the number of values which lie inside the unit circle. The ratio of the number of points inside the circle and the problem size multiplied by 4 gives us the desired value of π.

2. Description about the Parallel implementation

In this section we have implemented various methods and for the same we used the $rand\_r(\&seed)$ function then divided that by $RAND\_MAX$, to get it within the limit of (-1,1), with different values of seed being discussed in seed values part below. This function is known to be thread safe when the seed value is unique for each thread.

- Seed Values :
    1. In the first method we had no seed value for our random function. This would lead to a different set of random numbers being generated each time we run the loop.
    2. We then used a global seed value for all the available threads.
    3. Lastly in order to solve the problem of random generators not being thread safe we initialize the seed value as the thread id for each specific thread. This would lead to a different sets of random number being generated in each thread in contrast to the ones generated via global seed.
- Locking Mechanisms :
    1. **Reduction :** The reduction clause is implemented in all the parallel cases so as to protect the modification of the 'count' variable which counts the number of points inside the unit circle.
    2. **Critical :** It essentially makes a particular section accessible to a single thread at a time and in essence serializes that section of the code.
- Scope of seed :
    1. **Thread Private :** In order to solve the problem of thread safety for a global seed variable a thread private variable is used which makes a copy of the existing variable for that specific thread and is dependent upon the thread id hence solving the problem.
    2. Another Naive methodology is to just declare the seed variables inside the parallel region and declare them accordingly.

       Both the methods work in exactly the same way for all associated metrics and has been verified experimentally .Hence 'Thread Private' Legend is used interchangeably for both these methods.

Complexity

We consider complexity in terms of the number of computations done by each thread or each processor.

Complexity of serial code:

In serial code total number of computations are same as the problem size so, complexity of serial code is $O(N)$.

Complexity of parallel code:

In methods which explicitly define the for loops the number of computations are equally divided amongst all the threads. For the implicit methods we assume that similar condition will arise. Hence, the complexity of any parallelisation will be $O(N/P)$.

Cost of Parallel algorithm

We define the cost complexity of parallel implementation as the parallel run-time multiplied by the number of processors.

$$cost = \frac{N}{P} * P = N \tag{1}$$

For serial Code, cost complexity and time complexity are the same.

$$cost = N \tag{2}$$

Theoretical Speed Up:

We use above equations, the asymptotic approximation of speedup, to calculate the theoretical speedup for different number of threads.As one thread runs on one processor, we can safely approximate the speedup.

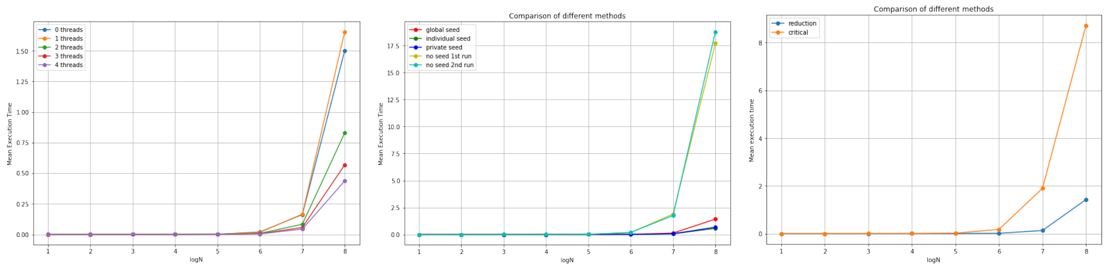
No. of threads	**Speedup**
1	P=1
2	P=2
3	P=3
4	P=4

Curve Related Analysis

For our experiments we have varied N from 10 to 10^8 in steps of powers of 10 and considered 1,2,3 and 4 threads(or processors P) for each value of N.

Time Curve Related analysis:

The time taken to complete the execution is directly proportional to the problem size(N) as expected. We can further observe that as the number of processors/threads increase for the same implementation the time reduces, with some exceptions arising due to random nature of the system.



(a) (b) (c)

Figure 1:
(a) Mean Execution time for parallelisation Techniques with varying threads
(b) Mean Execution time for different Techniques with 3 threads
(c) Mean Execution time for different locking techniques with 3 threads

We notice that in general with an increase in the number of threads the mean execution time reduces. This is because that the work is being divided to a larger number of threads. However on comparing the serial execution time and the parallel execution with 1 thread we find that the parallel implementation takes a slightly higher amount of time. This is because even though they are conceptually similar yet the parallel one requires a higher amount of overhead.

The following observations were also noted:

1. Whenever have a common/ no seed value among all threads the mean execution time increases. This is because the inbuilt random function has an internal mutex locking mechanism for every seed value. Hence when multiple calls to the random function are made at the same time the wait time increases resulting in the increase in the mean execution time.

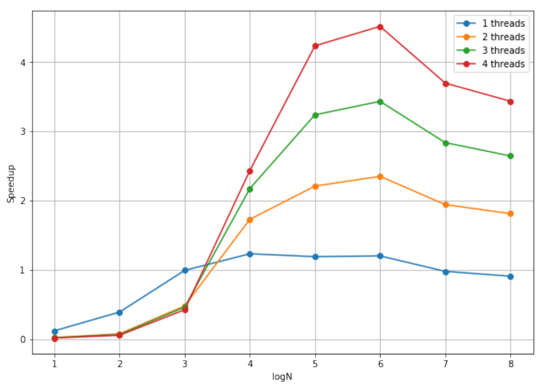
2. This can be rectified by using a thread safe function. We could either have different seed values for each thread or we can make our own random number generator. For the purpose of this experiment, different thread id as seed values, for each thread have been chosen.

3. The reduction clause takes a significantly lower amount of time as compared to the critical clause. This is on account of the reduction algorithms which is better than completely serializing the code (as present in critical).

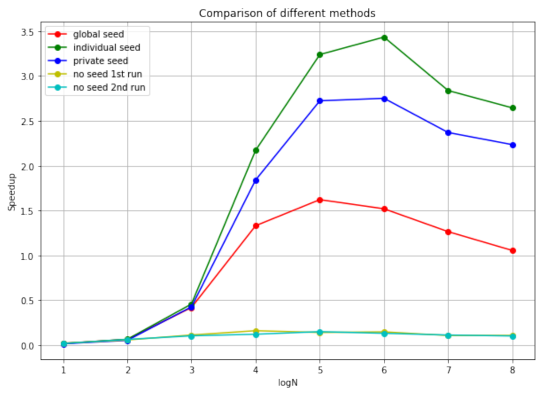
4. Both thread private clause and defining variable inside the parallel region gave the same valaue for mean time.

Speedup Curve Related analysis

We observe that by increasing the number of threads increases the performance but the maximum performance remains within the theoretical limit except in a few cases.



(a)



(b)

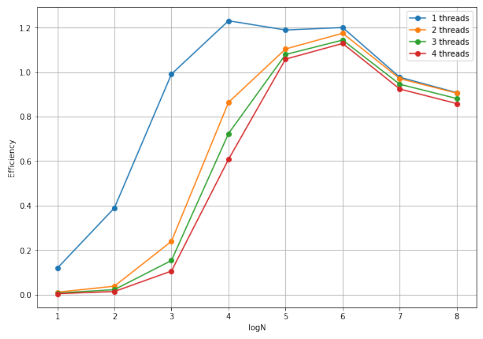
Figure 2:
(a) Speedup for parallelisation Techniques with varying threads
(b) Speedup for different Techniques with 3 threads

We observe the speedup is a direct consequence of the mean execution time, i.e higher the mean execution time, lower the speedup. Although theoretically the speedup shouldn't exceed the number of processors (3 in this case), however we observe a super-linear speedup in some cases (> 3). This is because we have a different cache for each thread in the parallel domain, hence their speedup with respect to the standard serial code (which has a single cache for computation) exceeds the theoretical value.Compiler optimization could also be another reason for this.

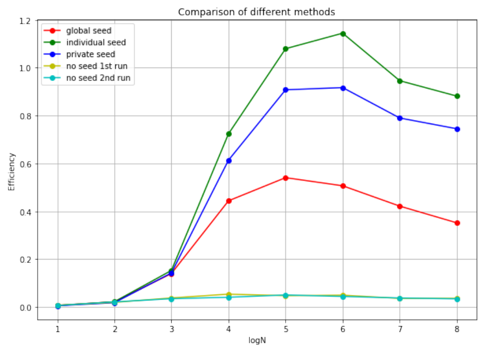
Efficiency Curve Related analysis

Although the net speedup increases with an increase in the number of processors , however, we can notice that the efficiency reduces as the number of threads/processors increase for a given problem size. So we could conclude from our experiment that the efficiency of an algorithm depends on the level of ease of synchronisation amongst the threads/processors and the parallel overhead.Even if the net speedup is higher however the work done per processor maybe inefficient.

The trend for various implementation mechanisms etc still remain the same (similar to speedup) in a relative sense.



(a)



(b)

Figure 3:
(a) Efficiency for simple parallelisation with varying threads
(b) Efficiency for parallelisation Techniques with 3 threads

Speedup v/s Number of cores analysis

.

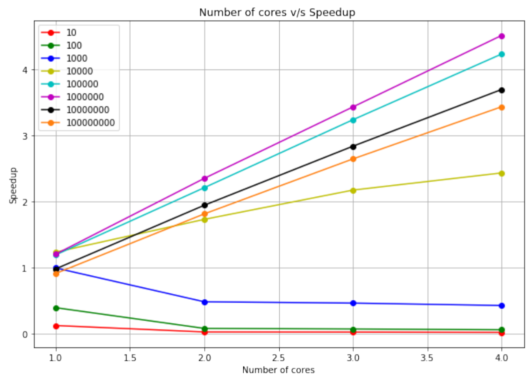
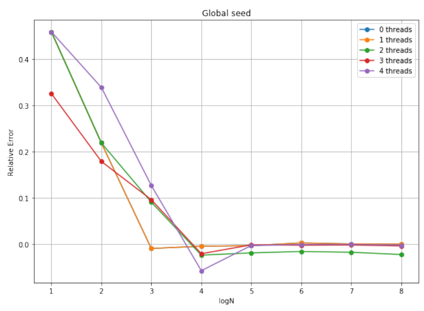


Figure 4: Speedup v/s Number of cores for a given parallelisation Technique

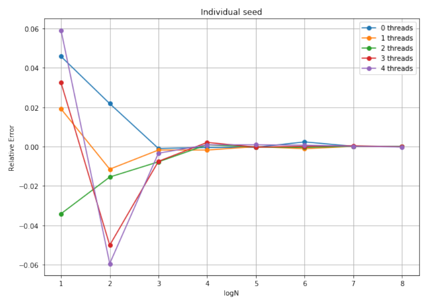
We can see from the plots that the speedup increases with increase in the number of cores for most problem sizes.For very low problem size we observe the reverse trend. That is perhaps due to the fact that the overhead exceeds the speedup gained by optimization. We could also observe that the maximum speedup that can be achieved for a given number of threads could exceed the number of threads used(P).This super-linear speedup is because each thread has its own cache and the reduced problem size fits into these cache resulting in this pattern.

We observe that for a higher problem size, with the increase in number of threads, the speedup increases. This is because the parallel overhead increases with a lower rate as compared to the problem size hence after a particular size it becomes negligible and we see a really good speedup. For lower values we see the opposite because the parallel overhead , overweighs the speedup. We also see that they tend to saturate after a point. This is because no more speedup via parallelization can be achieved for this particular implementation.

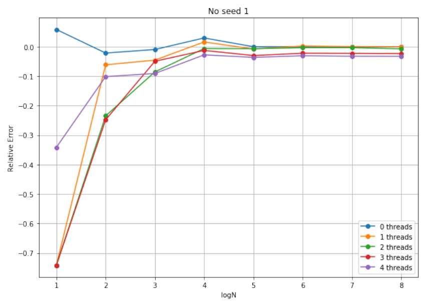
Accuracy analysis



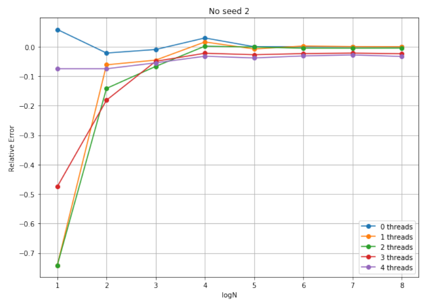
(a)



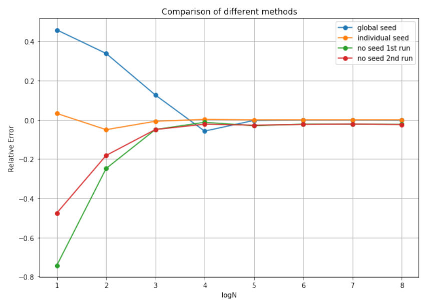
(b)



(c)



(d)



(e)

Figure 5:
(a) Relative error with global seed value for varying threads
(b) Relative error with thread specific seed value for varying threads
(c)Relative error with no seed value for varying threads - RUN 1
(d) Relative error with no seed value for varying threads - RUN 2
(b) Comparative Analysis of Relative error with 3 threads

Following are the primary observations :

1. As we increase the problem size is considerably increased, the net error slowly converges to 0 irrespective of the method used. This is because we will have enough points to gain a fairly good estimate of Pi.

2. When we dont initialize the seed value then every time we shall obtain a different value of Pi.

3. For lower values of the Problem size the thread specific values of seed will perform much

better than a global value of seed. This is because in the global seed value we will obtain the same random values in all the threads. Thus it wont help in getting a good estimate of Pi. On the other hand when we initialize a thread specific seed value we obtain different set of random values for all the threads thus giving us a better estimate of Pi.

4. Since the same set of random numbers will be generated in each thread for global seed, hence on increasing the number of threads the actual number of random number reduces. Hence, the relative error increases with increase in the number of threads for this case.