Parallel and Distributed Systems - HW 2 Report

Ahmet Özdemir

200104004062

Monte Carlo Estimation of π

The Monte Carlo method is a probabilistic method used to approximate the value of π. It is based on the rate at which randomly generated points fall within a unit circle. In this study, three different methods (single-core C program, Pthreads and MPI) are used to estimate π and their performance is compared.

Single Core C Program (Q2)

- Generate random coordinates and count the points satisfying the equation of the unit circle.

MPI Program (Q3)

- The total number of tosses is divided equally between all processes.
- Each process calculates its points in the circle and the results are combined using MPI_Reduce.
- Distributing the total number of shots with `MPI_Bcast`.
- Aggregation of results with `MPI_Reduce`.

Pthreads Program (Q4)

- Tosses are split between threads.
- Threads add the number of points in the circle to the global variable.
- Data integrity is ensured by using mutex.

Some experiment result from the table

Method	Number of Tosses	Thread/Process Number	Time (s)	Estimated π
C	1,000,000	1	0.0461106300354004	3.138932
MPI	1,000,000	4	0.688761234283447	3.143164
Pthreads	1,000,000	2	0.0199477672576904	3.143764

Analysis Section

1. Comparison of Single-Core C, MPI, and Pthreads Implementations

1. - **At Low Toss Counts (10^6):**
     - **MPI** shows the slow performance.
       - **Reason**: The communication overhead in MPI, particularly the operations `MPI_Bcast` (broadcast) and `MPI_Reduce` (aggregation), overshadows the advantages of parallel computation when the computational load is low. The cost of inter-process communication becomes significant relative to the small amount of computation being performed.
     - **Single-core C implementation** performs faster as it does not suffer from communication overhead.
     - **65536 Thread implementation** shows the slowest performance.
       - **Reason**:
         - **Thread Overhead**: Creating and managing a very large number of threads (e.g., 65536) incurs significant overhead. Each thread requires system resources such as stack memory, scheduling time, and synchronization, which add up to considerable delays.
         - **Context Switching**: The operating system must frequently switch between threads to share CPU time, leading to excessive **context switching overhead**. This results in reduced computational efficiency because time is spent switching rather than performing actual computations.
         - **Resource Contention**: When the number of threads far exceeds the number of physical or logical CPU cores, threads compete for CPU time and other resources (e.g., cache, memory bandwidth), causing significant contention and performance degradation.
     - **Pthreads implementation** is the fastest when using more than one thread.

        - **Reason**: Pthreads utilize shared memory, avoiding the cost of inter-process communication and benefiting from efficient thread management. This makes it well-suited for low computational loads where shared memory access is faster than communication-heavy alternatives like MPI.

2. - **At Medium Toss Counts (10^7):**
        - **MPI significantly outperforms** the single-core C implementation.
            - **Reason**: As the computational load increases, the communication overhead in MPI becomes negligible compared to the advantages of parallel computation.
            - **Observation**: MPI's execution time (0.14s) is faster than the single-core C implementation (0.50s).
        - **Pthreads implementation** shows slightly slower performance (0.18s) than MPI.
            - **Reason**: While threads benefit from shared memory, managing threads still incurs overhead, particularly when the computational load increases but is not large enough to saturate the available cores.

3. - **At High Toss Counts (10^8 and Above):**
        - **MPI** continues to **dominate** the single-core C implementation.
        - Observation: For 10^8 tosses:
            - MPI: **0.26s**
            - C: **4.0s**
        - **Reason**: The large computational workload allows MPI to leverage its parallelism effectively. Communication overhead becomes negligible compared to the time saved by distributing computations across processes.
        - **Pthreads implementation** still demonstrates the best performance for massive workloads but begins to plateau.
            - **Observation**:
                - For 10^9 tosses:
                    - MPI: **2.10s**
                    - Pthreads: **3.74s**
                    - C: **38.66s**
            - **Reason**:
                - Threads experience diminishing returns due to resource contention when too many threads are used (e.g., oversubscribed threads).
                - MPI continues to scale effectively by distributing processes across physical cores, avoiding the overhead of thread contention.

---

2. In-Depth Analysis of Pthreads Scaling

Optimum Performance with 8 Threads:

- When the thread count matches the system's physical core count (e.g., 8 threads for an 8-core processor), the

  best performance is achieved up to toss counts around 10^8.

  - **Reason**: The computational load is evenly distributed across the available CPU cores, ensuring maximum utilization without additional overhead.

Improved Performance with More Than 8 Threads:

At toss counts of **10^8 and above**, experiments using **32 or even 500 threads outperform the 8-thread configuration**. This behavior can be explained by the following factors:

1. **Hyper-Threading and Hardware Multi-Threading**:
   - Modern CPUs support **hardware-level multithreading** (e.g., hyper-threading), allowing each physical core to execute multiple threads simultaneously.
   - Even on an 8-core system, using more than 8 threads enables better utilization of the CPU's logical cores, improving parallel performance.
2. **Thread Pool and Task Scheduling**:
   - Operating systems efficiently schedule threads to balance CPU workload. Additional threads can fill idle time and ensure CPU cores are not left underutilized due to minor delays or load imbalances.
3. **Task Granularity and Load Balancing**:
   - In Monte Carlo simulations, tasks (dart tosses) are independent and easily parallelizable.
   - Increasing the number of threads allows the workload to be broken into smaller chunks, enabling better load balancing and ensuring no CPU core remains idle.
4. **Dynamic Workload Compensation**:
   - Differences in thread execution speeds due to small variations in computation or scheduling can be mitigated when more threads are used.
   - This helps maintain a consistent overall execution time, particularly for large workloads.
5. **Mutex Synchronization Impact**:
   - The global update of circle counts in Pthreads is protected by a **mutex**. However, in Monte Carlo simulations, the frequency of mutex contention is low because most of the computation occurs independently before aggregation.
   - This minimizes the overhead of synchronization, even when more threads are used.

---

3. In-Depth Analysis of MPI Scaling

Performance at Low Toss Counts (10^6):

- 4 Processes

  show the slowest performance (

  0.561 s

) compared to higher process counts like

  128 processes

  (

  0.078 s

).

  - **Reason**: At low computational loads, the **MPI communication overhead** (e.g., operations like `MPI_Bcast` and `MPI_Reduce`) dominates execution time. With fewer processes, the benefits of parallelism cannot overcome this overhead.

Performance at Medium Toss Counts (10^7 to 10^8):

- **8 to 16 Processes** demonstrate the **best performance**:
  - Execution times stabilize around **0.08 s**.
  - **Reason**: The computational workload increases, reducing the relative impact of communication overhead.
  - At this stage, adding more processes does not significantly improve performance as the overhead is already minimal.
- **32 and 64 Processes** show slightly higher times (**0.08–0.081 s**), indicating that scaling efficiency begins to plateau.

Performance at High Toss Counts (10^9):

- **4 Processes**: Execution time increases dramatically (**12.25 s**).
  - **Reason**: The limited number of processes leads to a significant imbalance in the computational load, with each process handling a larger share of the workload.
- **8, 16, and 32 Processes**: Performance remains **optimal and consistent** (~**0.077–0.079 s**).
  - **Reason**: At high workloads, the computational advantages of parallelism fully offset communication overhead, achieving near-ideal load distribution.
- **128 Processes**: Performance is comparable (**0.079 s**), but no substantial gains are observed beyond **32 processes**.
  - **Reason**: While the workload is highly parallelizable, communication overhead and diminishing returns in load distribution prevent further speedups.

Key Observations:

1. **MPI Scaling is Most Effective Between 8 and 32 Processes**:
    - Performance stabilizes, and additional processes yield negligible improvements due to communication bottlenecks.
2. **4 Processes are Insufficient for Large Workloads**:
    - Execution times increase significantly as the workload grows.
3. **At 128 Processes**:
    - Overhead associated with **process creation and communication** outweighs the benefits of further parallelization, leading to plateaued performance.

Conclusion:

MPI scaling is highly efficient for medium-to-large workloads, with **8 to 32 processes** delivering the best results. Beyond this range, diminishing returns and communication costs limit further performance gains.

Summary Table

Method	Low Toss Count (10^6)	Medium Toss Count (10^7)	High Toss Count (10^8 and Above)
Single Core	Fast	Moderate	Slowest
MPI	Slowest (Communication Overhead)	Matches Single-Core Performance	Fastest
Pthreads	Fastest	Fastest (8 threads optimal, even others)	Faster with > 8 threads, but fastest implementation is `65536` thread implementation.