Addis Ababa University

Addis Ababa Institute of Technology

School of Information Technology and Engineering

Course Title:Distributed Computing Concepts for AI ; Nature

Inspired

Parallel Programming: Assignment 1

By:

Mahlet Nigussie – GSR/3758/16

Submitted To:

Beakal Gizachew (PhD)

Introduction

This report presents the solutions to the parallel programming problems outlined in the assignment. The focus of the report is on performance analysis and optimization of parallel computing techniques using PThreads and OpenMP. The tasks involve evaluating the speedup and efficiency of parallelized algorithms, comparing serial and parallel execution times, and understanding the impact of parallelization overhead.

The experiments were conducted on a desktop system running MSYS on Windows, utilizing a multi-core CPU. The software environment included the GCC compiler with support for OpenMP and PThreads. The report will cover the results of parallel implementations for problems related to parallel sum, performance benchmarking, and estimation, highlighting the benefits and challenges associated with parallel programming. The analysis includes discussions of hardware and software limitations, as well as potential strategies for improving parallel efficiency and scalability.

1 Problem 1: Performance Analysis of Parallelization

In this report, we analyze the performance of a parallelized program compared to its serial counterpart, focusing on *speedup* and *efficiency* under different conditions. The two primary objectives of this analysis are:

- Investigating the behavior of speedup and efficiency as the number of processors increases, with the problem size fixed.
- Analyzing the effects of increasing the problem size while keeping the number of processors fixed, considering the impact of overhead.

Problem Description

We are given the following formulas:

- **Serial Time**: $T_{\text{serial}} = n^2$, where n is the problem size.
- **Parallel Time**:
  - Part (a): $T_{\text{parallel}} = \frac{n^2}{p} + \log_2(p)$, where p is the number of processors.
  - Part (b): $T_{\text{parallel}} = \frac{n^2}{p} + T_{\text{overhead}}$, where T_{overhead} depends on the growth rate of the overhead.

Speedup is defined as the ratio of serial time to parallel time:

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

Efficiency is defined as the speedup divided by the number of processors:

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

We explore the following cases:

- Part (a): Calculate speedup and efficiency for various values of n and p.
- Part (b): Investigate the impact of overhead growth on speedup and efficiency under two scenarios:
  - Slow Overhead Growth: $T_{\text{overhead}} = \log_2(n)$
  - Fast Overhead Growth: $T_{\text{overhead}} = n$

Implementation

1. **Part (a)**: We calculate the serial and parallel times for different values of n (problem sizes: 10, 20, 40, 80, 160, 320) and p (processors: 1, 2, 4, 8, 16, 32, 64, 128). Speedup and efficiency are computed using the formulas defined above, and the results are written to a CSV file for further analysis.
2. **Part (b)**: We analyze two cases of overhead growth: slow and fast.
   - For slow overhead growth, we use $T_{\text{overhead}} = \log_2(n)$.
   - For fast overhead growth, we use $T_{\text{overhead}} = n$.

   We calculate speedup and efficiency for different values of n and p under both overhead scenarios.

Results and Discussion

1.1 Part (a) - Speedup and Efficiency

We observe the following:

- **Speedup**: As the number of processors increases, speedup increases, but it becomes less significant as p grows. This is due to the logarithmic term in the parallel time formula $\log_2(p)$, which represents the diminishing returns as more processors are added.
- **Efficiency**: As the number of processors increases, efficiency decreases. This is expected due to the increasing overhead, which results in less work being done per processor.

1.2 Part (b) - Slow Overhead Growth

With slow overhead growth (i.e., $T_{\text{overhead}} = \log_2(n)$), we observe that as the problem size n increases, the speedup improves. This indicates that the slow increase in overhead has a lesser impact on the performance as the problem size grows.

1.3 Part (b) - Fast Overhead Growth

With fast overhead growth (i.e., $T_{\text{overhead}} = n$), we observe that as n increases, the speedup decreases. This suggests that when overhead grows faster than the serial time, the parallel efficiency decreases as the problem size increases.

Plots

The following plots were generated to illustrate the results:

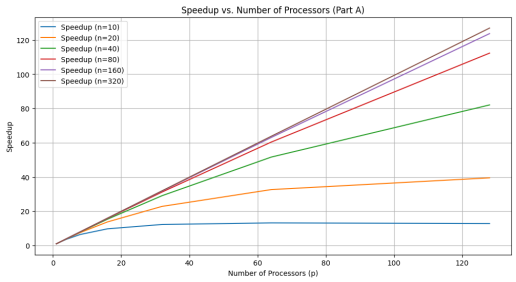

Figure 1: Speedup vs. Number of Processors (Part a)

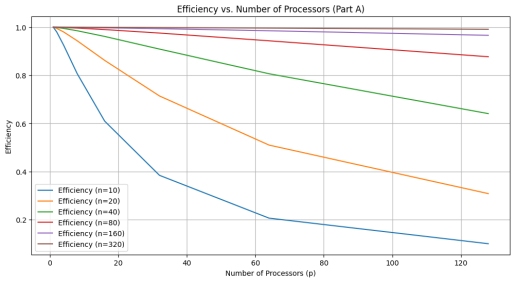

Figure 2: Efficiency vs. Number of Processors (Part a)

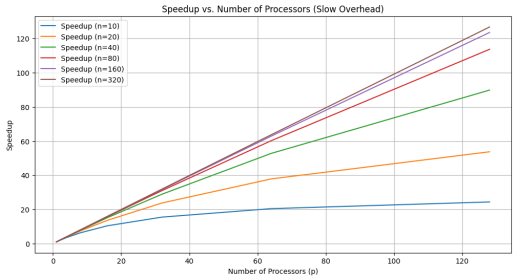

Figure 3: Speedup vs. Number of Processors (Slow Overhead Growth)

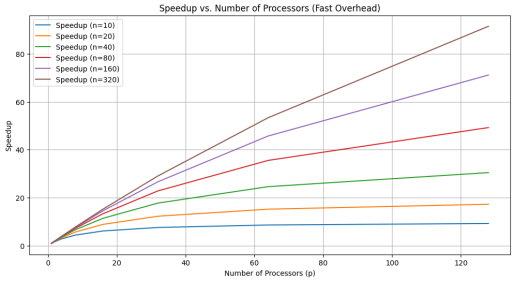

Figure 4: Speedup vs. Number of Processors (Fast Overhead Growth)

The results confirm the importance of managing overhead in parallel computing. While adding more processors can initially lead to significant speedup, diminishing returns are observed as overhead grows. Additionally, slow overhead growth (Figure 3) can improve parallel efficiency as the problem size increases, while fast overhead growth (Figure 4) has the opposite effect, decreasing efficiency. The speedup and efficiency trends for processors (Figures 1 and 2) further reinforce these observations.

2 Problem 2 - Parallel Performance Model

Execution Time of Parallel Versions

We need to express the execution time for different parallel versions using the following variables:

- S: Sequential execution time,
- T: Number of threads,

- O: Overhead (fixed for all versions),
- B: Barrier cost,
- M: Mutex cost,
- E_0: Extra work for thread 0 in Version 5.

Version 2: Parallel Summing with Overhead and Barrier

The execution time for Version 2 is given by:

$$T_2 = \frac{S}{T} + O + B$$

Where:

- $\frac{S}{T}$: Time each thread spends processing a portion of the work,
- O: Overhead for parallelization,
- B: Barrier cost.

Version 3: Parallel Summing with Overhead, Barrier, and Mutexes

The execution time for Version 3 is given by:

$$T_3 = \frac{S}{T} + O + B + M \times (T - 1)$$

Where:

- $\frac{S}{T}$: Time each thread spends processing a portion of the work,
- O: Overhead for parallelization,
- B: Barrier cost,
- $M \times (T - 1)$: Mutex cost for $T - 1$ threads (since thread 0 does not use a mutex).

Version 5: Parallel Summing with Overhead, Barrier, Mutexes, and Extra Work for Thread 0

The execution time for Version 5 is given by:

$$T_5 = \frac{S}{T} + O + B + M \times (T - 1) + E_0$$

Where:

- $\frac{S}{T}$: Time each thread spends processing a portion of the work,
- O: Overhead for parallelization,
- B: Barrier cost,
- $M \times (T - 1)$: Mutex cost for $T - 1$ threads,
- E_0: Extra work required by thread 0.

Conditions for Parallelization to be Profitable for Version 3

For parallelization to be profitable, the parallel execution time must be smaller than the sequential execution time, i.e.:

$$T_3 < S$$

Substituting the formula for T_3:

$$\frac{S}{T} + O + B + M \times (T-1) < S$$

Multiplying both sides by T to eliminate the fraction:

$$S + T \times O + T \times B + M \times T \times (T-1) < S \times T$$

Subtracting S from both sides:

$$T \times O + T \times B + M \times T \times (T-1) < S \times (T-1)$$

Dividing both sides by $T-1$ (assuming $T > 1$):

$$\frac{T \times O + T \times B + M \times T \times (T-1)}{T-1} < S$$

Thus, the condition for parallelization to be profitable is:

$$\frac{T \times O + T \times B + M \times T \times (T-1)}{T-1} < S$$

This inequality shows when parallel execution time becomes less than the sequential time, thus making parallelization beneficial.

3 Problem 3: STREAM Benchmark Performance with OpenMP

The STREAM benchmark was used to evaluate the performance of four computational kernels—`Copy`, `Scale`, `Add`, and `Triad`—on a system with varying numbers of OpenMP threads. The goal was to measure the impact of increasing the number of threads (1, 2, 4, 8, and 16) on the performance of these kernels while maintaining the same array size of 10,000,000 elements. The results were then analyzed to determine how the performance of each kernel scales with the number of threads.

The benchmark was then run with varying numbers of OpenMP threads using the `OMP_NUM_THREADS` environment variable. The performance results for `Copy`, `Scale`, `Add`, and `Triad` were recorded for each configuration, with threads set to 1, 2, 4, 8, and 16.

Results

The table below shows the best performance (in MB/s) for each of the four kernels for different numbers of OpenMP threads:

Threads (MB/s)	Copy	Scale	Add	Triad
1 Thread	6313.4	7242.8	8247.4	8012.6
2 Threads	9248.0	9137.0	10189.3	10323.9
4 Threads	9697.5	9426.7	10794.8	10690.4
8 Threads	9506.2	9434.5	10815.3	10741.1
16 Threads	9506.7	9250.8	10547.6	10637.5

Table 2: Performance of Functions with Different Numbers of Threads

Observations

The following observations can be made from the results:

- As the number of threads increases from 1 to 16, the performance for each kernel improves, showing the benefit of parallelization.
- For the `Copy` and `Scale` kernels, performance begins to level off after 4 threads. Beyond this point, increasing the number of threads (e.g., from 8 to 16) does not result in significant performance gains.
- The `Add` and `Triad` kernels show better scalability, with performance continuing to improve with up to 8 threads.
- The diminishing returns observed at higher thread counts suggest that factors such as memory bandwidth limitations, synchronization overhead, and cache contention become limiting as the number of threads increases.
- The variability between the minimum and maximum times for each test reflects the system's load balancing and potential cache contention, especially as the number of threads increases.

Analysis

The observed diminishing returns in performance with increasing threads can be attributed to several factors:

- Memory Bandwidth Limitation: With increasing threads, the available memory bandwidth is shared, leading to a bottleneck once the system reaches a certain threshold. This saturation reduces the performance improvements for additional threads.
- Overheads: As more threads are used, there is an increase in overhead due to thread management, synchronization, and context switching. These overheads reduce the efficiency of parallelization as the number of threads increases.
- Cache Contention: As the number of threads increases, so does the potential for cache contention. Multiple threads may try to access the same cache lines, reducing cache efficiency and leading to slower execution.

The STREAM benchmark results demonstrate that increasing the number of OpenMP threads generally improves performance, with diminishing returns as the number of threads increases. For the `Copy` and `Scale` kernels, the optimal performance is achieved with 4 threads, while the `Add` and `Triad` kernels benefit from up to 8 threads. However, after a certain point, further increases in threads do not yield significant performance gains due to memory bandwidth limitations and other system overheads.

4 Problem 4: Parallelization of π Calculation using Pthreads and OpenMP

This report presents the results of parallelizing the calculation of π using both **Pthreads** and **OpenMP**. The task is to approximate the value of π by utilizing parallel computing techniques and analyze the performance in terms of **speedup** and **efficiency**.

The serial implementation of the π calculation (`pi-serial.c`) is provided for validation. The parallel implementation is done using **Pthreads** and **OpenMP**, and the performance is evaluated by varying the number of threads from 1 to 16. The goal is to measure the **speedup** and **efficiency** of the parallelized version of the code and compare it with the sequential version.

Implementation Details

The approximation of π is performed using the following series:

$$\pi \approx 4 \times \sum_{i=0}^{n-1} \frac{1}{1 + (2i+1)^2}$$

Where n is the number of iterations, provided as input by the user. The program computes π using the **serial**, **Pthreads**, and **OpenMP** implementations.

- The sequential and parallel programs calculate π using a numerical approximation based on the following formula:
$$\pi = 4 \sum_{i=0}^{n} \frac{(-1)^i}{2i+1}$$
- The sequential implementation computes π in a single thread, while the parallel implementations utilize PThreads and OpenMP for parallelization.
- The number of iterations for the approximation is fixed at 5,000,000,000 to ensure sufficient accuracy.
- The number of threads used in the parallel versions varies from 1 to 16 to analyze the effect of parallelism on performance.
- Execution time, speedup, and efficiency are measured and recorded for both the sequential and parallel implementations.

- Speedup is calculated as:

$$\text{Speedup} = \frac{\text{Time(seq)}}{\text{Time(parallel)}}$$

- Efficiency is calculated as:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

- Parallelization Granularity The granularity of the parallelism is adjusted by increasing the number of iterations to ensure that the program runs for at least 5 seconds, providing reliable performance measurements.

The sequential and parallel programs computed π using a numerical approximation with the number of iterations fixed at 5,000,000,000. The number of threads used in parallel computations ranged from 1 to 16 to evaluate the performance improvements achieved through parallelism. Execution times for both the sequential and parallel versions were measured using the 'gettimeofday()' function. The programs were executed for a minimum of 5 seconds for reliable performance measurements.

4.1 Results

Performance Metrics

The performance of the serial, Pthreads, and OpenMP implementations is evaluated for varying thread counts (1, 2, 4, 8, 16). The execution times for the sequential and parallel versions of the program, as well as the corresponding speedup and efficiency, are shown in the following plots. The speedup was calculated as the ratio of sequential time to parallel time, while efficiency was computed by dividing the speedup by the number of threads used. The following results were obtained.

Metric	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
Pthreads Speedup	0.997344	1.980549	3.094126	3.928834	3.713049
OpenMP Speedup	0.889346	1.757215	2.922038	3.539373	3.542681
Pthreads Efficiency	0.997344	0.990275	0.773532	0.491104	0.232066
OpenMP Efficiency	0.889346	0.878608	0.730509	0.442422	0.221418

Table 4: Pthreads and OpenMP Speedup and Efficiency with Different Number of Threads

4.2 Analysis

- **Pthreads Speedup**: As expected, speedup increases with the number of threads. However, after 8 threads, the speedup starts to plateau due to overhead from synchronization and thread management.

- **Pthreads Efficiency**: Efficiency decreases with an increasing number of threads. With 16 threads, the efficiency drops significantly, showing that the overhead of managing multiple threads starts to outweigh the benefits of parallelism.
- **OpenMP Speedup**: OpenMP shows similar trends in speedup as Pthreads. However, OpenMP tends to outperform Pthreads slightly in terms of speedup for the same number of threads.
- **OpenMP Efficiency**: OpenMP also exhibits a decrease in efficiency as the number of threads increases, with a sharper decline at 16 threads.

Lines of Code for Serial and Parallel Implementations

- Total Lines of Code of (`pi-serial.c`): **85 lines**
- Total Lines of Code of (`pthreads_pi.c`): **98 lines**
- Total Lines of Code of (`pi-openmp.c`): **74 lines**

These lines of code include both the implementation of the computation logic and the timing functions used to measure the execution time of each parallel and serial version.

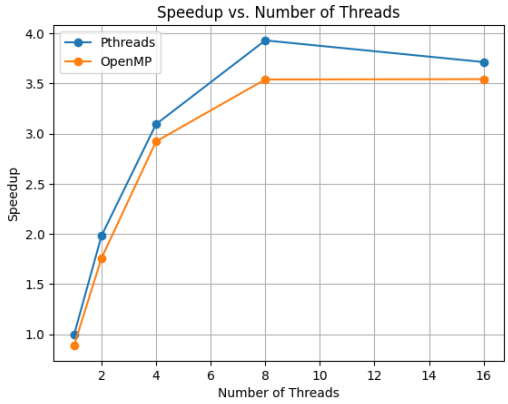

Figure 5: Speedup as a Function of Number of Threads

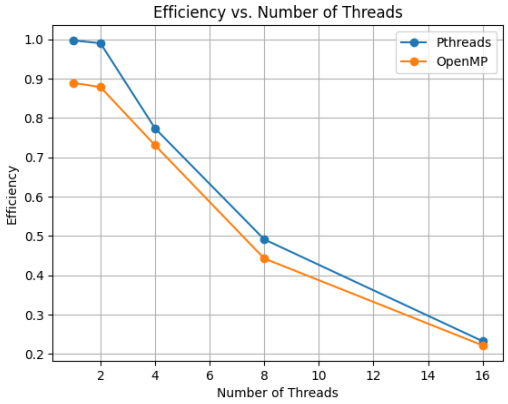

Figure 6: Efficiency as a Function of Number of Threads

The plot (Figure 5) shows the speedup of the parallel implementations as the number of threads increases from 1 to 16. As expected, the speedup increases with the number of threads up to a point, after which it begins to plateau due to the overhead introduced by thread management and synchronization.

The Plot (Figure 6) illustrates the **efficiency** as a function of the number of threads for both Pthreads and OpenMP implementations. The efficiency of the parallel implementation is shown as a function of the number of threads. Efficiency decreases as the number of threads increases beyond a certain point, reflecting the diminishing returns of adding more threads due to factors like thread contention and synchronization overhead.

- **Speedup**: The speedup achieved using both Pthreads and OpenMP improves with an increasing number of threads but shows diminishing returns for larger thread counts. The speedup starts to level off at 8 threads, indicating that there is a point beyond which adding more threads does not significantly reduce computation time.
- **Efficiency**: The efficiency of both implementations decreases as the number of threads increases. This is expected due to the overhead associated with thread synchronization and communication.
- **Pthreads vs. OpenMP**: OpenMP slightly outperforms Pthreads in terms of speedup and efficiency. However, both approaches have similar performance trends and exhibit diminishing returns with higher thread counts.

In this experiment, the parallel implementations using PThreads and OpenMP demonstrated significant speedup for a range of thread counts. However, the speedup achieved diminishes as the number of threads increases beyond a certain point, which is reflected in the decreasing efficiency. This behavior can be attributed to several factors, including the overhead from thread management and synchronization, as well as hardware limitations such as the number of available processor cores. The results highlight the importance of optimizing the parallelization strategy, especially for large-scale computations like this one. The system used in this experiment, a desktop running MSYS on Windows with a multi-core CPU, and the limitations of the GCC compiler with OpenMP and PThreads support, may have constrained the scalability of the parallel implementation. By adjusting the granularity of the parallel tasks and optimizing the parallelization approach to better match the available hardware, further improvements in speedup and efficiency could be achieved.