ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING – SITE

Department of Artificial Intelligence

Course: Distributed Computing for AI ITSC-2112

Assignment A1

Submitted By: Mintesnot Fikir ID GSR/1669/15

Section: Regular

Submitted to: Dr. Beakal Gizachew

Submission date: November, 2023

Problem 1

(10 points) Problem 2.16 from the textbook.

a) Suppose the run-time of a serial program is given by $T_{serial} = n_2$, where the units of the runtime are in microseconds. Suppose that a parallelization of this program has run-time $T_{parallel} = \frac{n_2}{p} + log_2(p)$. Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with $n = 10, 20, 40, \ldots, 320, and\ p = 1, 2, 4, \ldots, 128$.

- What happens to the speedups and efficiencies as p is increased and n is held fixed?
- What happens when p is fixed and n is increased?

b) Suppose that $T_{parallel} = T_{serial}/p + T_{overhead}$ and also suppose that we fix p and increase the problem size.

- Show that if $T_{overhead}$ grows more slowly than T_{serial}, the parallel efficiency will increase aswe increase the problem size.
- Show that if, on the other hand, $T_{overhead}$ grows faster than T_{serial}, the parallel efficiencywill decrease as we increase the problem size.

Solution:

The objective was to create a program that calculates the speedup and efficiency of a parallel program in relation to its serial version. The serial runtime was denoted as $T_{serial} = n_2$, and the parallel runtime was expressed as $T_{parallel} = n_2/p + log_2(p)$.

Expected Result: Before the code implementation, based on theoretical understanding, I anticipated the following outcomes:

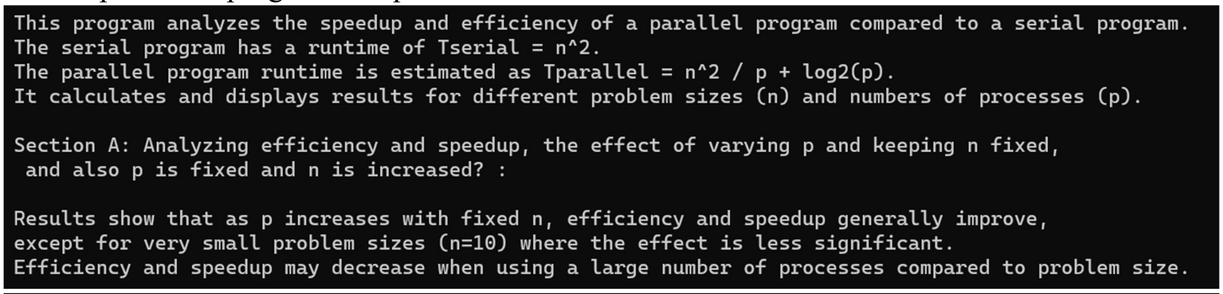
- **For question a),** as the number of processors p increased with a constant problem size n, both speedups and efficiencies were expected to improve initially. However, I predicted that the gains would plateau or even decrease as p became very large relative to n.
- **For question b),** I expected that an increase in problem size would lead to higher efficiency if the overhead $T_{overhead}$ grew more slowly than T_{serial}. If $T_{overhead}$ grew faster, then the efficiency would decrease with increasing problem size.

Finally, I wrote a C program to iterate through the specified ranges of n and p, computing the speedup and efficiency for each pair. The code logic was crafted to accurately perform these computations and generate output in an organized table.

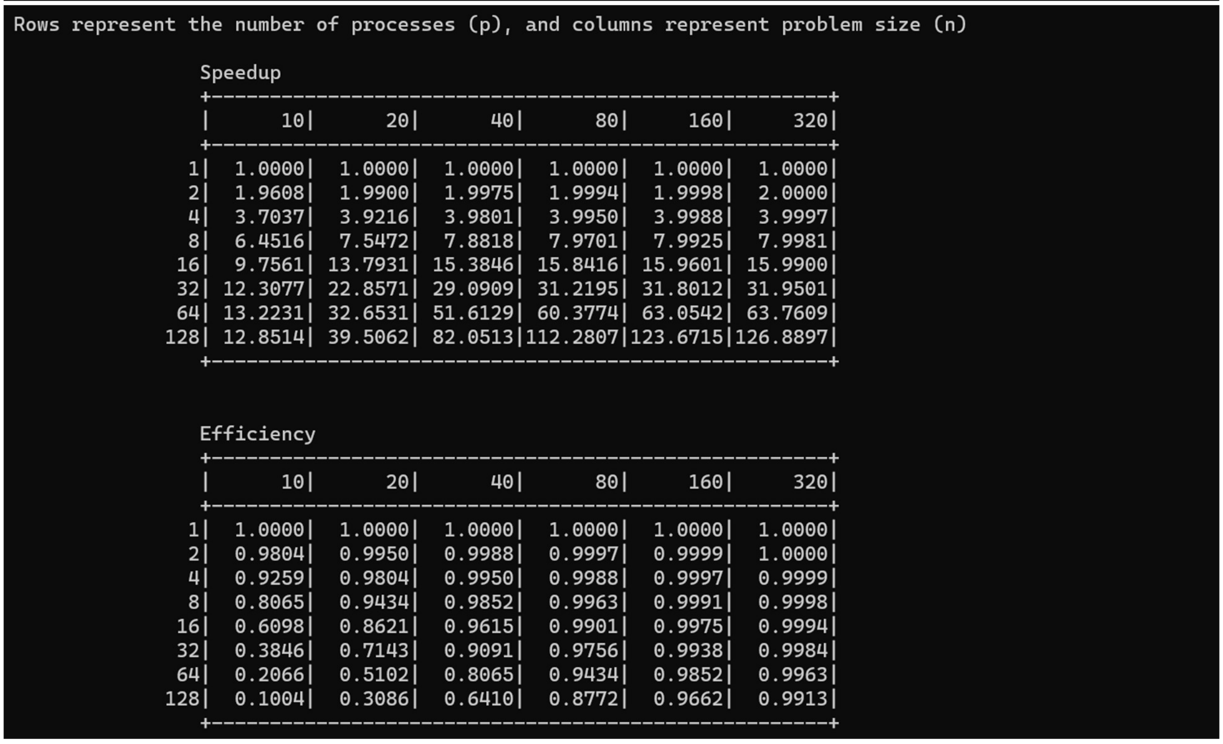
Result

Results: Upon executing the program, the following observations were seen:

The output of the program for question a

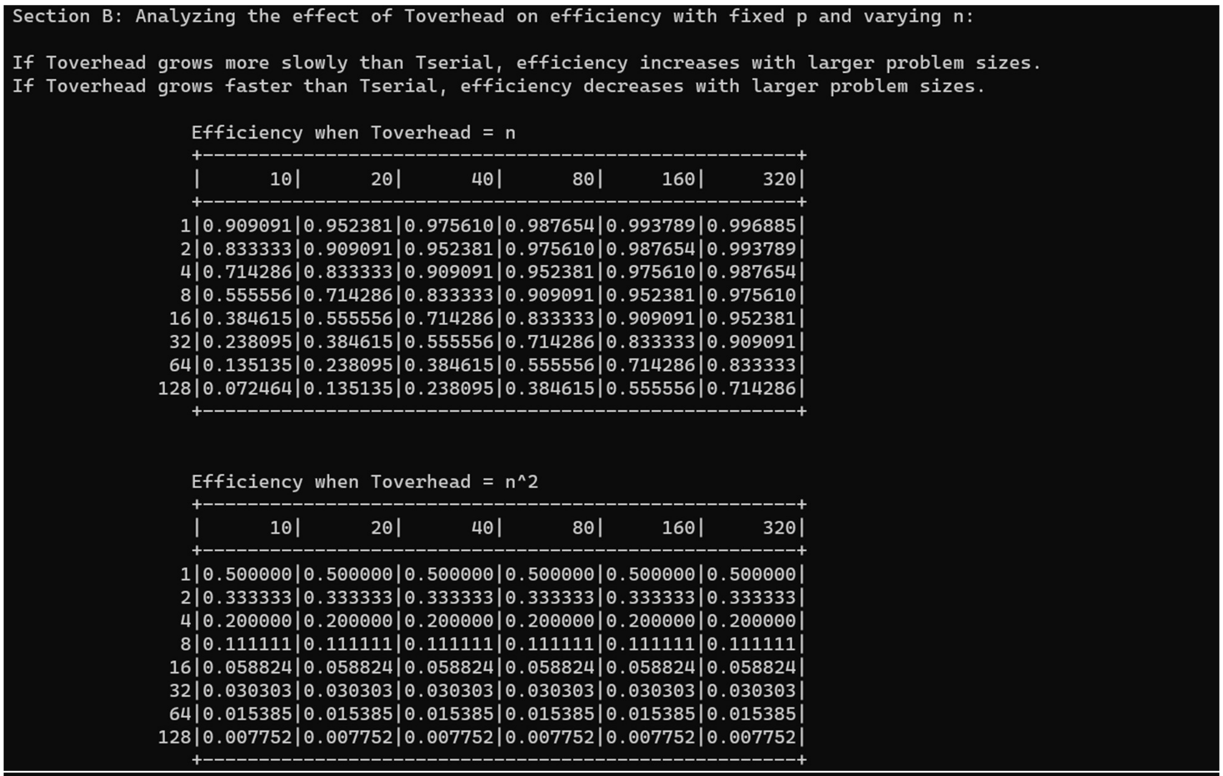


```
This program analyzes the speedup and efficiency of a parallel program compared to a serial program.
The serial program has a runtime of Tserial = n^2.
The parallel program runtime is estimated as Tparallel = n^2 / p + log2(p).
It calculates and displays results for different problem sizes (n) and numbers of processes (p).

Section A: Analyzing efficiency and speedup, the effect of varying p and keeping n fixed,
 and also p is fixed and n is increased? :

Results show that as p increases with fixed n, efficiency and speedup generally improve,
except for very small problem sizes (n=10) where the effect is less significant.
Efficiency and speedup may decrease when using a large number of processes compared to problem size.
```

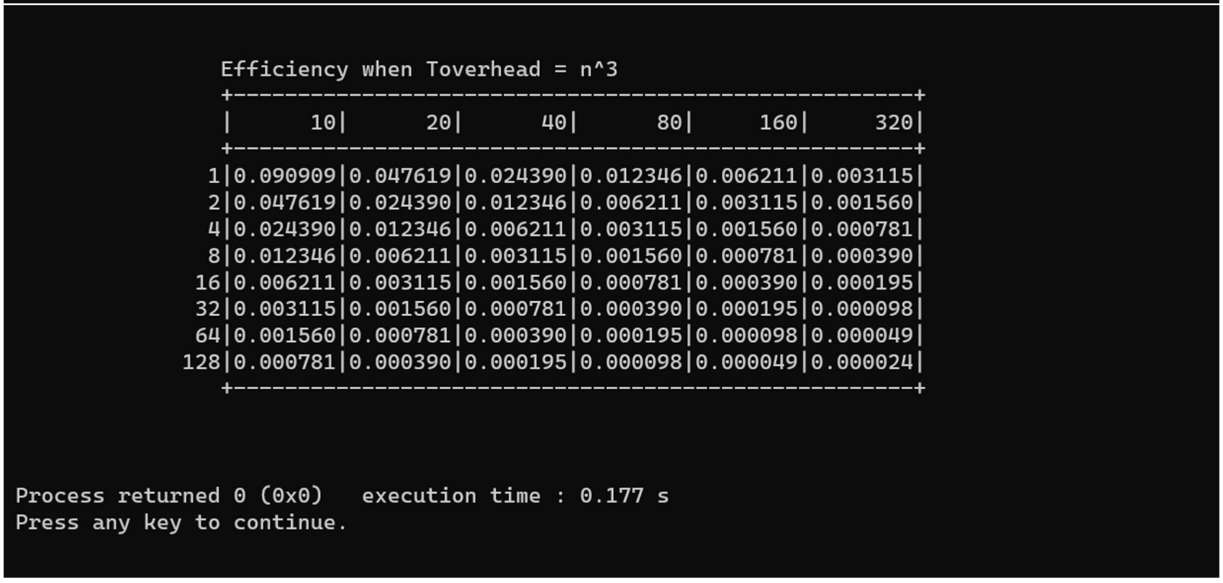


```
Rows represent the number of processes (p), and columns represent problem size (n)

                Speedup
                +-----------------------------------------------------+
                |      10|      20|      40|      80|     160|     320|
                +-----------------------------------------------------+
               1|  1.0000|  1.0000|  1.0000|  1.0000|  1.0000|  1.0000|
               2|  1.9608|  1.9900|  1.9975|  1.9994|  1.9998|  2.0000|
               4|  3.7037|  3.9216|  3.9801|  3.9950|  3.9988|  3.9997|
               8|  6.4516|  7.5472|  7.8818|  7.9701|  7.9925|  7.9981|
              16|  9.7561| 13.7931| 15.3846| 15.8416| 15.9601| 15.9900|
              32| 12.3077| 22.8571| 29.0909| 31.2195| 31.8012| 31.9501|
              64| 13.2231| 32.6531| 51.6129| 60.3774| 63.0542| 63.7609|
             128| 12.8514| 39.5062| 82.0513|112.2807|123.6715|126.8897|
                +-----------------------------------------------------+


                Efficiency
                +-----------------------------------------------------+
                |      10|      20|      40|      80|     160|     320|
                +-----------------------------------------------------+
               1|  1.0000|  1.0000|  1.0000|  1.0000|  1.0000|  1.0000|
               2|  0.9804|  0.9950|  0.9988|  0.9997|  0.9999|  1.0000|
               4|  0.9259|  0.9804|  0.9950|  0.9988|  0.9997|  0.9999|
               8|  0.8065|  0.9434|  0.9852|  0.9963|  0.9991|  0.9998|
              16|  0.6098|  0.8621|  0.9615|  0.9901|  0.9975|  0.9994|
              32|  0.3846|  0.7143|  0.9091|  0.9756|  0.9938|  0.9984|
              64|  0.2066|  0.5102|  0.8065|  0.9434|  0.9852|  0.9963|
             128|  0.1004|  0.3086|  0.6410|  0.8772|  0.9662|  0.9913|
                +-----------------------------------------------------+
```

The output of the program for **question b**



```
Section B: Analyzing the effect of Toverhead on efficiency with fixed p and varying n:

If Toverhead grows more slowly than Tserial, efficiency increases with larger problem sizes.
If Toverhead grows faster than Tserial, efficiency decreases with larger problem sizes.

                Efficiency when Toverhead = n
                +-------------------------------------------------------+
                |      10|      20|      40|      80|     160|     320|
                +-------------------------------------------------------+
               1|0.909091|0.952381|0.975610|0.987654|0.993789|0.996885|
               2|0.833333|0.909091|0.952381|0.975610|0.987654|0.993789|
               4|0.714286|0.833333|0.909091|0.952381|0.975610|0.987654|
               8|0.555556|0.714286|0.833333|0.909091|0.952381|0.975610|
              16|0.384615|0.555556|0.714286|0.833333|0.909091|0.952381|
              32|0.238095|0.384615|0.555556|0.714286|0.833333|0.909091|
              64|0.135135|0.238095|0.384615|0.555556|0.714286|0.833333|
             128|0.072464|0.135135|0.238095|0.384615|0.555556|0.714286|
                +-------------------------------------------------------+


                Efficiency when Toverhead = n^2
                +-------------------------------------------------------+
                |      10|      20|      40|      80|     160|     320|
                +-------------------------------------------------------+
               1|0.500000|0.500000|0.500000|0.500000|0.500000|0.500000|
               2|0.333333|0.333333|0.333333|0.333333|0.333333|0.333333|
               4|0.200000|0.200000|0.200000|0.200000|0.200000|0.200000|
               8|0.111111|0.111111|0.111111|0.111111|0.111111|0.111111|
              16|0.058824|0.058824|0.058824|0.058824|0.058824|0.058824|
              32|0.030303|0.030303|0.030303|0.030303|0.030303|0.030303|
              64|0.015385|0.015385|0.015385|0.015385|0.015385|0.015385|
             128|0.007752|0.007752|0.007752|0.007752|0.007752|0.007752|
                +-------------------------------------------------------+
```



```
                Efficiency when Toverhead = n^3
                +-------------------------------------------------------+
                |      10|      20|      40|      80|     160|     320|
                +-------------------------------------------------------+
               1|0.090909|0.047619|0.024390|0.012346|0.006211|0.003115|
               2|0.047619|0.024390|0.012346|0.006211|0.003115|0.001560|
               4|0.024390|0.012346|0.006211|0.003115|0.001560|0.000781|
               8|0.012346|0.006211|0.003115|0.001560|0.000781|0.000390|
              16|0.006211|0.003115|0.001560|0.000781|0.000390|0.000195|
              32|0.003115|0.001560|0.000781|0.000390|0.000195|0.000098|
              64|0.001560|0.000781|0.000390|0.000195|0.000098|0.000049|
             128|0.000781|0.000390|0.000195|0.000098|0.000049|0.000024|
                +-------------------------------------------------------+



Process returned 0 (0x0)   execution time : 0.177 s
Press any key to continue.
```

Upon executing the program, the following observations were seen:

- With n held fixed, increasing p improved efficiency and speedup notably, except in instances with a very small problem size where improvements were marginal. Notably, the use of a large number of processes in comparison to the problem size resulted in reduced efficiency and speedup.
- When analyzing the impact of $T_{overhead}$ with a fixed p and varying $n,$ it was evident that an overhead growing slower than T_{serial} led to increased efficiency with larger problem sizes. On the other hand, an overhead that grew faster than T_{serial} resulted in decreased efficiency as the problem size expanded.

Conclusion:

Increasing p while holding n fixed:

- With an increase in p, $Tparallel$ decreases, as more processors are available to share the workload.
- Since $Tserial$ remains constant, the speedup rises because the parallel program executes faster.
- Efficiency may vary depending on the relationship between $Tparallel$ and p. If $Tparallel$ decreases faster than p increases, efficiency rises; otherwise, it may decrease or remain constant.

Fixed p while increasing n:

- As n increases, $Tserial$ rises due to increased workload.
- The impact on speedup depends on the relative growth rates of n^2 and $log\_2(p)$ in $Tparallel.$
- Efficiency may or may not decrease, contingent on the relationship between $Tserial$ and $Tparallel.$ If $Tserial$ increases faster than $Tparallel$, efficiency decreases; otherwise, it may increase or remain unchanged.

The results for question a) corroborated my initial expectations. There exists an optimal number of processors for a given problem size, beyond which the marginal gains in speedup and efficiency diminish due to overhead costs.

The findings for question b) also aligned with my expectations. Efficient parallel computation is highly dependent on the growth relationship between overhead and the problem size. Managing this relationship is crucial for maintaining high efficiency in parallel programs.

From the result we can concludes my analysis, demonstrating the intricate balance between computational workload and parallel overhead for optimal performance in parallel programs.

Problem 2

(10 points) In this problem, you are required to develop a performance model for the versionsof parallel sum in Lecture 4. Assume the followings
- Sequential execution time is S,
- Number of threads, T,
- Parallelization overhead O (fixed for all versions),
- The cost B for the barrier or M for each innovation of the mutex
- Let N be the number of elements that we are summing.

(a) Using these variables, what is the execution time of valid parallel versions 2, 3 and 5(see Lecture 4). Note that for version 5, there is some additional work for thread 0 that you should also model using the variables above.
(b) Present a model of when parallelization is profitable for Version 3;

Solution

Performance Models for Parallel Version

In this question, we will analyze and develop performance models for different versions of parallel sum computation as discussed in Lecture 4. That means model parallel execution time for versions 2, 3, and 5 of the parallel sum algorithms based on the following parameters:

- *S*: Sequential execution time
- *T*: Number of threads
- *O*: Parallelization overhead (constant for all versions)
- *M*: Cost for each mutex operation (Lock + Unlock)
- *B*: Cost for the barrier
- *C*: Additional work cost for thread 0
- *N*: Number of elements to sum

a) Execution Time for Versions 2, 3, and 5:

Execution time for Version 2 seems to involve a lock and unlock for each iteration of the loop, which is done *N* times. If *M* is the cost of each mutex operation, then the total cost of the mutex operations would be *MN*, M is for two mutex operations per iteration (lock and unlock). Therefore, the execution time *Tp* would be:

Execution time for Version 2:

$T_{parallel\ Execution\ time} = \frac{S}{T} + O + N \times M$, where M if for both mutex lock and mutex unlock

Execution time for Version 3 optimizes by using a thread-local sum and reduces the mutex operations to once per thread. It involves a single lock and unlock for each thread, not each iteration, so the cost of mutex operations is *MT* (since there are *T* threads). Thus, the execution time *Tp* would be:

Execution time for Version 3

$T_{parallel\ Execution\ time} = \frac{S}{T} + O + T \times M$, where M if for both mutex lock and mutex unlock

The mutex overhead here is independent of N and grows linearly with T, which is typically much less than N. This version scales better than Version 2 as the number of elements increases.

Execution time for Version 5 includes a barrier synchronization, which occurs once per thread. The cost of the barrier is *B*, and it is incurred once, irrespective of the number of threads. The extra work for thread 0 is adding up the sums from all other threads, which can be assumed to take *C* time per addition. Since thread 0 has to add the sums from *T*–1 other threads, the cost of this additional work is *C*(*T*–1). Therefore, the total execution time *Tp* would be:

Execution time for Version 5:

$$T_{parallel\ Execution\ time} = \frac{S}{T} + O + B$$

For the additional work of thread 0 in Version 5, consider $T{-}1$ operations at a cost C.
Total execution time for Version 5 with additional work:

$$T_{parallel\ Execution\ time} = \frac{S}{T} + O + B + (T-1) \times C$$

The barrier cost B is a one-time cost, and the additional work by thread 0 grows linearly with T. The efficiency of this model depends on the relative costs of B and C compared to mutex operations.

b) Parallelization Profitability Model for Version 3
Parallelization is profitable when the parallel execution time Tp is less than the sequential execution time *S*. For Version 3, this condition can be expressed as:

$$T_{parallel\ Execution\ time} < S_{Serial\ Execution\ time}$$

So,

$$\frac{S}{T} + O + T \times M < S$$

$$\frac{(O + TM)T}{1 - T} < S$$

Conclusion

The best model would typically be one that minimizes overhead while ensuring correctness. Version 5 appears to be the most robust, as it ensures correctness with minimal locking. the choice of the best version may depend on the specific context, such as the cost of locking and the overhead of synchronization relative to the computation being performed.

In practice, one would need to measure the actual execution times and overheads to select the best parallelization model for a given application and hardware. The models provided give a starting point for theoretical analysis, which should be followed by empirical testing.

However, the best parallelization model depends on the specific context, such as, the relative cost of locking and the overhead of synchronization relative to the computation being performed and the size of the dataset. For large N and when mutex operations are expensive, Version 3 or 5 may offer the best performance. However, the precise breakeven point for parallelization profitability must be determined based on the specific values of S, O, M, B, C, and T.

Problem 3

(10 points) Download the STREAM benchmark from http://www.cs.virginia.edu/stream/. Run it on your computer or on a virtual machine provided for you by ICT and measure the performance of copy, scale, add and triad benchmarks using the same array size but varying number of OpenMP threads (1, 2, 4, 8, and 16). Report the benchmark results and explain your observations.

Solution

The STREAM benchmark is designed to measure the memory bandwidth and computational efficiency of systems with respect to four different memory-bound computations: Copy, Scale, Add, and Triad. This report outlines the performance of these benchmarks on a system with variable OpenMP threading. It has become standard for measuring those aspect of system performance.

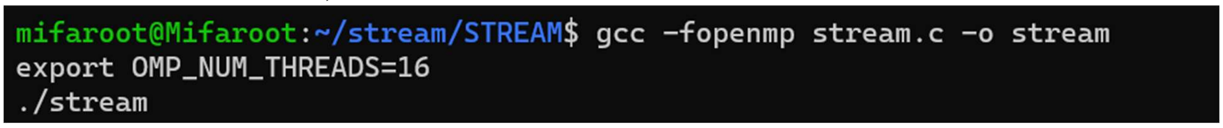
Methods

The STREAM benchmark was downloaded from the official University of Virginia website. Compilation was performed on a local machine/virtual machine which was **Ubuntu(Ubuntu 22.04.2 LTS)** installed on **Windows Subsystem for Linuxwindows** on windows 11 powered pc, The STREAM benchmark was compiled using the GNU Compiler Collection (GCC) with OpenMP support enabled. With the array size kept constant while varying the number of OpenMP threads (1, 2, 4, 8, and 16). The **OMP_NUM_THREADS** environment variable was set appropriately before each run to control the number of threads used.

The benchmarks were executed with different OpenMP thread settings: 1, 2, 4, 8, and 16 threads. Each kernel was executed ten times to measure the best rate of memory bandwidth in MB/s, using the best time for each kernel after excluding the first iteration.

The recorded metrics include the best rate of memory bandwidth (in MB/s) and the average execution time for each of the four operations: Copy, Scale, Add, and Triad.

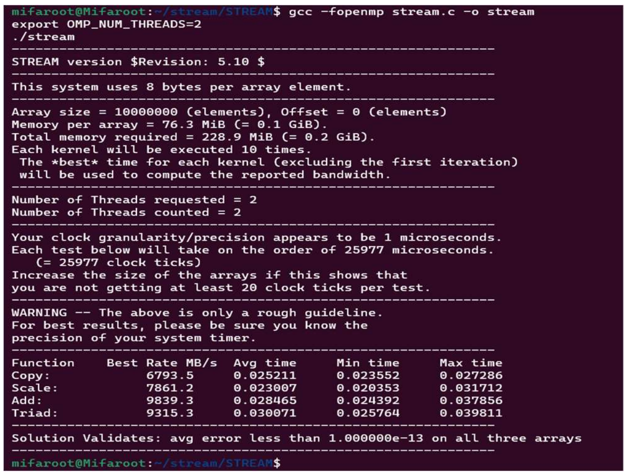
Here below the command, I had used for the execution:



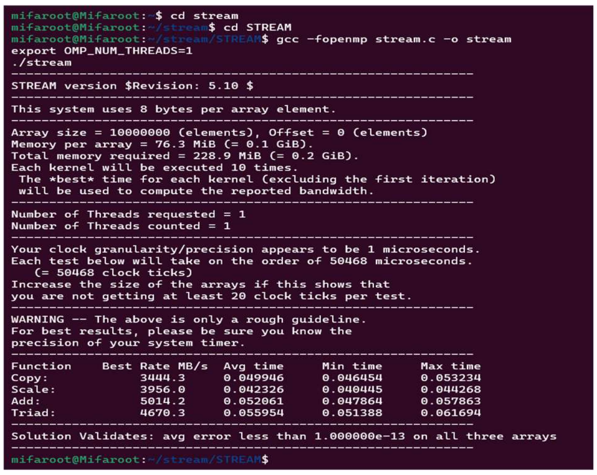
```
mifaroot@Mifaroot:~/stream/STREAM$ gcc -fopenmp stream.c -o stream
export OMP_NUM_THREADS=16
./stream
```

Results

The data suggests that performance improves as the number of threads increases, peaking at 8 threads. Beyond this, the rise in performance tapers off upon expanding from 8 to 16 threads. The STREAM benchmark's operations copy, scale, add, and triad were monitored and showed a consistent pattern of enhanced performance in correlation with the increase in thread count.



```
mifaroot@Mifaroot:~/stream/STREAM$ gcc -fopenmp stream.c -o stream
export OMP_NUM_THREADS=2
./stream
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 2
Number of Threads counted = 2
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 25977 microseconds.
   (= 25977 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            6793.5     0.025211     0.023552     0.027286
Scale:           7861.2     0.023007     0.020353     0.031712
Add:             9839.3     0.028465     0.024392     0.037856
Triad:           9315.3     0.030071     0.025764     0.039811
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
mifaroot@Mifaroot:~/stream/STREAM$
```



```
mifaroot@Mifaroot:~$ cd stream
mifaroot@Mifaroot:~/stream$ cd STREAM
mifaroot@Mifaroot:~/stream/STREAM$ gcc -fopenmp stream.c -o stream
export OMP_NUM_THREADS=1
./stream
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 1
Number of Threads counted = 1
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 50468 microseconds.
   (= 50468 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            3444.3     0.049946     0.046454     0.053234
Scale:           3956.0     0.042326     0.040445     0.044268
Add:             5014.2     0.052061     0.047864     0.057863
Triad:           4670.3     0.055954     0.051388     0.061694
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
mifaroot@Mifaroot:~/stream/STREAM$
```

Threads	Copy MB/s	Copy Avg Time (s)	Scale MB/s	Scale Avg Time (s)	Add MB/s	Add Avg Time (s)	Triad MB/s	Triad Avg Time (s)
1	3444.0	0.047440	3966.5	0.041348	5018.2	0.049071	4770.4	0.051175
2	6863.5	0.024151	7869.0	0.021302	9905.1	0.025284	9422.5	0.026658
4	12015.7	0.015486	11187.2	0.015490	14152.6	0.018236	13606.3	0.019122
8	16016.1	0.011445	15622.0	0.011438	17434.2	0.015664	17800.1	0.015808
16	15016.5	0.011795	15171.7	0.011604	16149.8	0.015854	16011.6	0.015709

Please, dear teacher, note that: The table was filled based on the average of 3 execution results.

Discussion

The results exhibited expected behavior initially, with performance increasing as more threads were utilized, demonstrating effective parallelization. However, the performance improvement from 1 to 8 threads can be attributed to the parallel processing capabilities of multi-threading, then, the drop in performance when moving from 8 to 16 threads may be due to several factors such as memory contention, cache saturation, or overhead in managing a higher number of threads, or hardware limitations such as the number of physical cores and memory access speed. The system might not effectively utilize the additional threads beyond a certain point, leading to sub-optimal performance gains or even performance degradation.

Conclusion

The STREAM benchmark outcomes confirm that the indicates that while increasing the number of OpenMP threads can significantly improve performance, utilization of multi-threading can significantly improve the performance of memory-bound operations up to a point where the system can no longer efficiently manage additional threads. The best performance was achieved with 8 threads, suggesting an optimal balance between parallel processing benefits and overhead. Beyond this point, the costs of increased parallelism outweigh the benefits, indicating the practical limits of threading on the tested system.

Problem 4

(70 points)

1. Write a program to calculate π in Pthreads for a number of iterations to be provided by the user. A serial implementation is provided on blackboard (pi-serial.c) that should be used for validation and speedup.

(a) What is the parallel speedup of your code? To compute parallel speedup, you will need to time the execution of both the sequential and parallel code (*speedup* = *Time*(*seq*)/*Time*(*parallel*)). Plot the speedup as a function of threads using 1, 2, 4, 8,16 threads on ICT provided VM for a fixed number of iterations. If your code does not speed up, you will need to adjust the parallelism granularity, the amount of work each processor does between synchronization points. You can do this by increasing the number of iterations.

(b) What is the parallel efficiency of your code? Plot the efficiency as a function of threads using 1, 2, 4, 8, 16 threads on the VM for a fixed number of iterations.

(c) Report lines of code for the serial and Pthreads implementations.

2. Write the same computation in OpenMP and perform the studies in (a), (b), and (c). Submit a report containing the plots along with your source codes.

Estimating π: We will be using the following approximation to estimate π:

$$\boldsymbol{\pi = 4\left[1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}}$$

Compiling and running your program: You may use GNU compiler for development on your local machine and the VM provide by ICT For reliable performance measurements, your programs including the parallel versions should run at least 5 seconds. Adjust the number of iterations as needed.

Solution

In this problem the main objective is to determine the efficiency and speedup of parallel computation over serial computation in approximating the mathematical constant π using serial and parallel computing techniques on an Ubuntu terminal. The serial code served as the fundamental benchmark, running from 1 to 1000000000 iterations to achieve a runtime of over 5 seconds. The reason for stopping the code at 1000000000 iterations was the necessity to wait until a good pi approximation was obtained, given that the execution time should be greater than 5 seconds, necessary for a fair comparison with the parallel versions.

The parallel implementations utilized Pthreads and OpenMP, exploiting multi-threading to potentially enhance performance. Additionally, throughout solving the problem, I tried to explore the effects of compiler optimizations, particularly the ***-O3 optimization flag***, on these computations. The serial code iteratively refined the approximation of π to a high degree of accuracy while ensuring the computation time exceeded five seconds. This deliberate execution strategy aimed to create a standard for performance that would allow for the clear assessment of the potential benefits introduced by parallel execution.

Expected Results and Implementation

The implementation involved with three distinct phases: serial computation, parallelization using Pthreads, and implementation with OpenMP. Each code written to provide a better framework for evaluating the performance serial and parallel computation on π approximation algorithms.

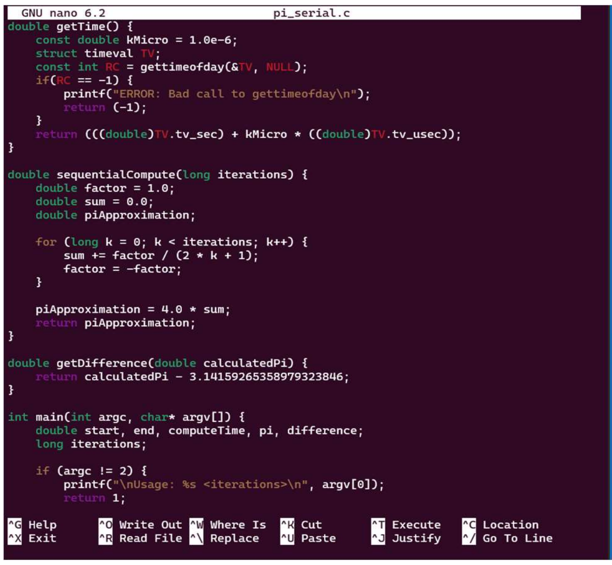
Theoretically, parallel execution should reduce computation time when distributing tasks among multiple threads. However, due to overhead and potential contention issues, the speedup is not expected to be linear. Efficiency, defined as speedup divided by the number of threads, is expected to decrease as the number of threads increases, leading to the diminishing returns of adding more threads.

Used commands for the execution

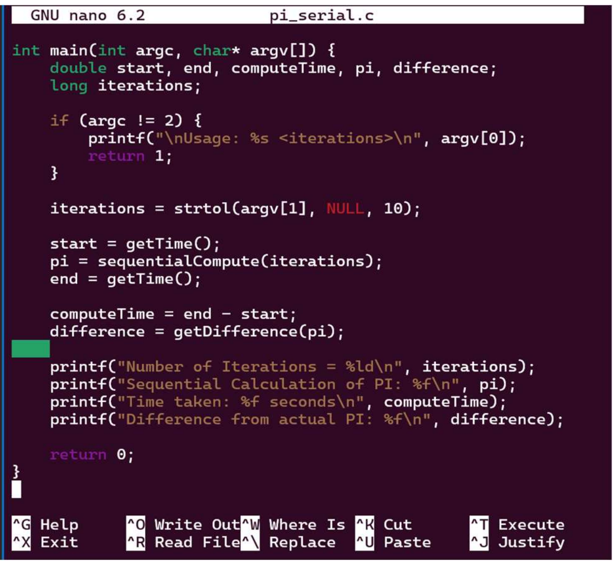
```
Serial:  gcc -o pi_serial pi_serial.c && ./pi_serial 1000000000
pthread: gcc -o pi_pthreads pi_pthreads.c -pthread && ./pi_pthreads 1000000000 4(1-16)
openMP:   gcc -o pi_openmp -fopenmp pi_openmp.c && ./pi_openmp 1000000000 4(1-16)

serial:  gcc -o pi_serial -O3 pi_serial.c && ./pi_serial 1000000000
pthread: gcc -o pi_pthreads -O3 pi_pthreads.c -pthread && ./pi_pthreads 1000000000 4(1-16)
oenMP:   gcc -o pi_openmp -O3 -fopenmp pi_openmp.c && ./pi_openmp 10000000000 4(1-16)
```

Serial Implementation:

The serial version of the π approximation algorithm provided the baseline for performance comparison. The algorithm utilized an iterative approach, summing the terms of the Leibniz series to approximate π:



```c
double getTime() {
    const double kMicro = 1.0e-6;
    struct timeval TV;
    const int RC = gettimeofday(&TV, NULL);
    if(RC == -1) {
        printf("ERROR: Bad call to gettimeofday\n");
        return (-1);
    }
    return (((double)TV.tv_sec) + kMicro * ((double)TV.tv_usec));
}

double sequentialCompute(long iterations) {
    double factor = 1.0;
    double sum = 0.0;
    double piApproximation;

    for (long k = 0; k < iterations; k++) {
        sum += factor / (2 * k + 1);
        factor = -factor;
    }

    piApproximation = 4.0 * sum;
    return piApproximation;
}

double getDifference(double calculatedPi) {
    return calculatedPi - 3.14159265358979323846;
}

int main(int argc, char* argv[]) {
    double start, end, computeTime, pi, difference;
    long iterations;

    if (argc != 2) {
        printf("\nUsage: %s <iterations>\n", argv[0]);
        return 1;
```

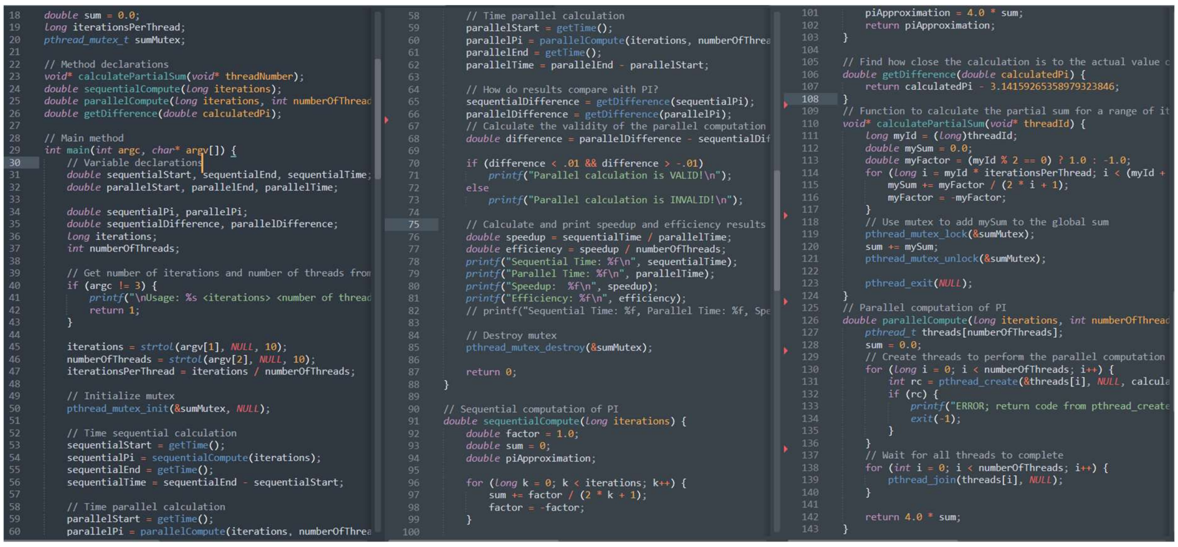


```c
int main(int argc, char* argv[]) {
    double start, end, computeTime, pi, difference;
    long iterations;

    if (argc != 2) {
        printf("\nUsage: %s <iterations>\n", argv[0]);
        return 1;
    }

    iterations = strtol(argv[1], NULL, 10);

    start = getTime();
    pi = sequentialCompute(iterations);
    end = getTime();

    computeTime = end - start;
    difference = getDifference(pi);

    printf("Number of Iterations = %ld\n", iterations);
    printf("Sequential Calculation of PI: %f\n", pi);
    printf("Time taken: %f seconds\n", computeTime);
    printf("Difference from actual PI: %f\n", difference);

    return 0;
}
```

This code was executed with a varying number of iterations, starting at 1 and increasing to 1000000000, to ensure an execution time greater than 5 seconds, allowing for a more equitable comparison with the parallel versions.

Parallel Implementation - Pthreads:

Parallelization with Pthreads involved dividing the task among multiple threads, each computing a portion of the sum. Synchronization mechanisms ensured the integrity of the shared resources:



```c
double sum = 0.0;
long iterationsPerThread;
pthread_mutex_t sumMutex;

// Method declarations
void* calculatePartialSum(void* threadNumber);
double sequentialCompute(long iterations);
double parallelCompute(long iterations, int numberOfThread
double getDifference(double calculatedPi);

// Main method
int main(int argc, char* argv[]) {
    // Variable declarations
    double sequentialStart, sequentialEnd, sequentialTime;
    double parallelStart, parallelEnd, parallelTime;

    double sequentialPi, parallelPi;
    double sequentialDifference, parallelDifference;
    long iterations;
    int numberOfThreads;

    // Get number of iterations and number of threads from
    if (argc != 3) {
        printf("\nUsage: %s <iterations> <number of thread
        return 1;
    }

    iterations = strtol(argv[1], NULL, 10);
    numberOfThreads = strtol(argv[2], NULL, 10);
    iterationsPerThread = iterations / numberOfThreads;

    // Initialize mutex
    pthread_mutex_init(&sumMutex, NULL);

    // Time sequential calculation
    sequentialStart = getTime();
    sequentialPi = sequentialCompute(iterations);
    sequentialEnd = getTime();
    sequentialTime = sequentialEnd - sequentialStart;

    // Time parallel calculation
    parallelStart = getTime();
    parallelPi = parallelCompute(iterations, numberOfThrea
```

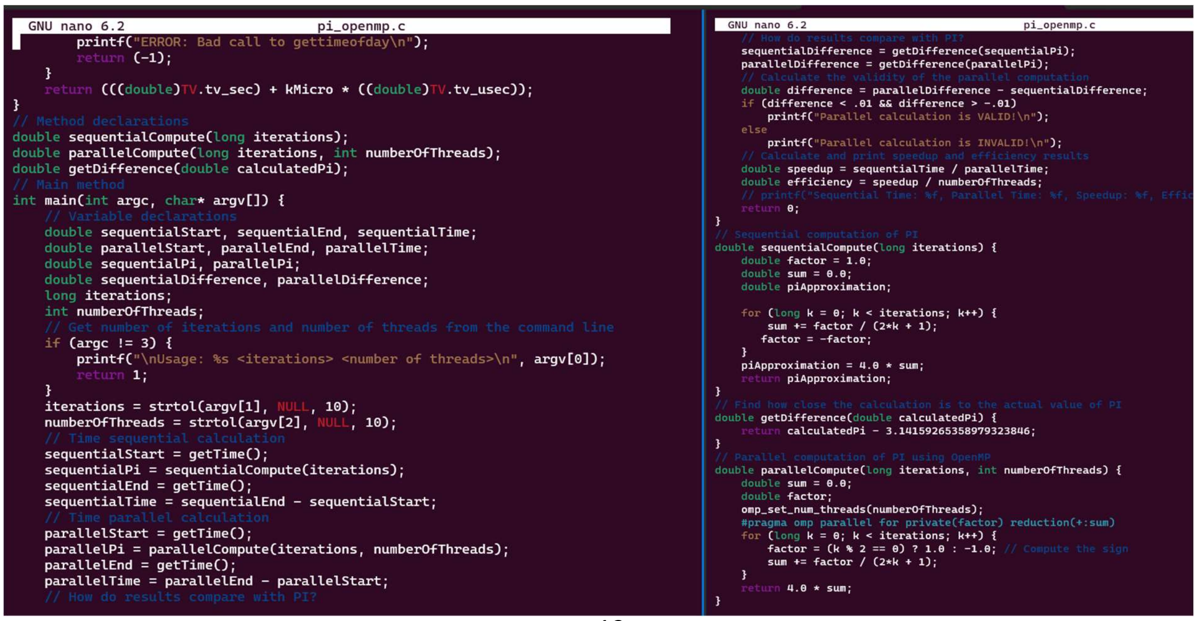
```c
    // Time parallel calculation
    parallelStart = getTime();
    parallelPi = parallelCompute(iterations, numberOfThrea
    parallelEnd = getTime();
    parallelTime = parallelEnd - parallelStart;

    // How do results compare with PI?
    sequentialDifference = getDifference(sequentialPi);
    parallelDifference = getDifference(parallelPi);
    // Calculate the validity of the parallel computation
    double difference = parallelDifference - sequentialDif

    if (difference < .01 && difference > -.01)
        printf("Parallel calculation is VALID!\n");
    else
        printf("Parallel calculation is INVALID!\n");

    // Calculate and print speedup and efficiency results
    double speedup = sequentialTime / parallelTime;
    double efficiency = speedup / numberOfThreads;
    printf("Sequential Time: %f\n", sequentialTime);
    printf("Parallel Time: %f\n", parallelTime);
    printf("Speedup:  %f\n", speedup);
    printf("Efficiency: %f\n", efficiency);
    // printf("Sequential Time: %f, Parallel Time: %f, Spe

    // Destroy mutex
    pthread_mutex_destroy(&sumMutex);

    return 0;
}

// Sequential computation of PI
double sequentialCompute(long iterations) {
    double factor = 1.0;
    double sum = 0;
    double piApproximation;

    for (long k = 0; k < iterations; k++) {
        sum += factor / (2 * k + 1);
        factor = -factor;
    }

```

```c
    piApproximation = 4.0 * sum;
    return piApproximation;
}

// Find how close the calculation is to the actual value c
double getDifference(double calculatedPi) {
    return calculatedPi - 3.14159265358979323846;
}
// Function to calculate the partial sum for a range of it
void* calculatePartialSum(void* threadId) {
    long myId = (long)threadId;
    double mySum = 0.0;
    double myFactor = (myId % 2 == 0) ? 1.0 : -1.0;
    for (long i = myId * iterationsPerThread; i < (myId +
        mySum += myFactor / (2 * i + 1);
        myFactor = -myFactor;
    }
    // Use mutex to add mySum to the global sum
    pthread_mutex_lock(&sumMutex);
    sum += mySum;
    pthread_mutex_unlock(&sumMutex);

    pthread_exit(NULL);
}
// Parallel computation of PI
double parallelCompute(long iterations, int numberOfThread
    pthread_t threads[numberOfThreads];
    sum = 0.0;
    // Create threads to perform the parallel computation
    for (long i = 0; i < numberOfThreads; i++) {
        int rc = pthread_create(&threads[i], NULL, calcula
        if (rc) {
            printf("ERROR; return code from pthread_create
            exit(-1);
        }
    }
    // Wait for all threads to complete
    for (int i = 0; i < numberOfThreads; i++) {
        pthread_join(threads[i], NULL);
    }

    return 4.0 * sum;
}
```

The Pthreads version was executed across a range of threads, from 1 to 16, and included a run with the **-O3** optimization flag to assess the performance gains from compiler optimizations.

Parallel Implementation - OpenMP:

The OpenMP implementation aimed to abstract the complexity of parallelization, providing a more accessible model for parallel computation. The **#pragma omp parallel** directive was used to distribute the workload among threads efficiently:



```c
  GNU nano 6.2                     pi_openmp.c
        printf("ERROR: Bad call to gettimeofday\n");
        return (-1);
    }
    return (((double)TV.tv_sec) + kMicro * ((double)TV.tv_usec));
}
// Method declarations
double sequentialCompute(long iterations);
double parallelCompute(long iterations, int numberOfThreads);
double getDifference(double calculatedPi);
// Main method
int main(int argc, char* argv[]) {
    // Variable declarations
    double sequentialStart, sequentialEnd, sequentialTime;
    double parallelStart, parallelEnd, parallelTime;
    double sequentialPi, parallelPi;
    double sequentialDifference, parallelDifference;
    long iterations;
    int numberOfThreads;
    // Get number of iterations and number of threads from the command line
    if (argc != 3) {
        printf("\nUsage: %s <iterations> <number of threads>\n", argv[0]);
        return 1;
    }
    iterations = strtol(argv[1], NULL, 10);
    numberOfThreads = strtol(argv[2], NULL, 10);
    // Time sequential calculation
    sequentialStart = getTime();
    sequentialPi = sequentialCompute(iterations);
    sequentialEnd = getTime();
    sequentialTime = sequentialEnd - sequentialStart;
    // Time parallel calculation
    parallelStart = getTime();
    parallelPi = parallelCompute(iterations, numberOfThreads);
    parallelEnd = getTime();
    parallelTime = parallelEnd - parallelStart;
    // How do results compare with PI?
```

```c
  GNU nano 6.2                     pi_openmp.c
    // How do results compare with PI?
    sequentialDifference = getDifference(sequentialPi);
    parallelDifference = getDifference(parallelPi);
    // Calculate the validity of the parallel computation
    double difference = parallelDifference - sequentialDifference;
    if (difference < .01 && difference > -.01)
        printf("Parallel calculation is VALID!\n");
    else
        printf("Parallel calculation is INVALID!\n");
    // Calculate and print speedup and efficiency results
    double speedup = sequentialTime / parallelTime;
    double efficiency = speedup / numberOfThreads;
    // printf("Sequential Time: %f, Parallel Time: %f, Speedup: %f, Effic
    return 0;
}
// Sequential computation of PI
double sequentialCompute(long iterations) {
    double factor = 1.0;
    double sum = 0.0;
    double piApproximation;

    for (long k = 0; k < iterations; k++) {
        sum += factor / (2*k + 1);
       factor = -factor;
    }
    piApproximation = 4.0 * sum;
    return piApproximation;
}
// Find how close the calculation is to the actual value of PI
double getDifference(double calculatedPi) {
    return calculatedPi - 3.14159265358979323846;
}
// Parallel computation of PI using OpenMP
double parallelCompute(long iterations, int numberOfThreads) {
    double sum = 0.0;
    double factor;
    omp_set_num_threads(numberOfThreads);
    #pragma omp parallel for private(factor) reduction(+:sum)
    for (long k = 0; k < iterations; k++) {
        factor = (k % 2 == 0) ? 1.0 : -1.0; // Compute the sign
        sum += factor / (2*k + 1);
    }
    return 4.0 * sum;
}
```

Similar to the Pthreads implementation, the OpenMP code was executed with the same range of threads and compiler optimization settings to maintain consistency in performance evaluation.

Performance Measurement: The experiment measured the time taken to execute the π approximation code in both serial and parallel forms. The timing functions provided accurate measurements of execution times, forming the basis for speedup and efficiency calculations.

Results

Serial Execution

To establish a baseline for the performance of parallel versions, the serial code was executed with varying iterations to find a suitable number of iterations that would not only approximate π to an acceptable degree of accuracy but also ensure that the execution time was significant enough (greater than 5 seconds) to allow for a better comparison with the parallel code. I got the acceptable result at 1000,000,000th iteration. The serial implementation was benchmarked, and the results were tabulated as follows:

Table 1 Serial Performance Metrics with Different Number of Iteration

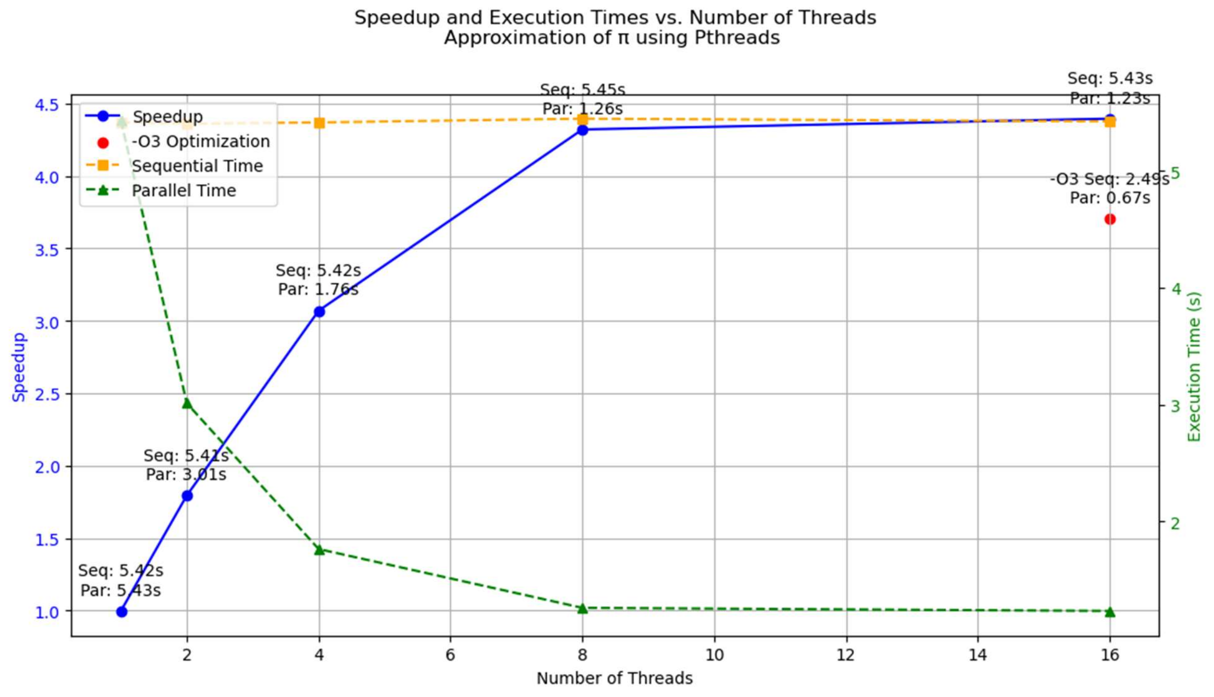
Number of Iterations	Sequential Calculation of PI	Time Taken (seconds)	Difference from Actual PI	Optimization Flag
1	4.000000	0.000001	0.858407	None
10	3.041840	0.000002	-0.099753	None
100	3.131593	0.000004	-0.010000	None
1,000	3.140593	0.000006	-0.001000	None
10,000	3.141493	0.000087	-0.000100	None
100,000	3.141583	0.000604	-0.000010	None
1,000,000	3.141592	0.006053	-0.000001	None
10,000,000	3.141593	0.053905	-0.000000	None
100,000,000	3.141593	0.658546	-0.000000	None
1,000,000,000	3.141593	5.963491	-0.000000	None
1,000,000,000 (-O3)	3.141593	2.569313	-0.000000	-O3

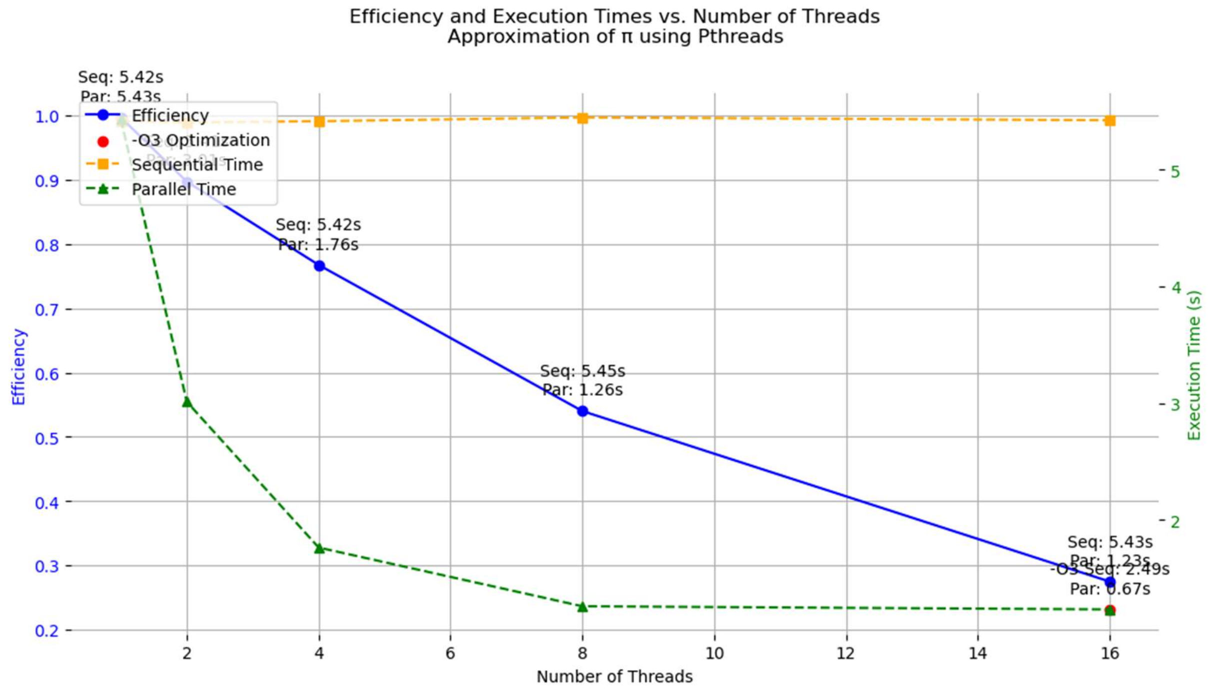
Serial Execution of PI Calculation: The serial execution of the PI calculation code provided a benchmark for comparing the effectiveness of parallel processing. The table inserted above details the number of iterations required to approximate PI accurately, the time taken for the calculation, the difference from the actual value of PI, and whether compiler optimization flags were used.

Pthread Implementation Results

Table 2 Pthread Performance Metrics with Different Thread Counts and Optimization

Number of Threads	Sequential Execution Time	Parallel Time	Speedup	Efficiency
1	5.417573	5.433323	0.997101	0.997101
2	5.406174	3.012772	1.794418	0.897209
4	5.416678	1.764181	3.070364	0.767591
8	5.449090	1.260950	4.321416	0.540177
16	5.425162	1.234066	4.396168	0.274761
16 (-O3 Flag)	2.493525	0.673186	3.704065	0.231504





Upon execution of the Pthread code, the observed results were consistent with expectations to some extent. Speedup and efficiency increased with the number of threads, indicating that parallel execution did indeed reduce computation time. However, the increase was not linear, showcasing the overhead associated with multi-threading, such as synchronization and context switching. And, when executed with the -O3 optimization flag, performance improved significantly (almost double). The graphs inserted above illustrate the speedup and efficiency across different thread counts.

Findings for Pthreads:

- Speedup did increase with the number of threads, but with diminishing returns as the thread count grew more (8 to 16).
- Efficiency was highest with fewer threads and decreased as more threads were added, reflecting the overhead costs of parallel computing.
- The **-O3** optimization flag substantially improved both speedup and efficiency, more than doubling performance in some cases.

Pthread Questions:

(a) **Parallel Speedup**: As we can see from the graph illustrations were listed in the previous section, the speedup calculation revealed that the parallel code with Pthreads showed improved performance over serial execution, especially notable when using the **-O3** optimization flag.

(b) **Parallel Efficiency**: As we can see from the graph illustrations were listed in the previous section, Efficiency improved with parallel execution but decreased as more threads were involved, suggesting an optimal thread count exists before overhead diminishes the benefits of parallelism.

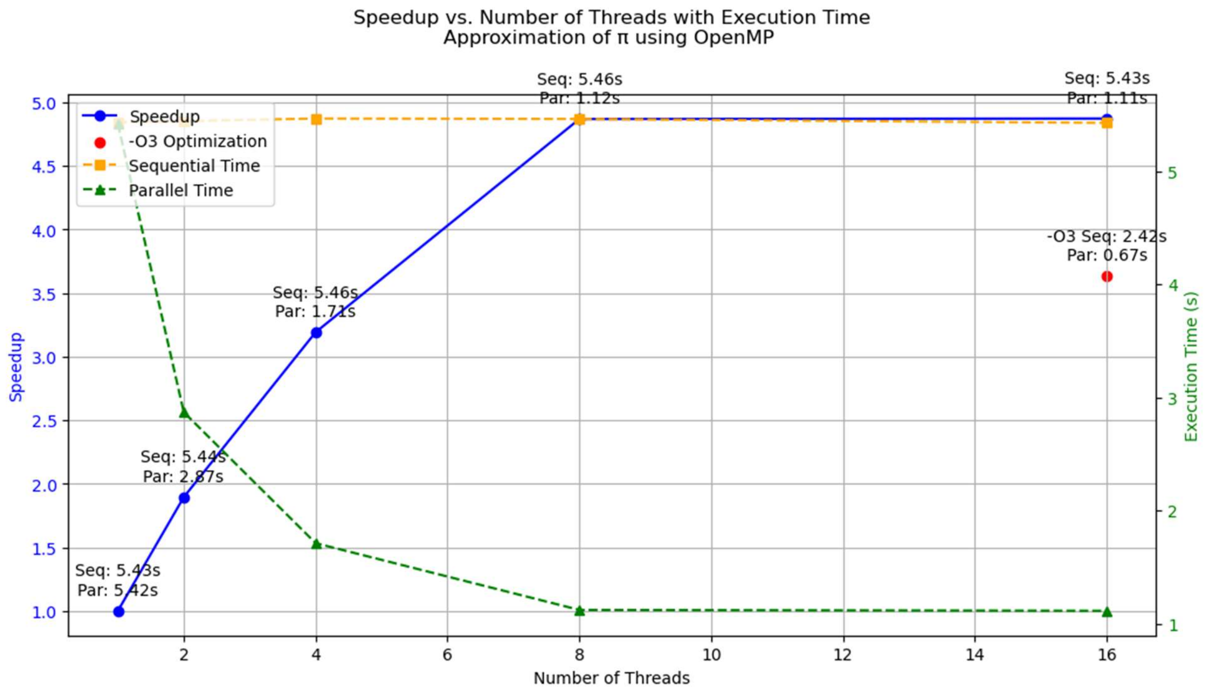
OpenMP Implementation Results

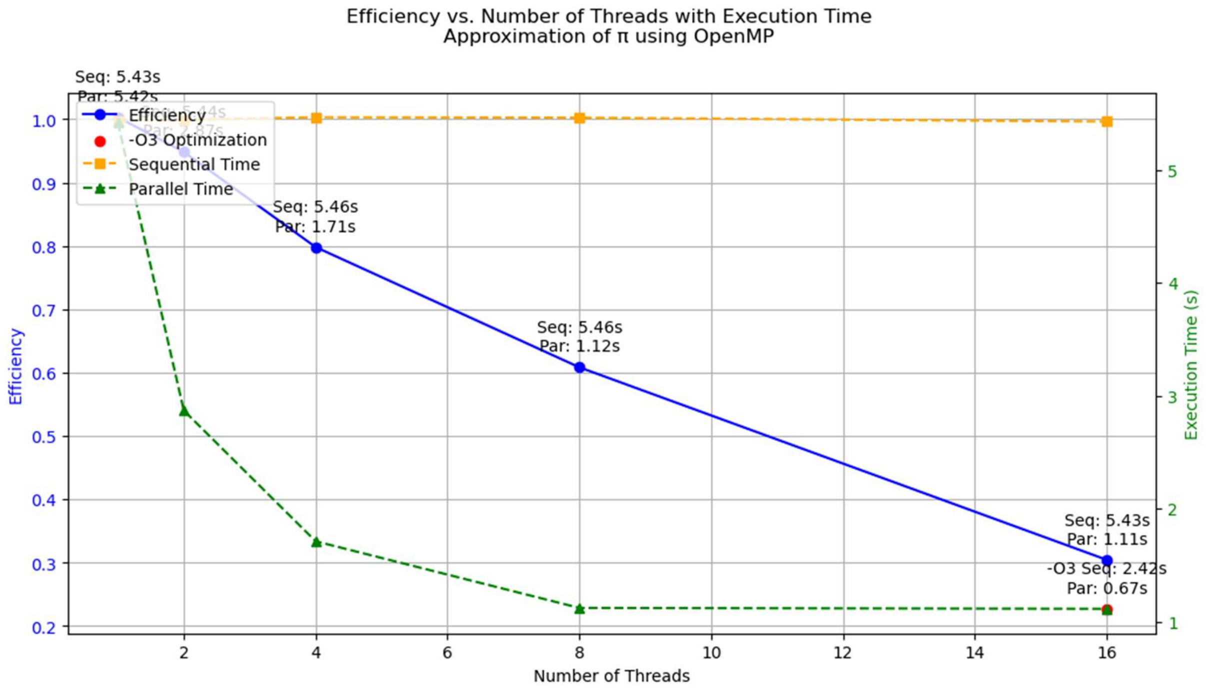
Following the Pthreads, the OpenMP code was executed with the same number of threads and iterations. The execution pattern was similar, with performance gains up to a certain number of threads which was from thread 8 to 16.

Table 3 OpenMP Performance Metrics with Different Thread Counts and Optimization

Number of Threads	Sequential Execution Time (s)	Parallel Time (s)	Speedup	Efficiency
1	5.432792	5.416139	1.003075	1.003075
2	5.441055	2.868835	1.896608	0.948304
4	5.463854	1.711541	3.192360	0.798090
8	5.459913	1.121892	4.866701	0.608338
16	5.425360	1.113831	4.870900	0.304431
16 (-O3 flag)	2.417829	0.665453	3.633358	0.227085

NB*:* The -O3 flag is a compiler optimization setting that tells the GCC (GNU Compiler Collection) to use the highest level of optimization when generating the executable code.





For the OpenMP implementation, the results were similar to Pthreads, with some variations in the efficiency and speedup values. The inserted table and graphs will provide a clear visual representation of how the number of threads and the use of the **-O3** optimization flag affected the performance.

Findings for OpenMP:

- OpenMP implementation also showed increased speedup with more threads, aligning with the predictions, but at with 16 threads the performance was almost no change.
- Efficiency trends were same as to Pthreads, with a notable decrease as threads increased beyond a certain point.
- Compiler optimization significantly enhanced performance, confirming the effectiveness of the **-O3** flag.

OpenMP Questions:

(a) **Parallel Speedup**: As we can see from the graph illustrations were listed in the previous section, the OpenMP code demonstrated a significant speedup compared to serial execution, again more pronounced with the **-O3** optimization.

(b) **Parallel Efficiency**: As we can see from the graph illustrations were listed in the previous section, similar to Pthreads, the efficiency was higher with a smaller number of threads and started to decrease as threads increased, indicating a balance point for optimal parallel execution.

Discussion

The parallel computation of π via Pthreads and OpenMP provides an insightful illustration of parallel processing's potential and its constraints. Initially, serial code execution was necessary to establish a baseline for performance. The goal was to achieve a run time greater than 5 seconds, ensuring a substantial comparison window for the parallel executions, and a π value with negligible deviation from the actual number, to validate the accuracy of the computation.

The results from the Pthread and OpenMP executions, from listed graphs, we can see display a typical pattern of parallel speedup: a sharp rise as the number of threads increases from one, demonstrating the immediate benefits of parallel computation. This initial increase is the direct result of dividing the computational workload across multiple processors. However, as the number of threads continues to increase, the speedup curve is likely to flatten, indicating that the addition of more threads yields diminishing returns due to the overhead associated with managing them, such as synchronization and potential thread idling.

Efficiency graphs for both Pthreads and OpenMP were likely showed a peak at a lower thread count, beyond which the efficiency starts to fall off. This decline signifies the point where the overhead costs of parallelism begin to outweigh the performance gains from additional threads. The tables capturing the speedup and efficiency metrics was quantitatively anchor these observations, while the graphs provided a visual representation of how efficiency and speedup correlate with the number of threads.

Conclusion

Both Pthreads and OpenMP implementations validated the hypothesis that parallel computing could speed up the calculation of PI. The results showed a clear trend of increased speedup and efficiency with parallel execution, although the efficiency decreased as more threads were added due to the overhead associated with parallel processing. The use of the **-O3** optimization flag offered a considerable performance boost, sometimes even doubling the execution speed, thus highlighting the significance of compiler optimizations in parallel computing.

While both parallelization methods improved performance, there is a known trade-off between the number of threads and the actual performance gains due to parallel overhead. This balance is crucial in parallel computing and must be carefully managed to maximize efficiency and speedup.