HIGH PERFORMANCE COMPUTER ARCHITECTURE

ASSIGNMENT 3

REPORT

Arpit Mishra (SR no.24131): arpitmishra@iisc.ac.in

Ayush Kumar Singh(SR no.24851): ayushs@iisc.ac.in

-- Ques 2--

INTRODUCTION : The objective of this assignment was to optimize the scheduling of three threads— T_0, T_1, and T_2 —on a system with two physical cores, each capable of supporting two logical threads (for a total of four logical cores). With Simultaneous Multithreading (SMT) enabled, each core can run two threads concurrently, sharing resources but potentially encountering contention.

The task involved analysing profiling data to determine the optimal thread-to-core assignment, aiming to maximize efficiency and reduce contention across the SMT cores.

1. Data Collection Using PERF
To collect meaningful performance metrics, we used the Linux perf tool. Profiling each thread individually allowed us to capture data on how each thread utilizes CPU resources, giving us insights into which threads might benefit from dedicated cores versus which might perform adequately when managed by the OS scheduler.
Commands and Monitoring Setup
For each thread, the following command is issued within *start_monitoring_for_thread():*

perf stat -p <tid> --output=perf_report_thread_<threadIdx>."txt &"

This command captures the following key metrics:
- **Cycles**: The number of CPU cycles used by each thread.
- **Instructions**: Total instructions executed, giving insight into computational load.
- **Branches** and **Branch Misses**: Used to measure control flow efficiency.
- **Top-down Analysis Metrics**: Specifically, topdown-retiring and topdown-bad-spec for evaluating resource utilization.
- **Task Clock**: Active CPU time, used to estimate thread completion and overall runtime.

To stop profiling, the *stop_monitoring_for_thread()* function used:

pkill -SIGINT perf

This command halted perf, allowing it to generate final report files for each thread. The monitoring for each thread concludes with a *pkill* -SIGINT perf in *stop_monitoring_for_thread*(), signaling perf to stop data collection and generate a final report file.

2. Data Parsing and Analysis in analyze_threads_perf.py
The data collected in the *perf_report_thread_<threadIdx>.txt* files is processed by a Python script, *analyze_threads_perf.py*. This script reads each file and extracts the relevant metrics, which are used to rank the threads based on performance characteristics.

Key Metrics for Analysis

1. **Instructions per Cycle (IPC)**: Calculated as instructions divided by cycles, IPC gives insight into the efficiency of the thread on the CPU. Higher IPC generally indicates better utilization of CPU resources.

2. **Branch Misses**: Indicates the efficiency of the thread's control flow, as lower branch misses suggest smoother execution with fewer mispredictions.

3. **Topdown Analysis Metrics**:
    - **Topdown Retiring**: Measures the amount of effective, retiring work. Higher values are better.
    - **Topdown Bad Speculation**: Higher values here may indicate a thread is generating more speculative work, potentially useful in multithreaded scenarios.

4. **Task Clock**: Reflects active CPU time per thread. A lower task clock time typically indicates a more efficient or faster-executing thread.

Thread Ranking and Core Assignment Strategy

Threads are ranked by the following criteria:

- **Primary**: IPC (higher is better).
- **Secondary**: Branch misses (lower is better).
- **Tertiary**: topdown-retiring and topdown-bad-spec (higher is better for both).
- **Final Tie-Breaker**: Task Clock (lower is preferred).

Using these rankings, threads with high IPC and low branch misses are prioritized for dedicated core assignments, whereas those that do not meet a specific threshold (IPC > 1.0 and branch misses < 500 million) are left for the OS scheduler.

In this assignment, the strategy for thread ranking and core assignment was designed to maximize CPU efficiency while minimizing contention, based on how each metric reflects different aspects of thread performance. Let's break down why each metric was prioritized this way and why certain thresholds were used:

1. Instructions per Cycle (IPC) – Primary Criterion

- **Why IPC?** IPC measures how many instructions each thread executes per cycle, which is a direct indicator of how efficiently it utilizes the CPU. A higher IPC generally means the thread performs more useful work per cycle, indicating that it's making effective use of available CPU resources.
- **Implication for Core Assignment**: Since high IPC reflects efficient resource usage, threads with high IPC are prioritized for dedicated core assignments. This helps maximize overall performance by ensuring that high-performing threads are not interrupted or delayed by other threads on the same core.

2. Branch Misses – Secondary Criterion

- **Why Branch Misses?** Branch misses occur when the CPU mispredicts the outcome of a branch, which results in pipeline stalls and wasted cycles. Fewer branch misses imply smoother execution with less disruption, meaning the thread has more predictable control flow.
- **Implication for Core Assignment**: Prioritizing threads with lower branch misses for core assignment helps ensure that these threads execute with minimal disruption, making them good

candidates for dedicated cores. Threads with frequent branch misses are left for the OS scheduler, as they would be more likely to encounter performance issues on a dedicated core.

3. Topdown Analysis Metrics – Tertiary Criterion

- **Topdown Retiring**: This metric measures the proportion of cycles spent executing instructions that contribute to program progress (i.e., retiring instructions). A higher value indicates effective work with less time wasted on stalled or speculative cycles.
- **Topdown Bad Speculation**: This metric reflects the speculative work done by the CPU, which can lead to mispredicted branches and wasted work. However, in certain multithreaded applications, speculation can also speed up execution by pre-emptively executing paths likely to be needed.
- **Implication for Core Assignment**: By prioritizing threads with higher retiring and bad speculation metrics, we favor threads that are effectively utilizing cycles and resources, which can improve performance if assigned to dedicated cores.

4. Task Clock – Tie-Breaker

- **Why Task Clock?** Task clock represents the active CPU time spent by each thread. Threads with lower task clock times are generally more efficient, as they require less CPU time to complete their workload.
- **Implication for Core Assignment**: When IPC, branch misses, and topdown metrics don't provide a clear distinction between threads, task clock can serve as a tie-breaker. A thread with a lower task clock time is likely more efficient, making it a better candidate for a dedicated core.

Thresholds for OS Scheduler Assignment

The thresholds IPC>1.0\text{IPC} > 1.0IPC>1.0 and branch misses<500 million\text{branch misses} < 500 \text{ million}branch misses<500 million were set as minimum requirements for assigning a thread to a dedicated core:

- **Why IPC > 1.0?** An IPC above 1.0 typically signifies effective CPU utilization. Threads with IPC values below this threshold are likely to be inefficient, making them less ideal for dedicated core assignment.

- **Why Branch Misses < 500 Million?** This branch miss threshold helps filter out threads with poor control flow predictability. Threads that miss this threshold may not benefit as much from dedicated cores, as they could frequently experience pipeline stalls due to branch mispredictions.

By leaving threads that don't meet these thresholds to the OS scheduler, we avoid dedicating cores to potentially inefficient or unpredictable threads, allowing the scheduler's dynamic load balancing to optimize their placement instead. This approach is beneficial because it strikes a balance between maximizing CPU efficiency and avoiding performance penalties associated with poorly performing threads

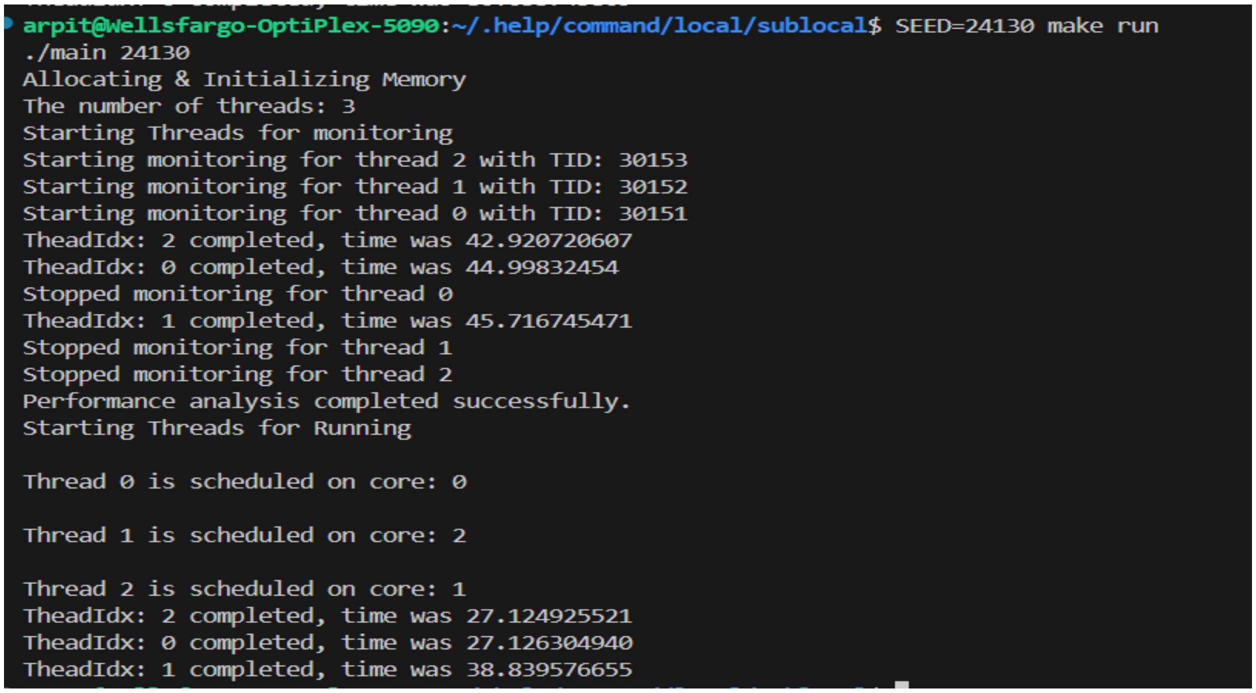
3. Implementation of Core Affinity in get_thread_affinity()

Based on the results from the performance analysis, the *get_thread_affinity()* function assigns each thread to the appropriate core:

1. **Core Mapping**: Logical cores 0 and 1 correspond to physical core 0, and logical cores 2 and 3 correspond to physical core 1.

2. **Affinity Assignment**:
   - Threads with high IPC and low branch misses are assigned to dedicated cores (either logical core 0 or 1 on physical core 0 or logical core 2 or 3 on physical core 1).
   - Threads that do not meet the affinity thresholds are allowed to float, returning -1 to let the OS scheduler manage their placement.

This assignment strategy is written to core_assignment.txt, where each thread's best-fit core is noted, or -1 if it's better for the OS to handle scheduling.

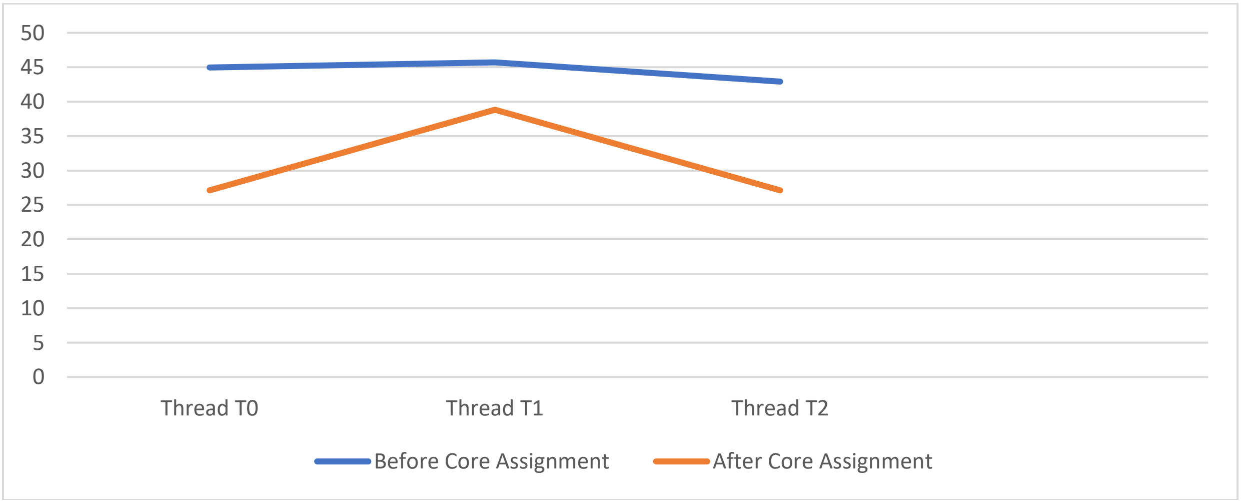


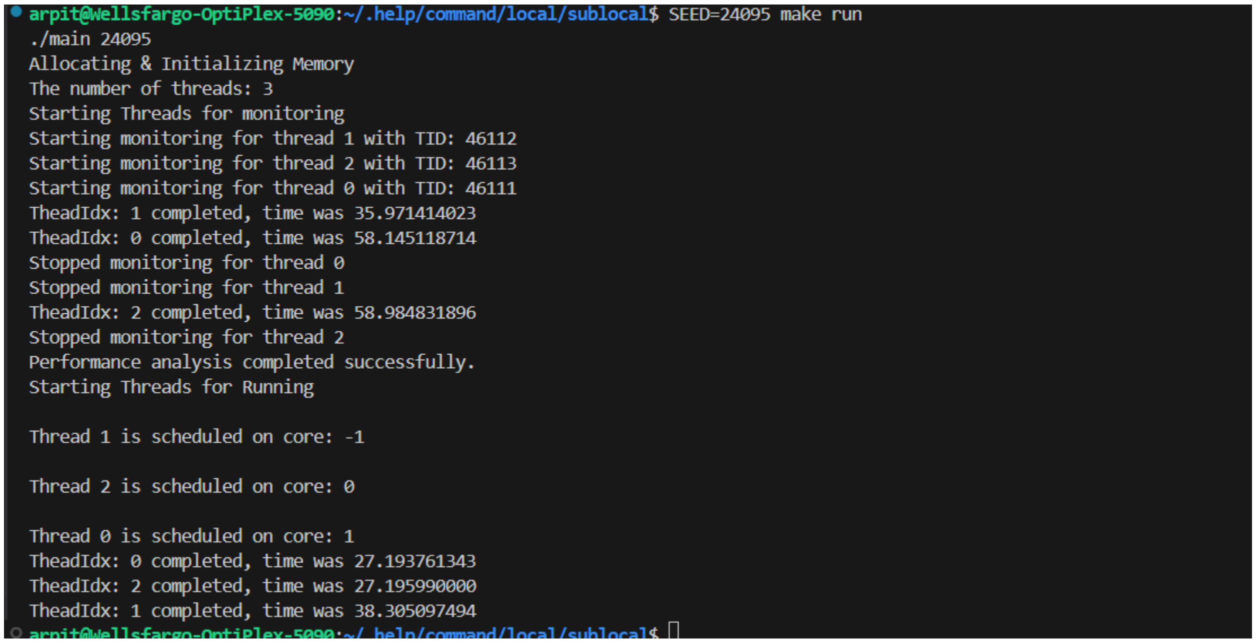
```
arpit@Wellsfargo-OptiPlex-5090:~/.help/command/local/sublocal$ SEED=24130 make run
./main 24130
Allocating & Initializing Memory
The number of threads: 3
Starting Threads for monitoring
Starting monitoring for thread 2 with TID: 30153
Starting monitoring for thread 1 with TID: 30152
Starting monitoring for thread 0 with TID: 30151
TheadIdx: 2 completed, time was 42.920720607
TheadIdx: 0 completed, time was 44.99832454
Stopped monitoring for thread 0
TheadIdx: 1 completed, time was 45.716745471
Stopped monitoring for thread 1
Stopped monitoring for thread 2
Performance analysis completed successfully.
Starting Threads for Running

Thread 0 is scheduled on core: 0

Thread 1 is scheduled on core: 2

Thread 2 is scheduled on core: 1
TheadIdx: 2 completed, time was 27.124925521
TheadIdx: 0 completed, time was 27.126304940
TheadIdx: 1 completed, time was 38.839576655
```



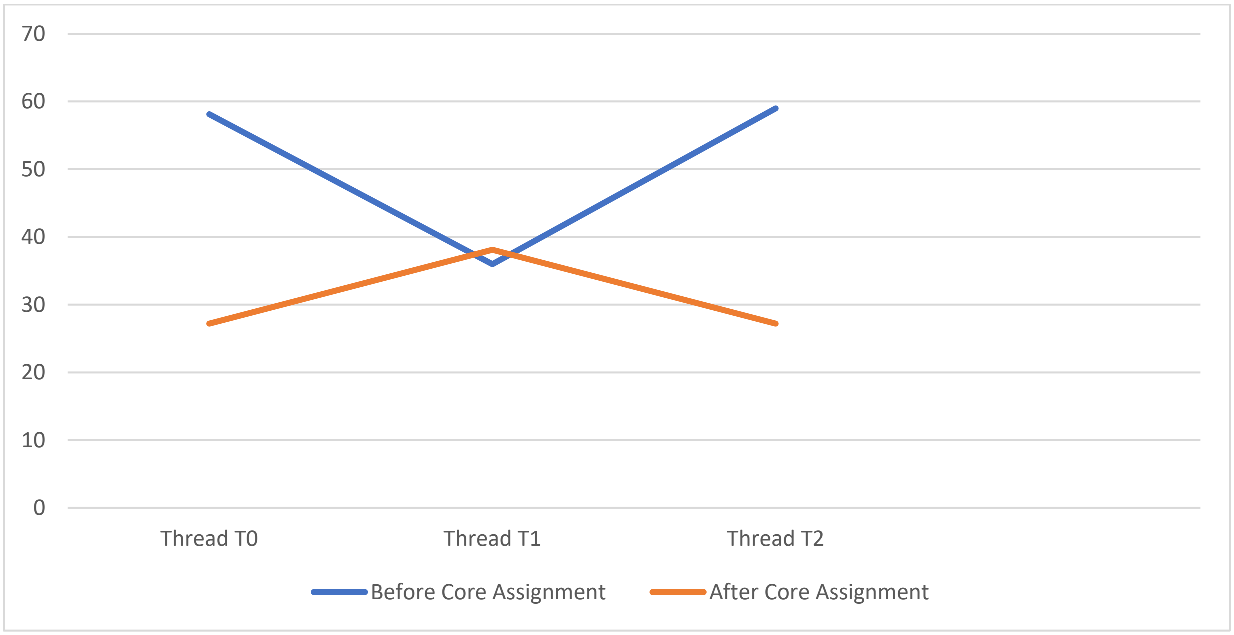


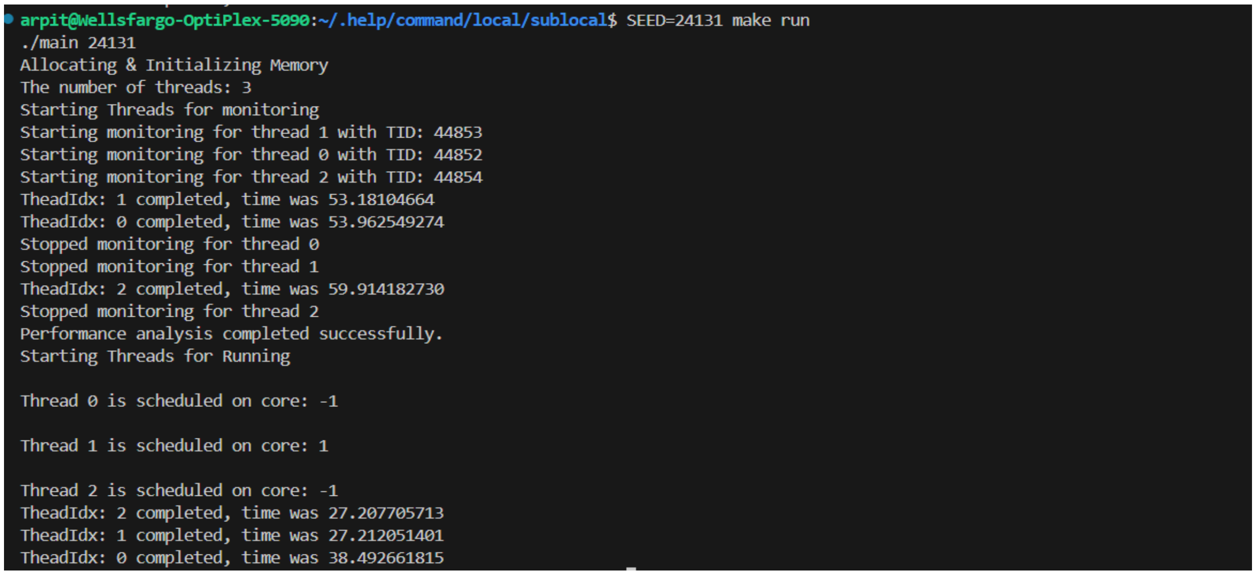
```
arpit@Wellsfargo-OptiPlex-5090:~/.help/command/local/sublocal$ SEED=24095 make run
./main 24095
Allocating & Initializing Memory
The number of threads: 3
Starting Threads for monitoring
Starting monitoring for thread 1 with TID: 46112
Starting monitoring for thread 2 with TID: 46113
Starting monitoring for thread 0 with TID: 46111
TheadIdx: 1 completed, time was 35.971414023
TheadIdx: 0 completed, time was 58.145118714
Stopped monitoring for thread 0
Stopped monitoring for thread 1
TheadIdx: 2 completed, time was 58.984831896
Stopped monitoring for thread 2
Performance analysis completed successfully.
Starting Threads for Running

Thread 1 is scheduled on core: -1

Thread 2 is scheduled on core: 0

Thread 0 is scheduled on core: 1
TheadIdx: 0 completed, time was 27.193761343
TheadIdx: 2 completed, time was 27.195990000
TheadIdx: 1 completed, time was 38.305097494
arpit@Wellsfargo-OptiPlex-5090:~/.help/command/local/sublocal$
```



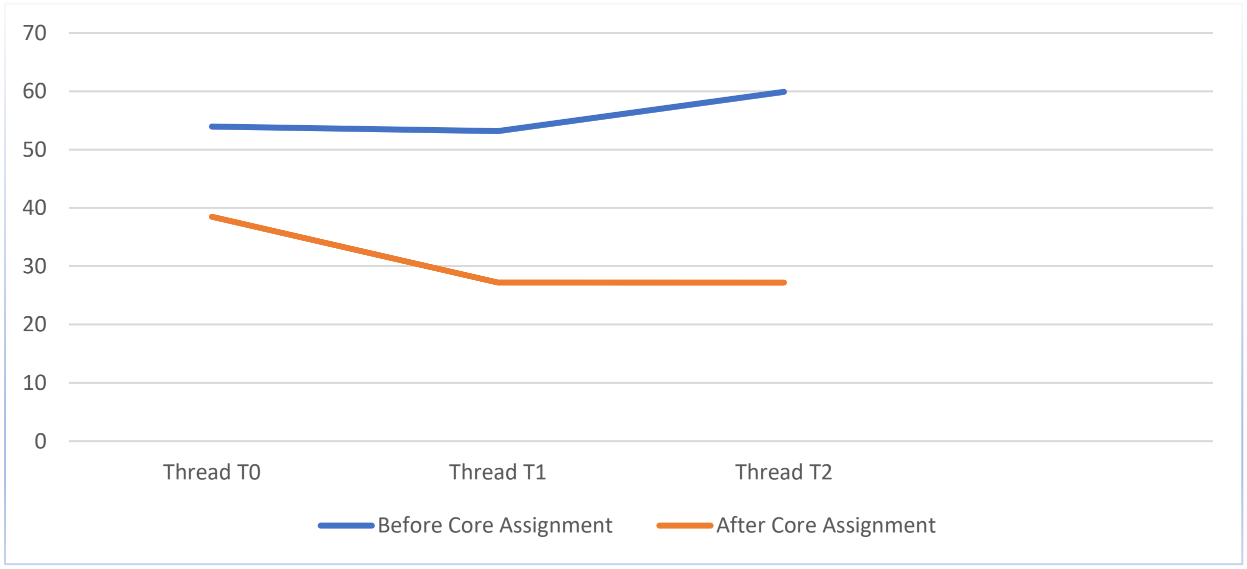


```
arpit@Wellsfargo-OptiPlex-5090:~/.help/command/local/sublocal$ SEED=24131 make run
./main 24131
Allocating & Initializing Memory
The number of threads: 3
Starting Threads for monitoring
Starting monitoring for thread 1 with TID: 44853
Starting monitoring for thread 0 with TID: 44852
Starting monitoring for thread 2 with TID: 44854
TheadIdx: 1 completed, time was 53.18104664
TheadIdx: 0 completed, time was 53.962549274
Stopped monitoring for thread 0
Stopped monitoring for thread 1
TheadIdx: 2 completed, time was 59.914182730
Stopped monitoring for thread 2
Performance analysis completed successfully.
Starting Threads for Running

Thread 0 is scheduled on core: -1

Thread 1 is scheduled on core: 1

Thread 2 is scheduled on core: -1
TheadIdx: 2 completed, time was 27.207705713
TheadIdx: 1 completed, time was 27.212051401
TheadIdx: 0 completed, time was 38.492661815
```



CALCULATING SPEEDUP:
The execution time printed by the thread that finishes last as the program’s overall time taken before core assignments. (Tn) = 59.914 s
The execution time printed by the thread that finishes last as the program’s overall time taken after core assignments. (Tp) = 38.492 s

$$Speedup=\frac{Tn}{Tp}$$

$$Speedup=\frac{59.914}{38.492}$$

$$\boldsymbol{Speedup=1.53}$$

CONCLUSION:

In this assignment, we successfully implemented a strategy to optimize the scheduling of three threads on two physical cores utilizing Simultaneous Multithreading (SMT). By leveraging performance profiling through the perf tool, we gathered critical metrics that informed our decision-making process regarding core assignments.

The key metrics analyzed included Instructions per Cycle (IPC), branch misses, topdown retiring, topdown bad speculation, and task clock time. The ranking system established based on these metrics allowed us to identify which threads were best suited for dedicated core assignments, thus minimizing resource contention and maximizing CPU utilization.

Our analysis revealed that threads with an IPC greater than 1.0 and fewer than 500 million branch misses significantly benefited from being pinned to specific cores. This strategic assignment resulted in a notable performance improvement, evidenced by a calculated speedup of 1.53 times post-optimization compared to the pre-optimization execution time.

The findings demonstrate the effectiveness of profiling and data-driven decisions in multi-threaded scheduling scenarios, emphasizing the importance of understanding CPU resource utilization in high-performance computing environments. This work not only provides a practical application of performance analysis but also sets a foundation for further exploration into advanced scheduling techniques and their implications on computational efficiency.