CS 301
High-Performance Computing

Lab 5 - Calculation of Pi using Random Numbers

Rakshit Pandhi (202201426)
Kalp Shah (202201457)

March 31, 2025

Contents

1 Problem - Calculation of Pi using Random Numbers

1.1 Brief description of the problem

The Monte Carlo method is a probabilistic approach used to estimate values through random sampling. In this program, we use it to approximate the value of (Pi). The key idea is based on the geometric probability of randomly generated points falling inside a quarter-circle within a square.

1.2 Description of Algorithm

Define the Square and Circle

A square of side length 1 is considered with a quarter-circle inscribed within it, having radius $R = 1$.

Generate Random Points

Random points (x, y) are generated within the square using a uniform distribution.

Check if Points Lie Inside the Circle

The equation of a circle is:

$$x^2 + y^2 \leq R^2$$

If a point satisfies this condition, it is inside the quarter-circle.

Calculate the Ratio

The probability of a randomly chosen point falling inside the circle is approximately equal to the ratio of their areas:

$$\frac{\text{Points in Circle}}{\text{Total Points}} \approx \frac{\frac{\pi R^2}{4}}{R^2} = \frac{\pi}{4}$$

Rearranging:

$$\pi \approx 4 \times \frac{\text{Points in Circle}}{\text{Total Points}}$$

Iterate for Higher Accuracy

More points lead to a more accurate estimate of π.

1.3 Complexity of the Algorithm

Serial

The time complexity is O(N), where N is the number of random points generated. More iterations improve precision but increase execution time.

Parallel

- The serial version has a complexity of:
$$O(N)$$
- The parallel version distributes computation, reducing the per-thread workload to:
$$O\left(\frac{N}{P}\right)$$
- However, OpenMP reduction introduces an additional $\log P$ synchronization overhead, giving a final complexity of:
$$O\left(\frac{N}{P} + \log P\right)$$

Synchronization Overhead

- The `reduction(+:sum,flops)` operation introduces $O(\log P)$ overhead for summing the partial results across threads.

1.4 Compute to Memory Access Ratio

Total compute operations per iteration:
C=2(rand)+2(square)+1(add)+1(compare)+1(increment)=7
Total memory accesses per iteration:
M=2(rand loads)+1(counter read)+1(counter write)=4
Therefore CMA is 7/4(moderately compute bound).

1.5 Memory Bound versus Compute Bound

From calculation we can see that the CMA is slightly more than 1 , suggesting algorithm spends more time in computing as in memory access. Hence we can say that this is a comoute bound problem.

1.6 Hardware Details

1.6.1 Hardware Details for LAB207 PCs

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 12
- On-line CPU(s) list: 0-3
- Thread(s) per core: 2

- Core(s) per socket: 6
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel Core i5-12500
- Stepping: 3
- CPU MHz: 799.992
- CPU max MHz: 4.6G
- CPU min MHz: 800.0000
- BogoMIPS: 6584.55
- Virtualization: VT-x
- L1d cache: 288K
- L1i cache: 192K
- L2 cache: 7.5M
- L3 cache: 18M
- NUMA node0 CPU(s): 0-3

1.6.2 Hardware Details for HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 24
- On-line CPU(s) list: 0-23
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 2
- NUMA node(s): 2

- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- Stepping: 2
- CPU MHz: 1419.75
- BogoMIPS: 4804.69
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K
- NUMA node0 CPU(s): 0-5,12-17
- NUMA node1 CPU(s): 6-11,18-23

1.7 Optimization Strategy

Optimization	**Impact**
OpenMP Parallelization	Reduces execution time by leveraging multiple CPU cores
Reduction for Safe Aggregation	Prevents race conditions without locks
Thread-Safe Random Number Generator (`rand_r`)	Improves performance by avoiding contention
Private Thread Variables	Eliminates data races and memory conflicts
Local Variable Optimization	Minimizes memory latency and cache misses
Loop Optimization (`#pragma omp for`)	Ensures efficient workload distribution

Table 1: Optimization Strategies and Their Impact

1.8 Algorithm Time versus Problem Size

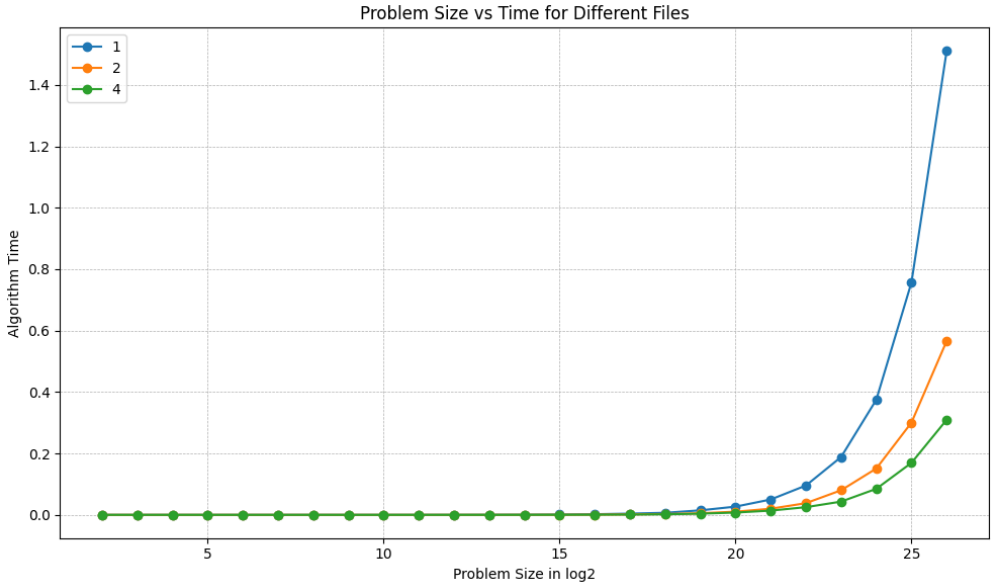

Figure 1: Screenshot from Lab-207 PC

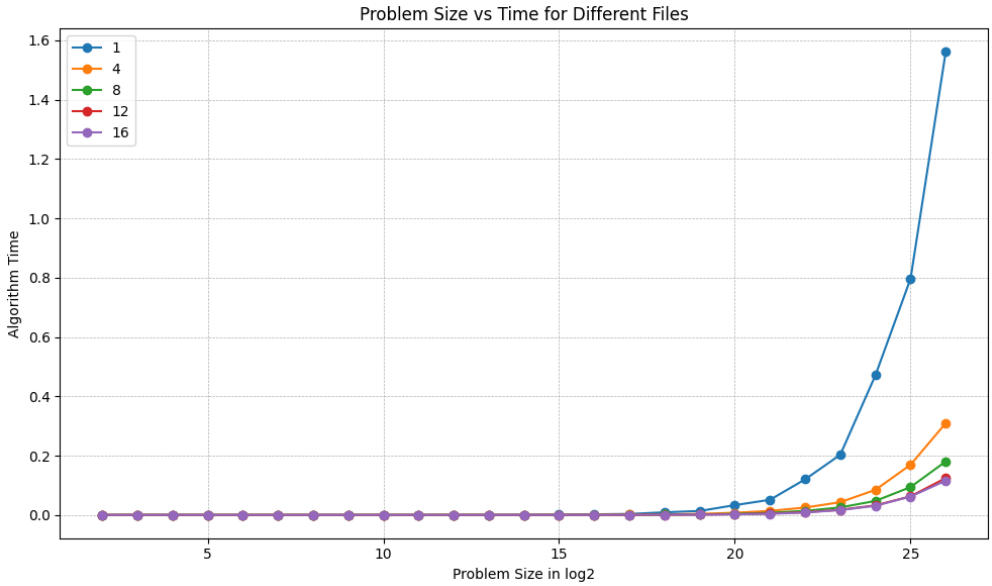

Figure 2: Screenshot from HPC Cluster

1.9 Speedup

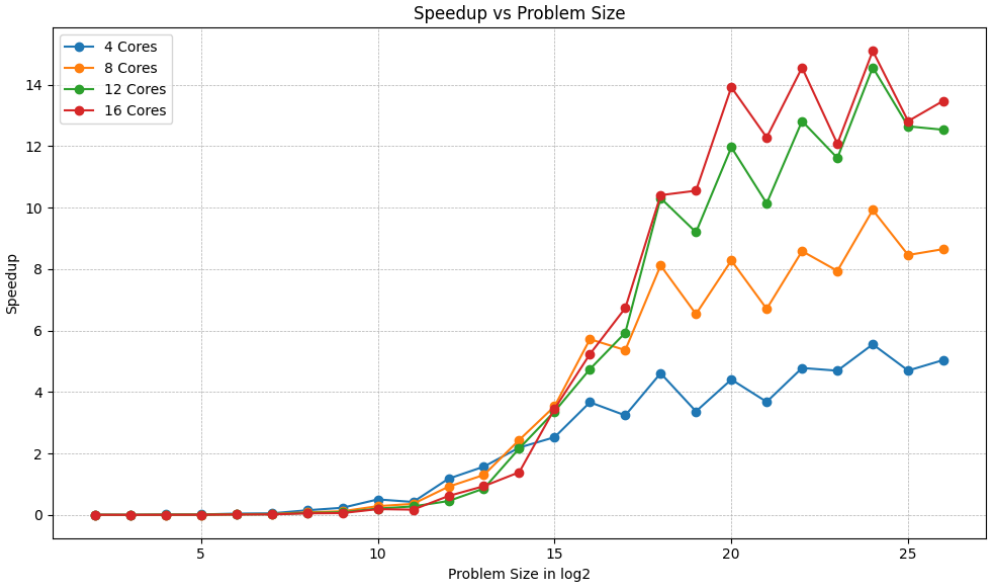

Figure 3: Screenshot from HPC Cluster

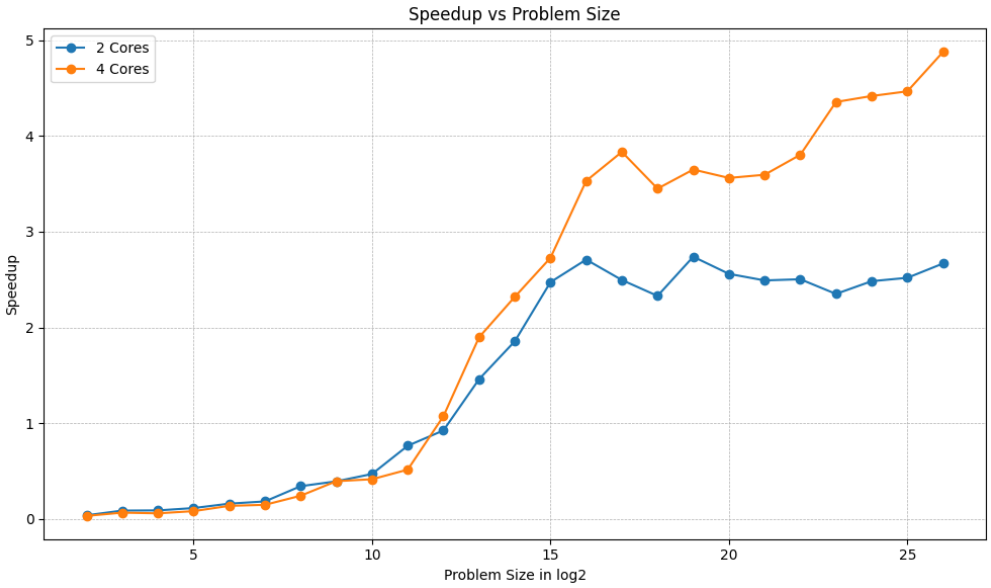

Figure 4: Screenshot from LAB207 PC

Theoretical speedup should be 1/number of processors. But it is not seen practically because of parallel overheads and all other reasons.

1.10 Issues with Parallel Random Number Generation in OpenMP

- **Race Conditions on `rand()`**: `rand()` is not thread-safe, leading to incorrect or non-deterministic results due to interference between threads.
- **Lack of Independent Random Sequences**: A single global seed (`srand(time(NULL))`) can cause all threads to generate the same sequence, reducing randomness.
- **Poor Random Number Distribution**: Shared seeds may lead to clustering, affecting statistical accuracy in Monte Carlo simulations.
- **Thread Contention & Overhead**: Multiple threads accessing `rand()` create contention; using locks (`#pragma omp critical`) adds overhead, reducing performance.

2 Conclusions

- We were able to see nearly super linear speedup when we tried to parallize the code.
- OpenMP enables efficient parallelization by automatically distributing workload across multiple threads, allowing computations to be executed concurrently rather than sequentially. This significantly reduces execution time by utilizing the full potential of multi-core processors.
- Commands like `#pragma omp parallel for` ensure that loop iterations are divided among threads, while Additionally, OpenMP provides built-in synchronization mechanisms such as `#pragma omp reduction`, which ensures safe aggregation of results while minimizing bottlenecks.