High Performance Computing

Shantanu Tyagi, Kirtan Delwadia
201801015, 201801020

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

Lab Date: 24/02/2021

Assignment 4

Hardware Details:

- Architecture: x86_64
- Byte Order: Little Endian
- CPU(s): 4
- Sockets: 1
- Cores per socket: 1
- Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 6144K

Question 1: Calculating π using Monte Carlo Simulation

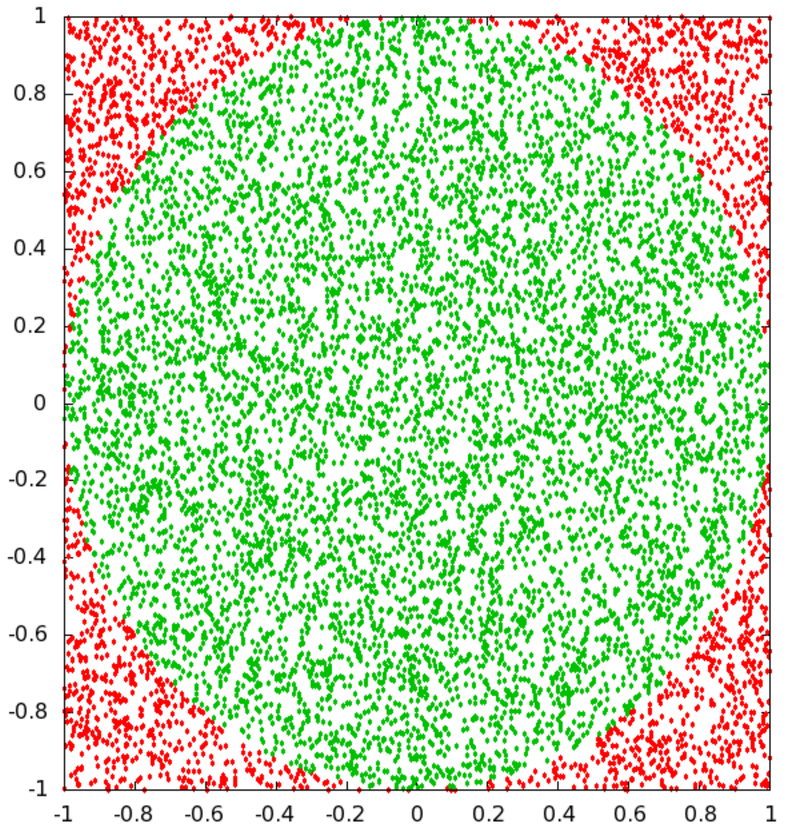


Figure 1: Source: http://www.physics.smu.edu/fattarus/MonteCarloLab.html

Implementation Details:

1. Description about the Serial implementation

We numerically evaluate π using Monte Carlo simulations using a single loop that runs for the entire length of the problem(N) generating random numbers x and y which lie between -1 and 1, giving us the size of square these random numbers will be forming as 2. Now we can inscribe a circle of radius 1 in this square and keep counting the how many times(n) $x^2 + y^2 \leq 1$, i.e. lying inside the circle. We can then find the ratio of area of circle to that of the square as $\frac{n}{N} = \frac{\pi * R * R}{2R * 2R}$ giving, $\pi = \frac{4n}{N}$ where R(radius) = 1.

```
srand(SEED);
count=0;
for ( i=0; i<niter; i++) {
  x = (double)rand()/RAND_MAX;
  y = (double)rand()/RAND_MAX;
  z = x*x+y*y;
  if (z<=1) count++;
  }
pi=(double)count/niter*4;
```

2. Description about the Parallel implementation

In parallel programming, we divide the entire problem into sub problems according to the number of cores/processors available and each processor calculates the result of the sub problem assigned to it. The serial code is summing up a variable(*count*) and the end of the loop. Thus, we can divide this task among multiple cores and take the sum of the of the *count* variables calculated by each core to a shared variable.
One problem that we encounter here is that *rand*() function is not thread safe and also, we cannot use a global seed value because then all the threads will generate same random numbers. *rand*() function's performance will be affected if there is a race condition between threads because its next value is determined by its current value. Instead we can use a thread safe function, *rand_r*() and put a different seed value in each thread.

1. **Atomic**

```
float count = 0;
#pragma omp parallel
{
        int i;
        int c= 0;
        unsigned int thd_id = omp_get_thread_num();
        #pragma omp for private(i)
        for(i=0;i<n;i++){
                float x = rand_r(&thd_id)/(float)RAND_MAX;
                float y = rand_r(&thd_id)/(float)RAND_MAX;

                if((x*x + y*y) < 1)
                c = c + 1;
        }
        #pragma omp atomic
                count = count + c;
}
```

2. **Critical**

```
float count = 0;
#pragma omp parallel
{
        int i;
        int c= 0;
        unsigned int thd_id = omp_get_thread_num();
        #pragma omp for private(i)
        for(i=0;i<n;i++){
                float x = rand_r(&thd_id)/(float)RAND_MAX;
                float y = rand_r(&thd_id)/(float)RAND_MAX;

                if((x*x + y*y) < 1)
                c = c + 1;
        }
        #pragma omp critical
                count = count + c;
}
```

3. **Reduction**

```c
omp_set_num_threads(p);
int c= 0;
#pragma omp parallel
{
        int i;
        unsigned int thd_id = omp_get_thread_num();
        #pragma omp for reduction(+:c)
        for(i=0;i<n;i++){
                float x = rand_r(&thd_id)/(float)RAND_MAX;
                float y = rand_r(&thd_id)/(float)RAND_MAX;
//                          printf("%d %f %f\n",i,x,y);
                if((x*x + y*y) < 1)
                c = c + 1;
        }
}
```

4. **By using threadsafe**

```c
int i;
double x, y;
double sum = 0.0;
double Q = 0.0;

omp_set_num_threads(P);

#pragma omp threadprivate(seed)
#pragma omp parallel for private(x,y) reduction(+:sum)
for(i = 0; i < N; i++)
{
    seed = omp_get_thread_num()+1;
    //rand_r() is thread safe
        x = rand_r(&seed)/(float)RAND_MAX;
        y = rand_r(&seed)/(float)RAND_MAX;

        if((x*x + y*y) < 1)
        {
                sum = sum+1.0;
        }
}
Q = 4.0*sum*1.0/N;
printf("%.9g\n", Q);
}
```

Complexity:

1. Complexity of serial code:

The time varies linearly with the problem size since we just have one loop which runs for n times. When the problem size is n, we have the time complexity as $O(n)$.

2. Complexity of parallel code:

Here again we have a problem with size n, however it is divided into p processors each of which runs in parallel. This means that the loop runs $\frac{n}{p}$ times on each processor giving us a time complexity of $O(\frac{n}{p})$.

3. Cost of Parallel algorithm

Cost is the product of time complexity and number of processors which is this case will be $p * \frac{N}{p} = N$.

4. Theoretical Speed Up:

For a problem of size n and total p processors, we have theoretical speed up as $\frac{n}{(\frac{n}{p})} = p$
Thus for 1 processor, we get $\frac{n}{(\frac{n}{1})} = 1$
and for 4 processors, we get $\frac{n}{(\frac{n}{4})} = 4$ Speedup is calculated as ratio of time taken to execute the code serially to time taken to execute the code parallelly. The expected speed up is p but this if often not achieved due to memory access, parallel overhead, and thread synchronisation.

5. Memory Accesses:

For a problem of size N, we are generating 2 random numbers for each iteration thus giving us total memory accesses of $2N$.

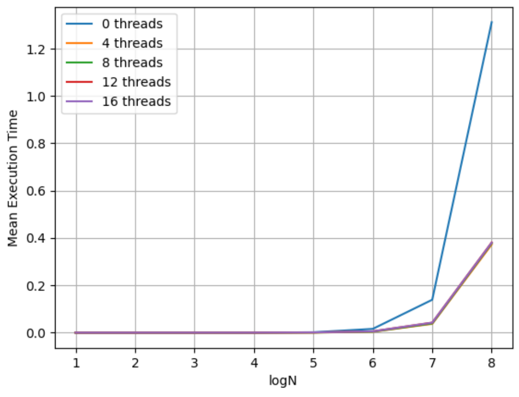
5. Computations:

For each iterations, we perform 2 additions, divisions and multiplications giving us a total of $6N$ operations in case of serial and $\frac{6N}{p}$ in case of parallel. At the end of the loop, we perform 1 multiplication and division for both serial and parallel but parallel also has $p - 1$ additions to add the result of all threads before this.

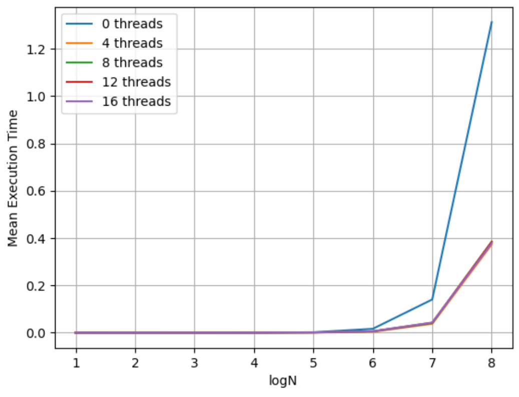
Curve Related Analysis:

1. Time Curve Related analysis:

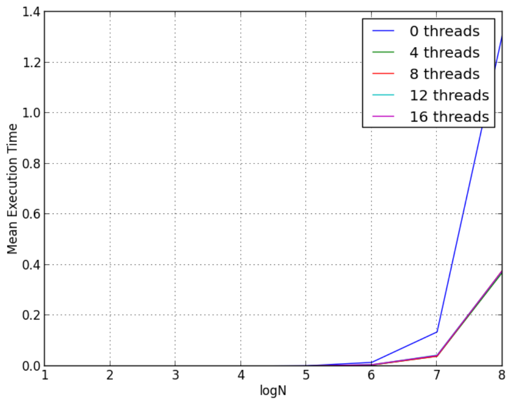
It is observed that as the number of processors on which the code runs increases the time taken for executing or calculating the result reduces. Therefore, the time for executing the code is inversely proportional to the number of processors on which the code is working since the computations get divided on more processors as opposed to less processors, hence reducing the computation time. As the problem size increases, the time taken also increases for both serial and parallel implementations. This is because with increased problem size, more calculations need to be done and the loop runs longer. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results. On the other hand, for smaller input sizes, serial is faster than parallel because of parallel overhead and thread synchronization.



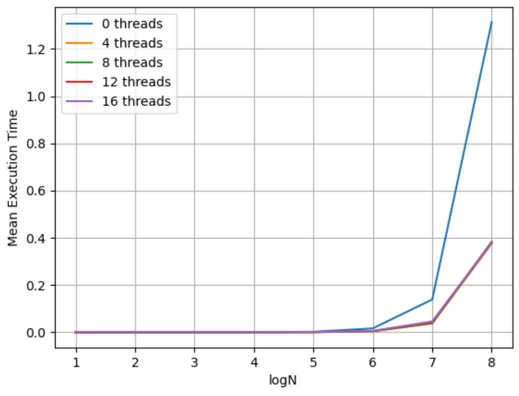
(a) Atomic



(b) Critical



(c) Reduction

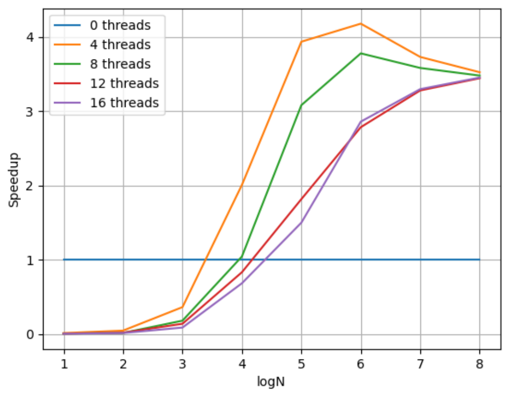


(d) Thread Private

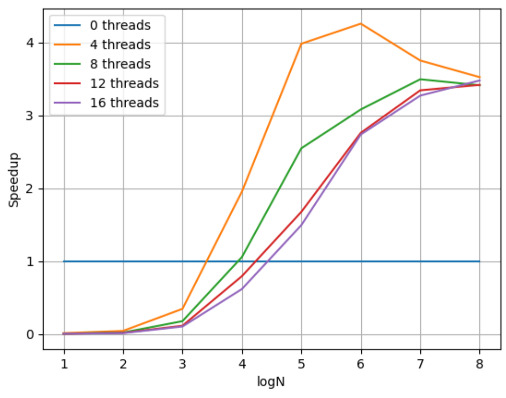
Figure 2: Mean Execution Time

2. Speedup Curve Related analysis:

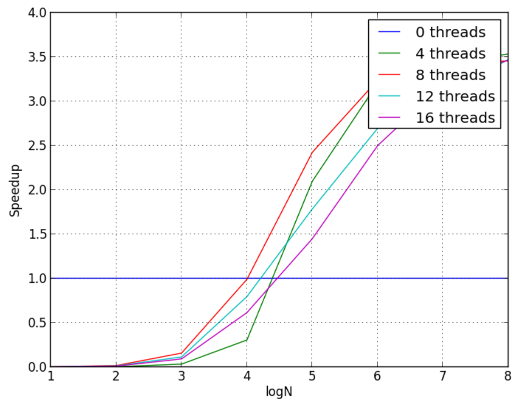
The speed up increases as the number of processes increases on which the code runs. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results. As the problem size increases initially the parallel threads have a speedup less than one due to parallel overhead and pipeline not being utilised efficiently, however as size increases the speed increases because the pipeline is now being used efficiently, the speed up reaches closer to 4. The deviation in these values from the theoretical values depicts that for large problem size the memory access time also comes into consideration. It is also noted that for small problem sizes the speedup of 1 thread is greater than speedup of multiple threads which might be due to parallel overhead. The speed up remains almost constant for larger problem sizes because marginal gain in speedup for every new core added keeps reducing



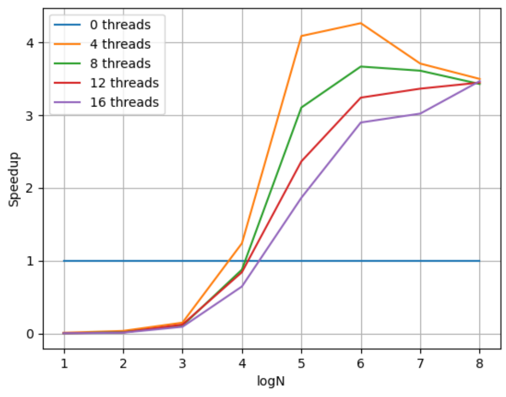
(a) Atomic



(b) Critical



(c) Reduction

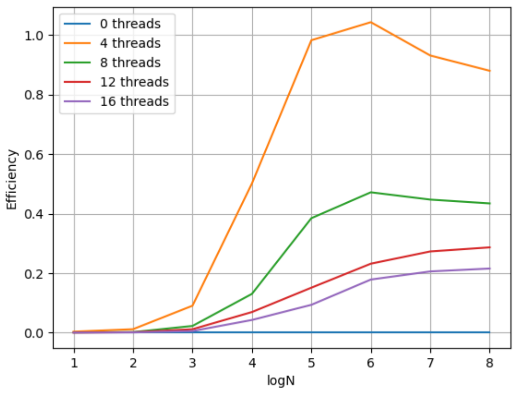


(d) Thread Private

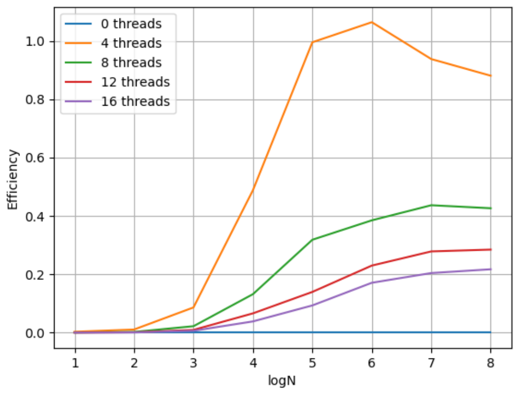
Figure 3: Speedup

3. Efficiency Curve Related analysis:

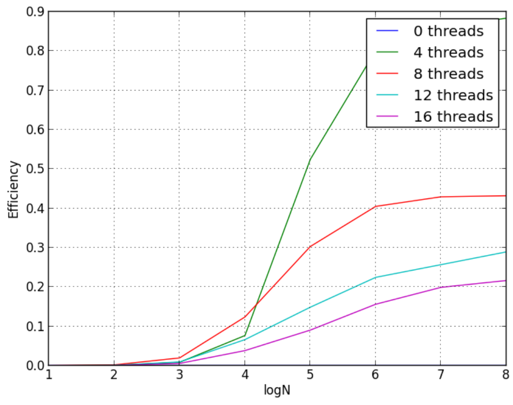
Efficiency is speedup divided by the number of threads, p. It is also observed that with greater number of processors working together the efficiency also decreases. The greater number of threads lesser will be the efficiency because managing so many threads also requires time and the parallel overhead starts dominating. Cost of creating more and more threads is not able to balance how much more performance it gives as compared to lesser number of threads. As the problem size increases the efficiency also gradually starts increasing for a given implementation and gradually reaches 1.



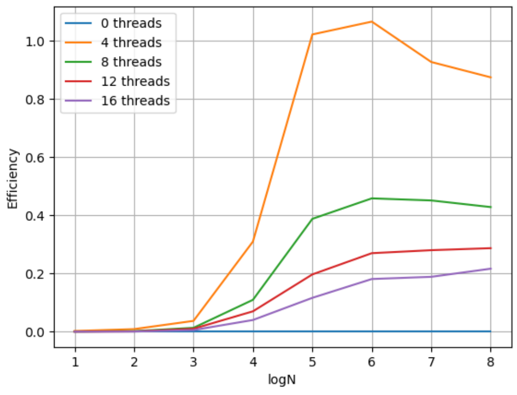
(a) Atomic



(b) Critical



(c) Reduction



(d) Thread Private

Figure 4: Efficiency

4. Conclusion:

It is seen that as the problem size increases, the parallel approach gives much better results than the serial approach. Moreover, the speedup for larger problem size remains constant thus the parallel approach is highly scalable. Atomic implementation was the fastest.