Parallel & Distributed Computing

Assignment 1

Anas Bin Rashid
22I-0907
CS-6A

Table of Contents

Tasks Chosen

2. Word Analysis in a Large Text File

Dataset Link: https://www.euromatrixplus.net/multi-un/

3. Term Frequency Analysis in a Corpus

Dataset Link: https://www.euromatrixplus.net/multi-un/

Links for Reference

YouTube Video Link: https://youtu.be/l33r9ZE-FVA

Dataset Link: https://www.euromatrixplus.net/multi-un/

Solution Approach

The goal of this implementation was to optimize word frequency analysis using multithreading, leveraging CPU cores efficiently to reduce execution time while maintaining accuracy.
To achieve high performance, the solution focused on three key areas:

1. Parallelization Strategy – Efficient distribution of workload among threads.
2. Synchronization & Data Sharing – Optimizing memory access to avoid contention.
3. Scalability & Affinity – Maximizing CPU utilization while avoiding bottlenecks.

Parallelization Strategy

The primary challenge was to divide the dataset evenly among multiple threads without causing load imbalance or excessive synchronization overhead.

Dynamic Task Queue-Based Processing

Instead of statically assigning chunks to threads (which can lead to uneven workloads), a task queue approach was implemented:

- The dataset was split into fixed-size chunks (4096 bytes each).
- These chunks were pushed into a queue, and threads dynamically pulled tasks when ready.
- This ensured better load balancing because no thread was left idle while others were still processing.

This approach scales well and prevents the issue of some threads finishing early while others remain occupied.

Thread Synchronization & Memory Access Optimization

Minimizing Lock Contention

One major performance bottleneck in multithreading is mutex contention, where multiple threads try to access the same data structure simultaneously.
To solve this:

- Each thread maintained its own local word frequency map while processing chunks.
- Only after processing, local maps were merged into a global word frequency map using mutex locks.
- This greatly reduced synchronization overhead and improved performance.

Efficient Memory Access & Cache Optimization

To minimize memory bandwidth bottlenecks, the following optimizations were made:

- Thread-local storage was used to improve cache locality.
- Fixed chunk sizes (4096 bytes) ensured that each thread accessed contiguous memory, reducing cache misses.
- Avoided unnecessary memory allocations inside loops, leading to faster processing.

CPU Affinity for Cache Locality & Core Utilization

To further optimize CPU usage, thread affinity was applied in Q2_Affinity & Q3_Affinity versions using: *pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);*

- This binds threads to specific CPU cores, reducing context switching overhead.
- Helps reuse cached data, minimizing memory access latency.

Vtune Performance Results

Question 2 (Without Affinity)

1 Thread

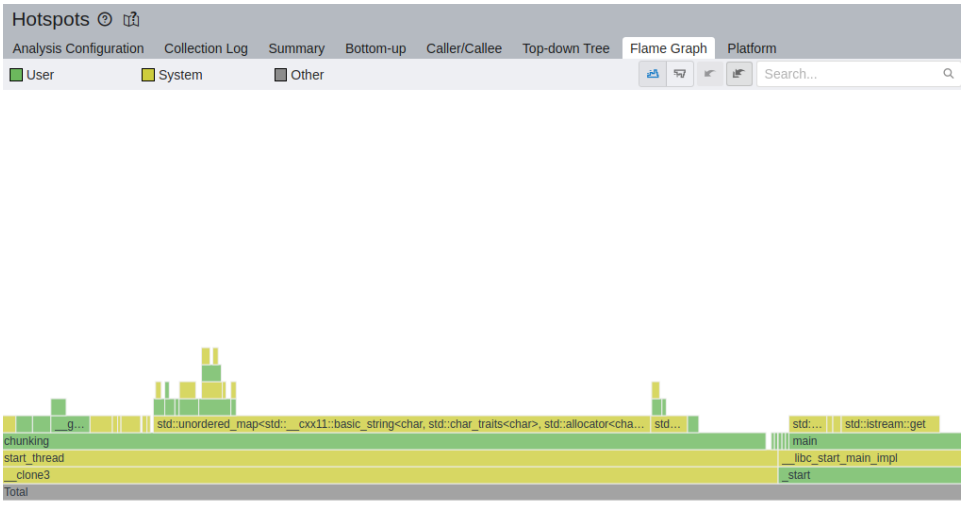


Figure 1 FLAME GRAPH

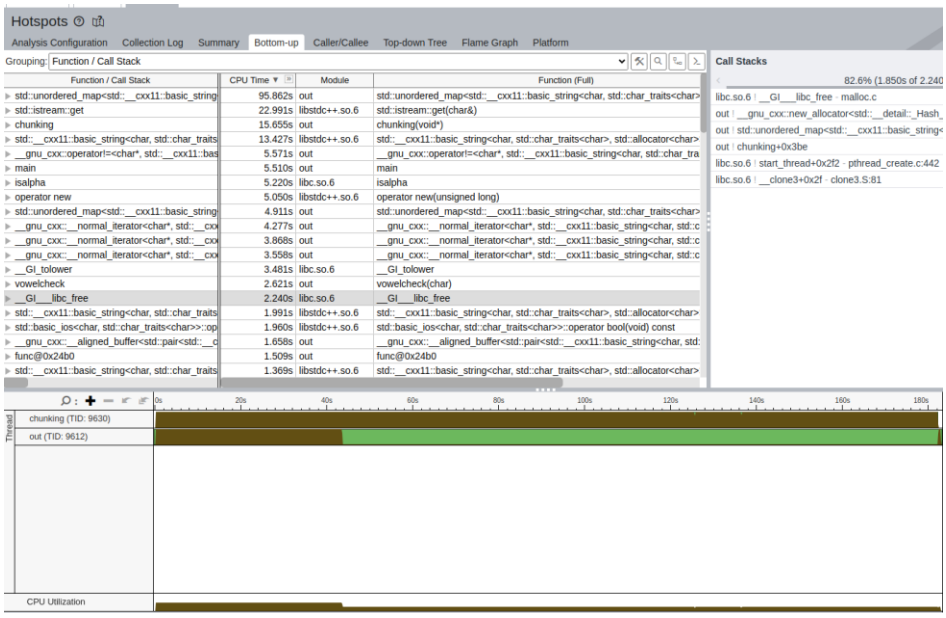


Figure 2 HOTSPOT ANALYSIS

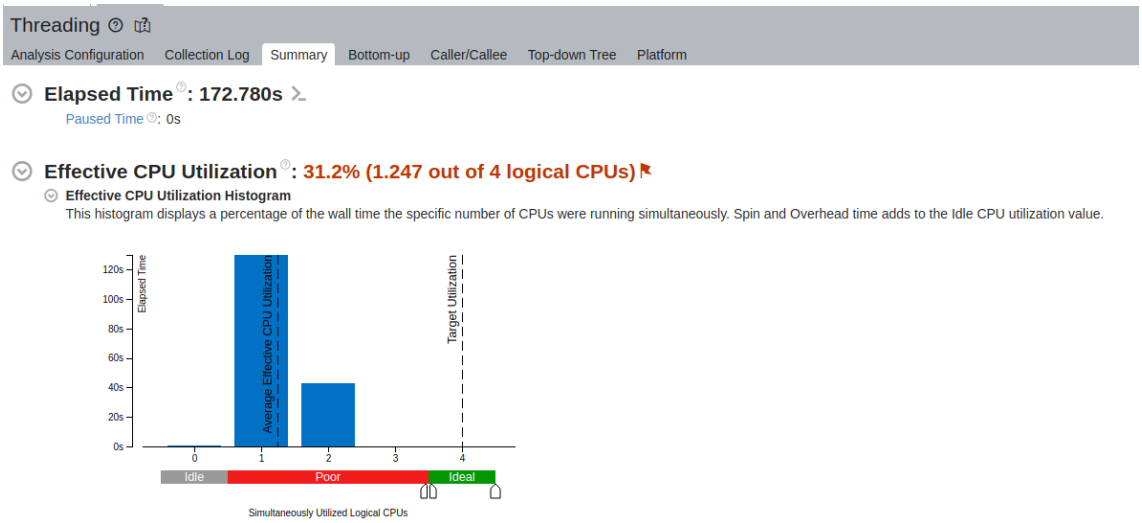


Figure 3 CPU UTILIZATION

2 Thread

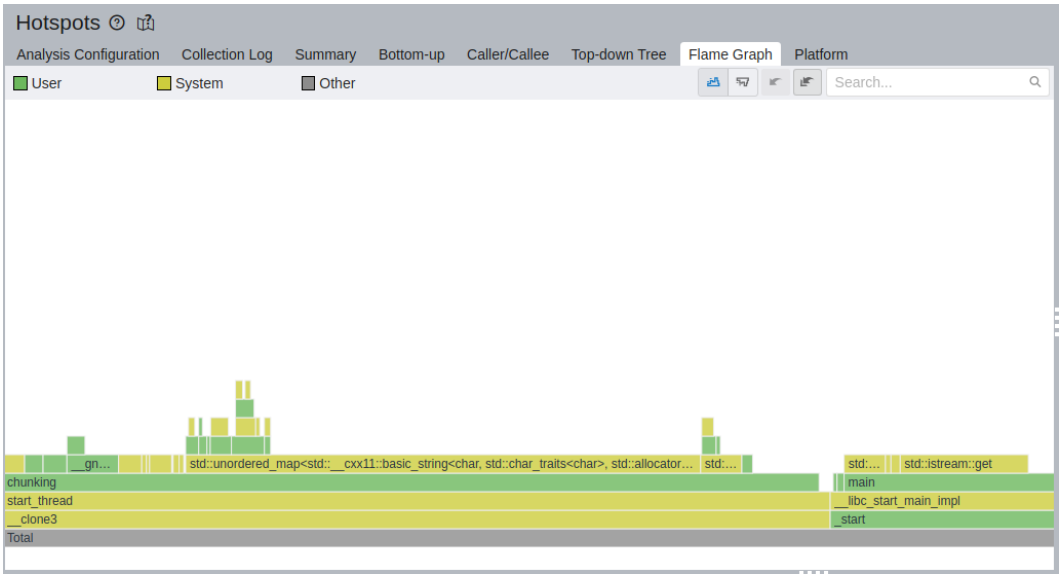


Figure 4 FLAME GRAPH

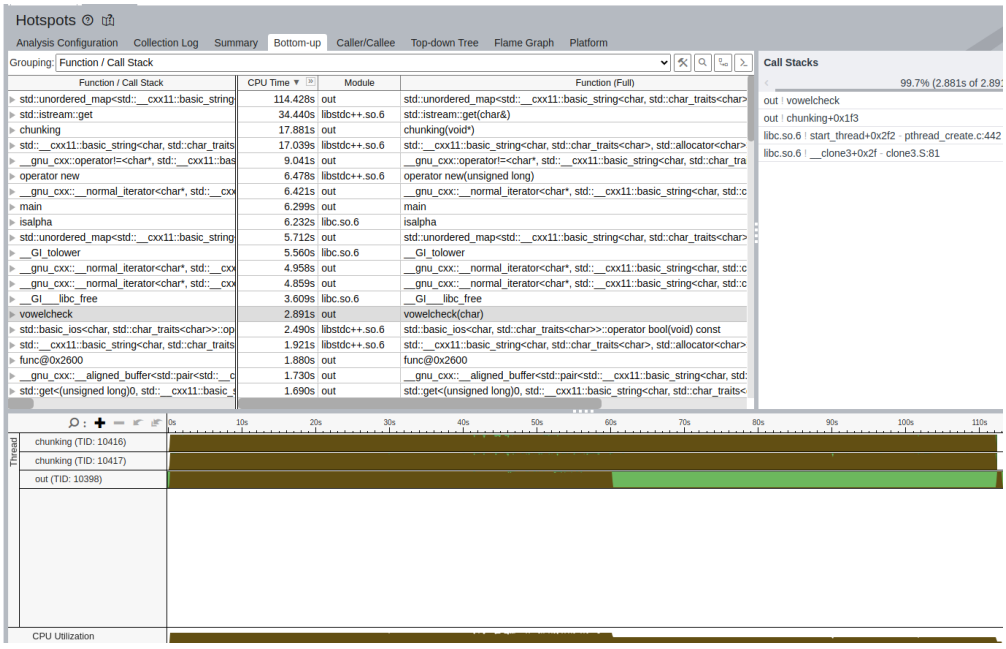


Figure 5 HOTSPOT ANALYSIS

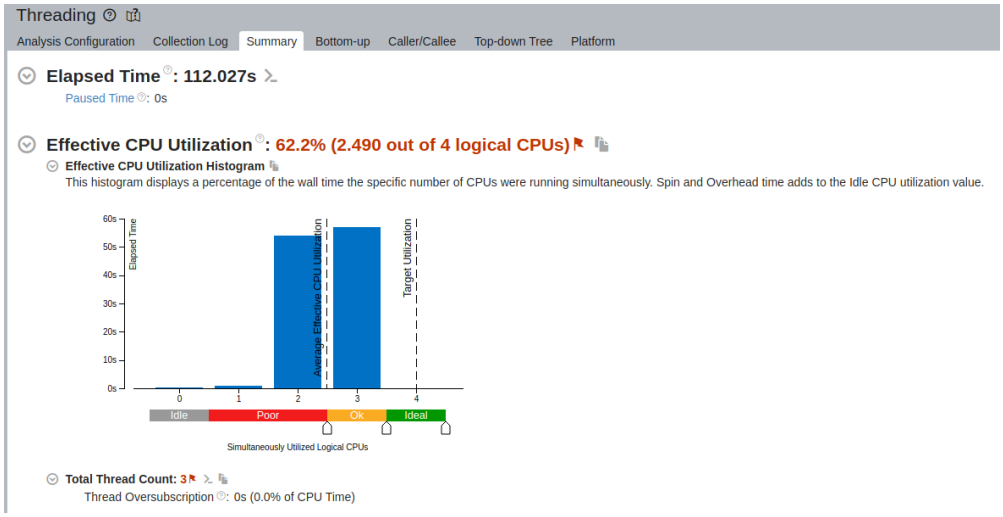


Figure 6 CPU UTILIZATION

4 Thread

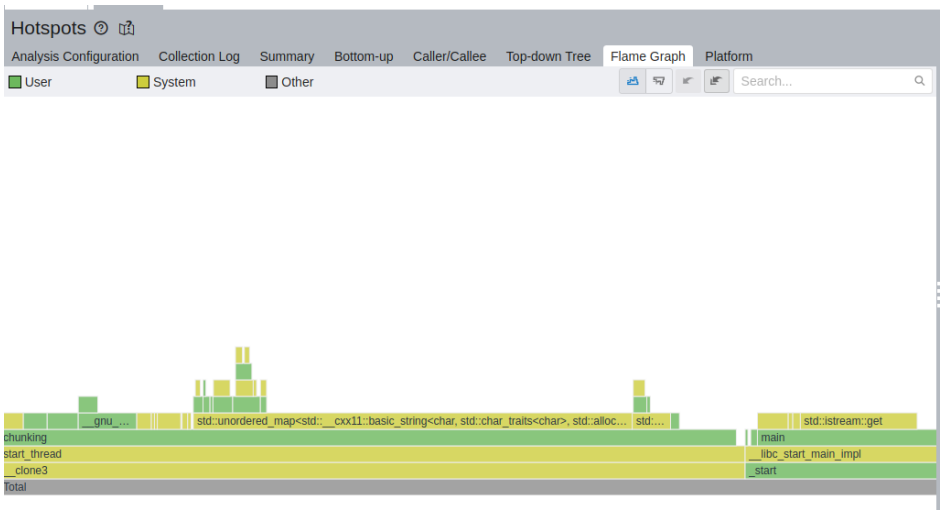


Figure 7 FLAME GRAPH

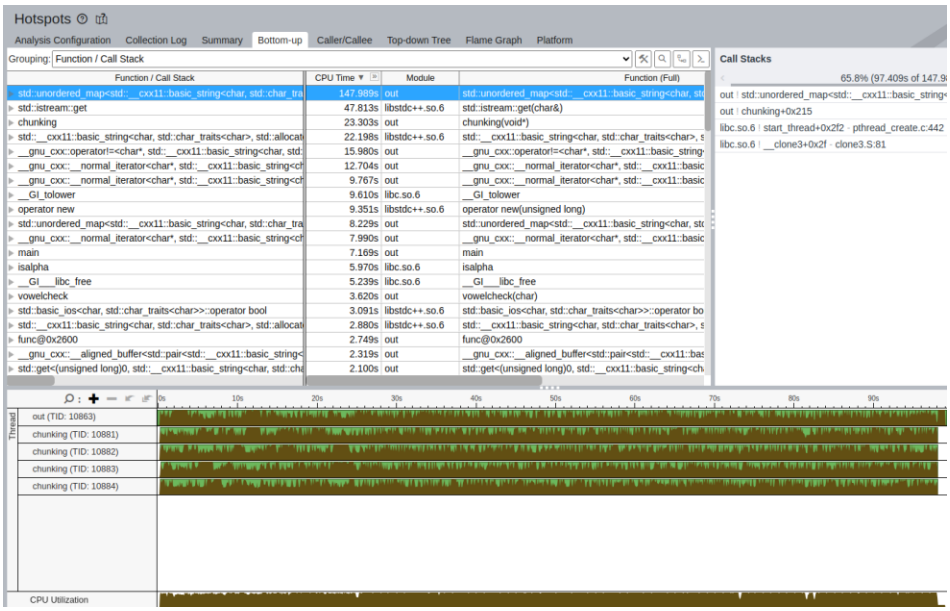


Figure 8 HOTSPOT ANALYSIS

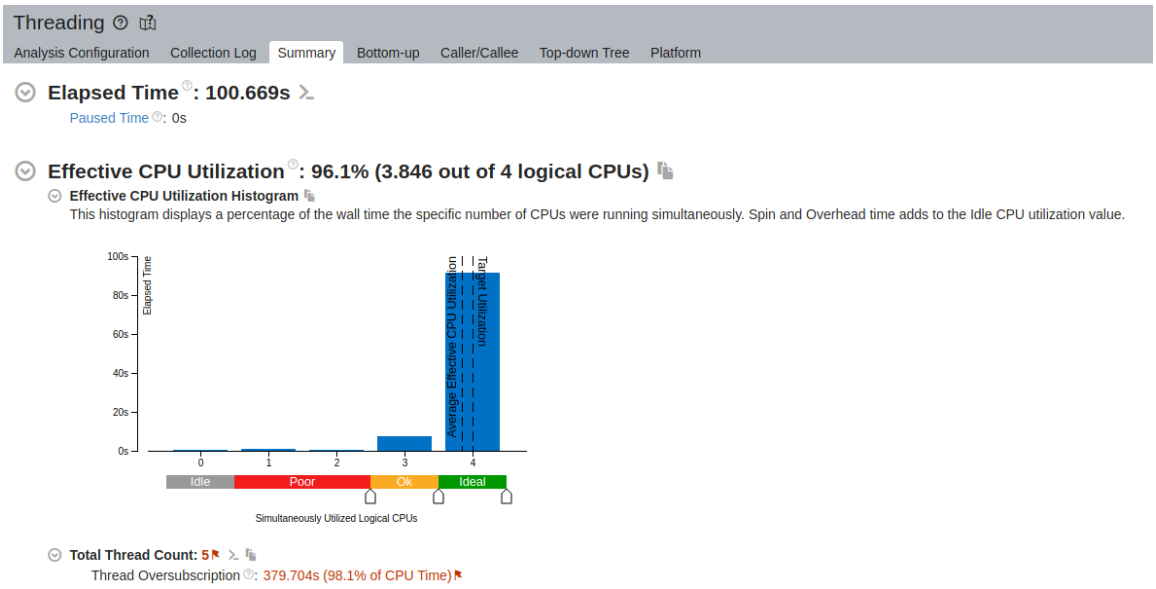


Figure 9 CPU UTILIZATION

8 Thread

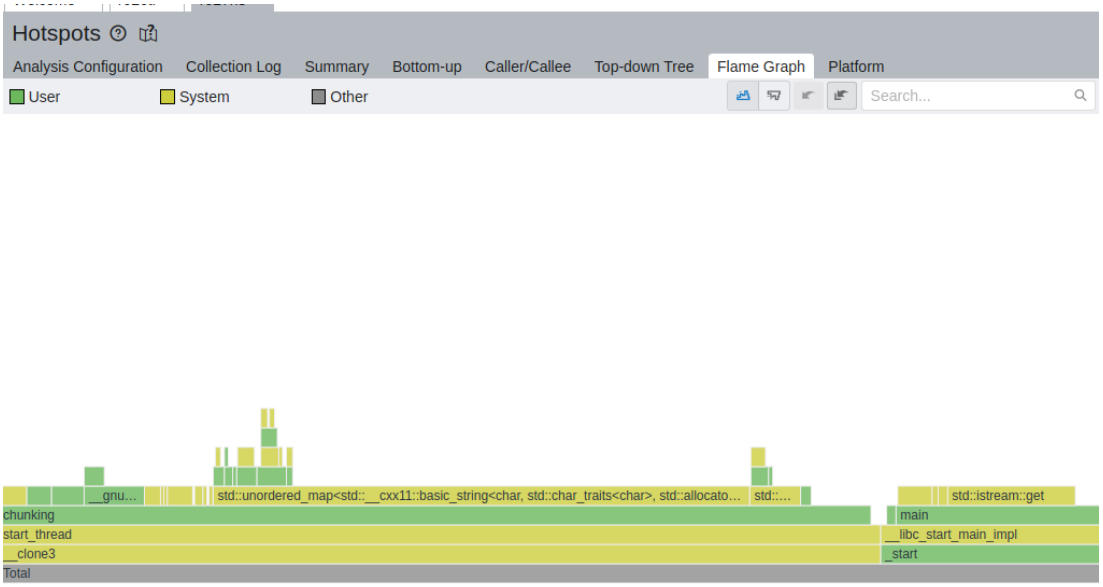


Figure 10 FLAME GRAPH

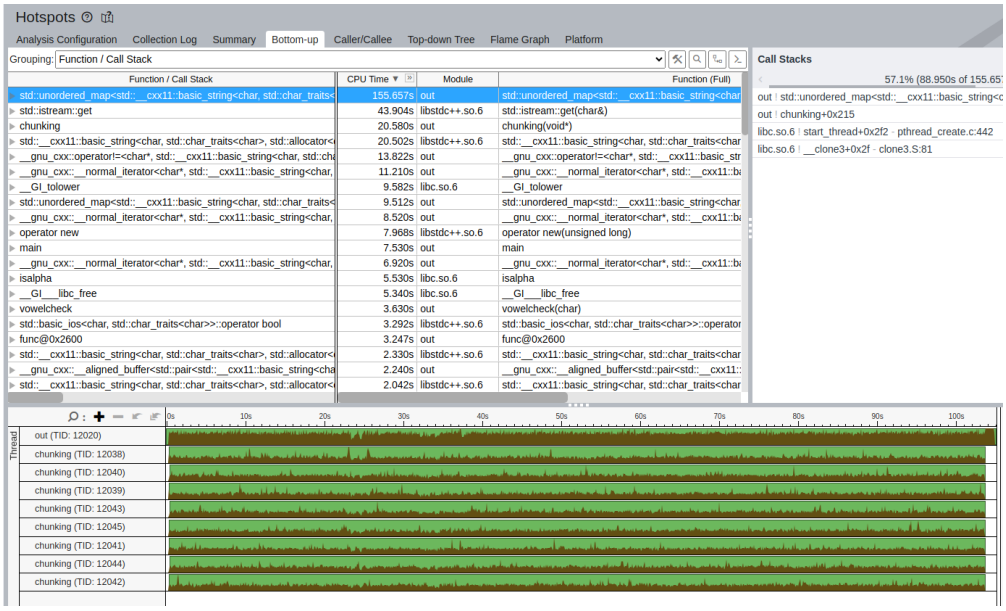


Figure 11 HOTSPOT ANALYSIS

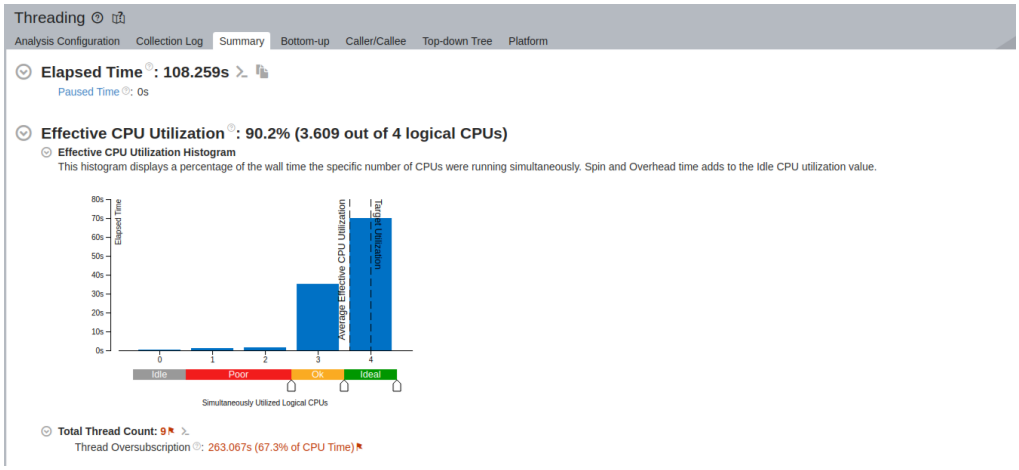


Figure 12 CPU UTILIZATION

Question 2 (With Affinity)

1 Thread

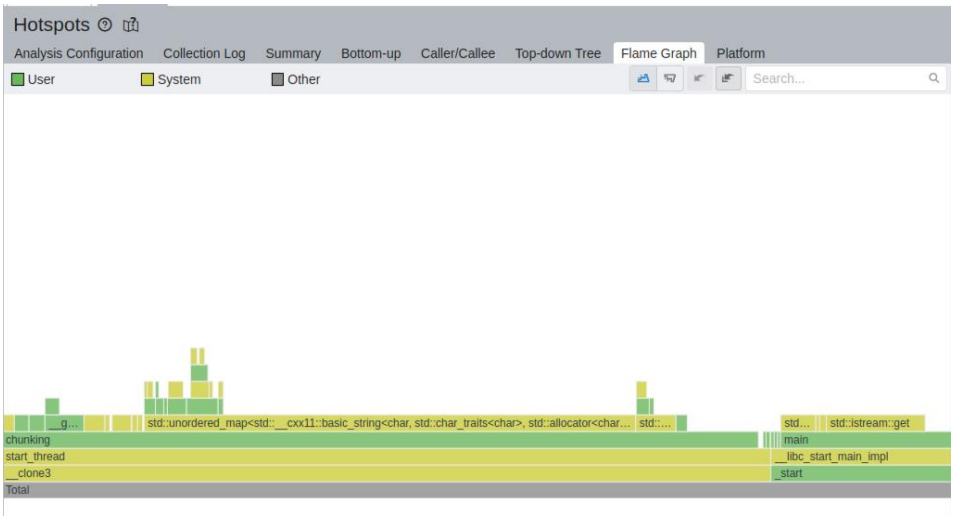


Figure 13 FLAME GRAPH

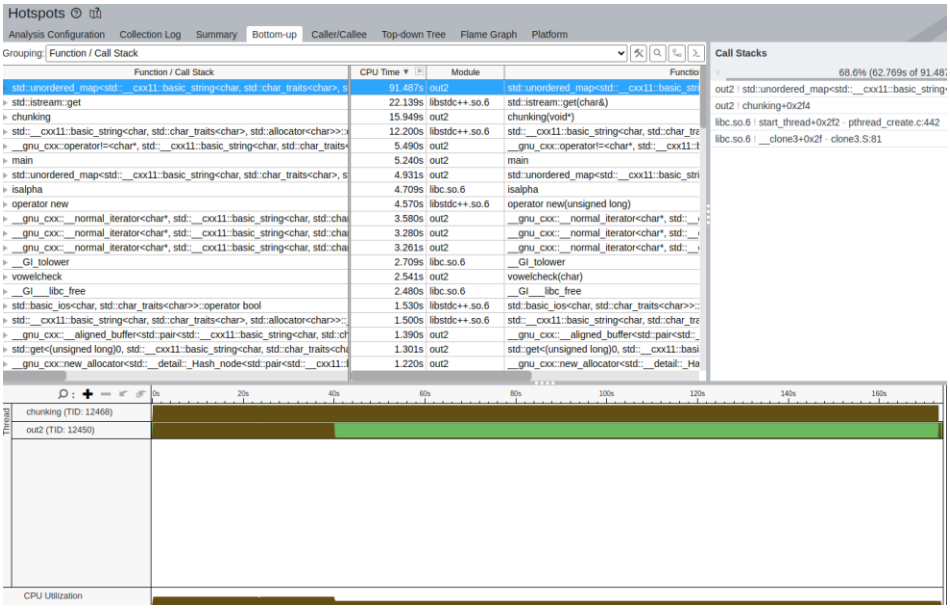


Figure 14 HOTSPOT ANALYSIS

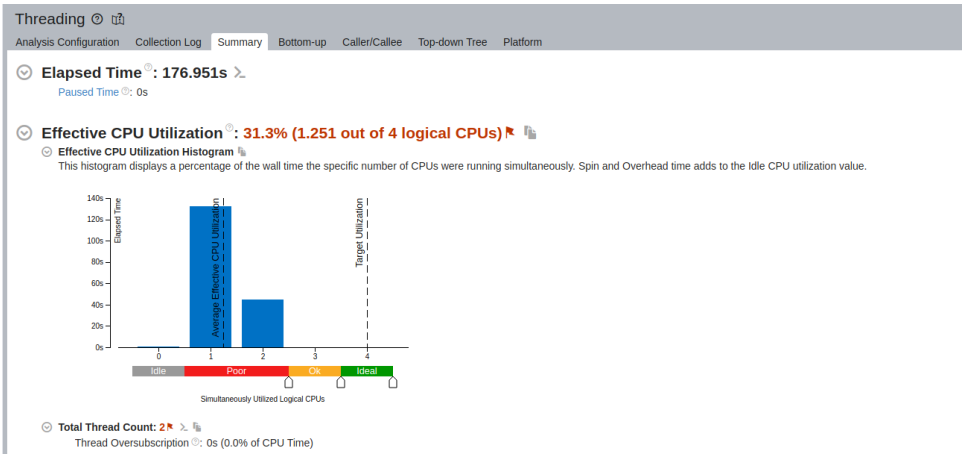


Figure 15 CPU UTILIZATION

2 Thread

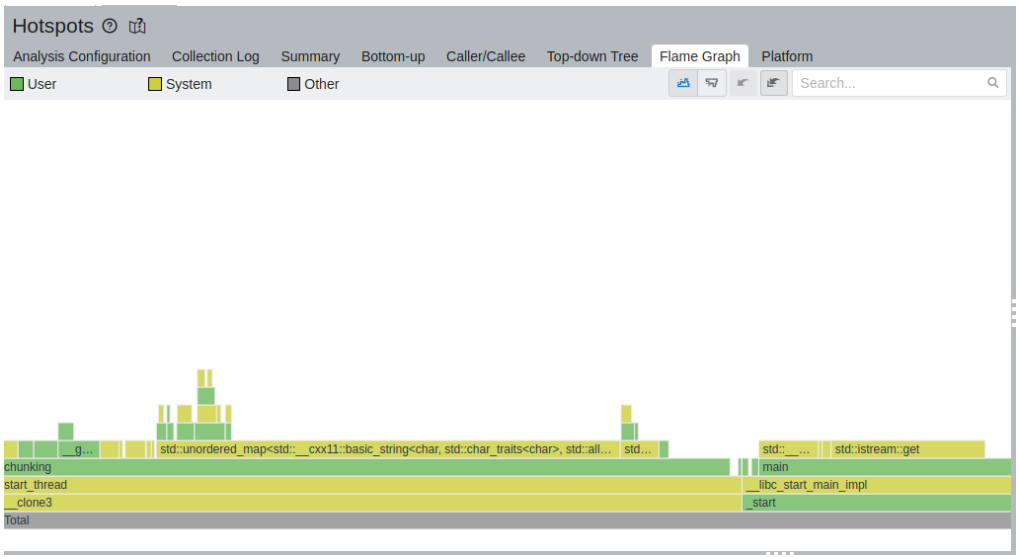


Figure 16 FLAME GRAPH

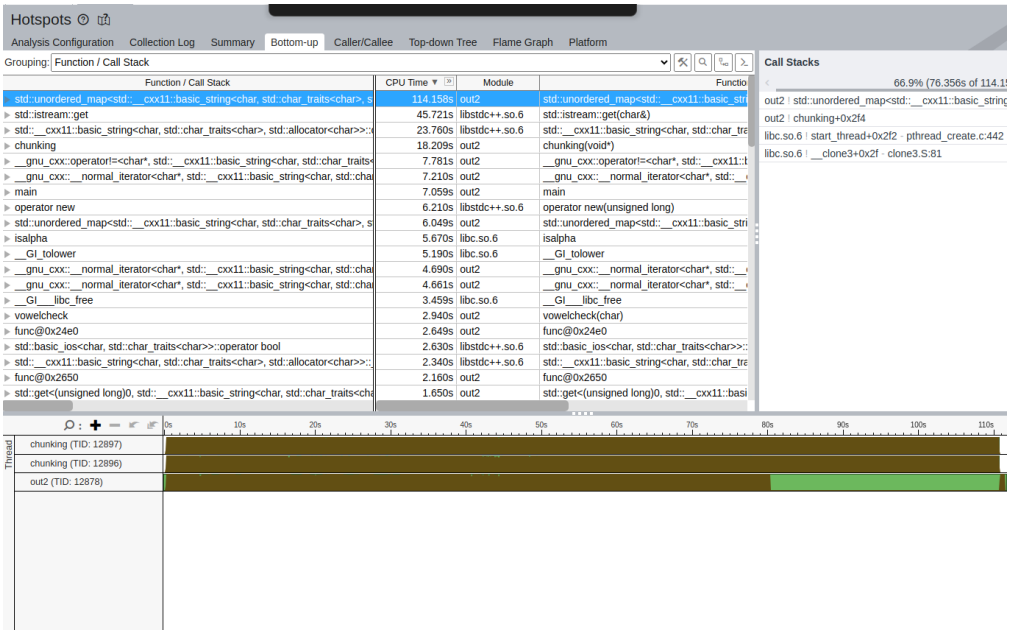


Figure 17 HOTSPOT ANALYSIS

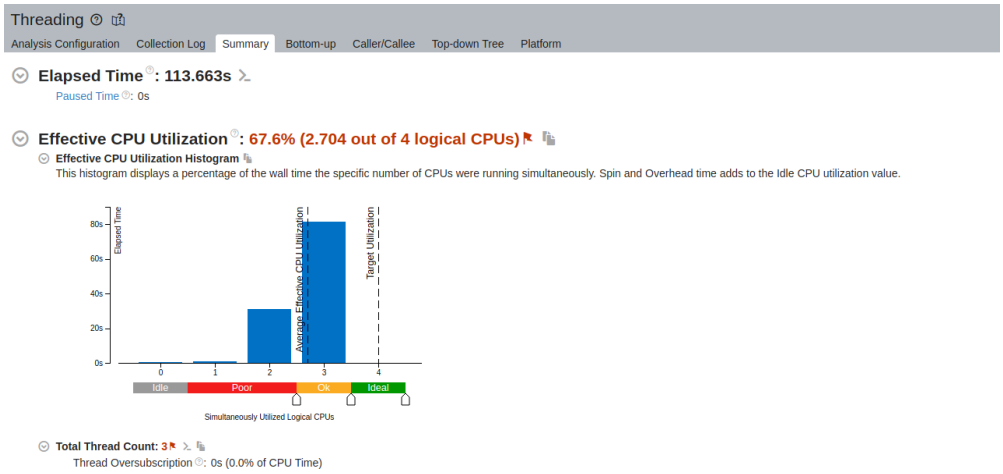


Figure 18 CPU UTILIZATION

4 Thread

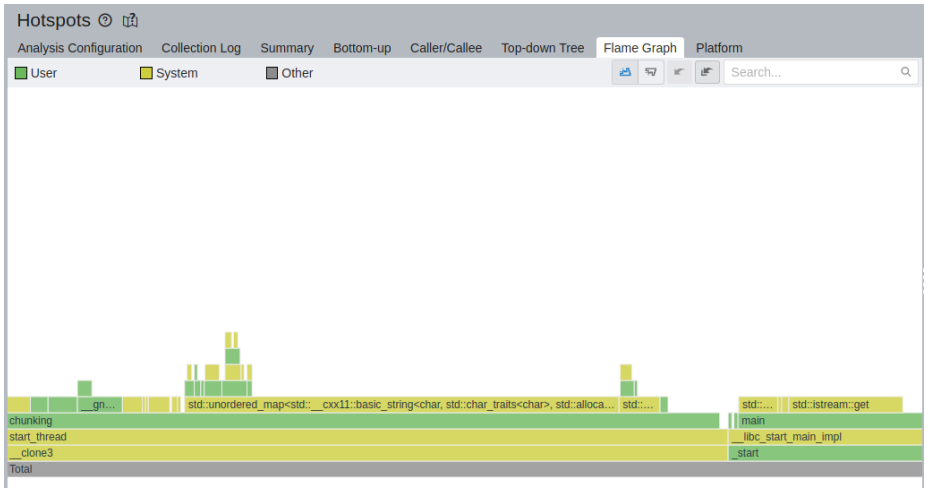


Figure 19 FLAME GRAPH

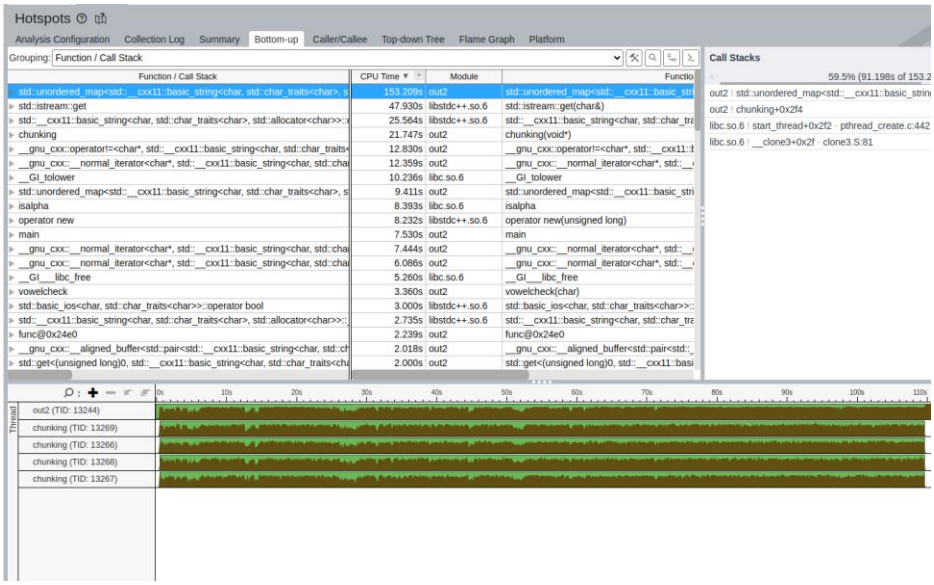


Figure 20 HOTSPOT ANALYSIS

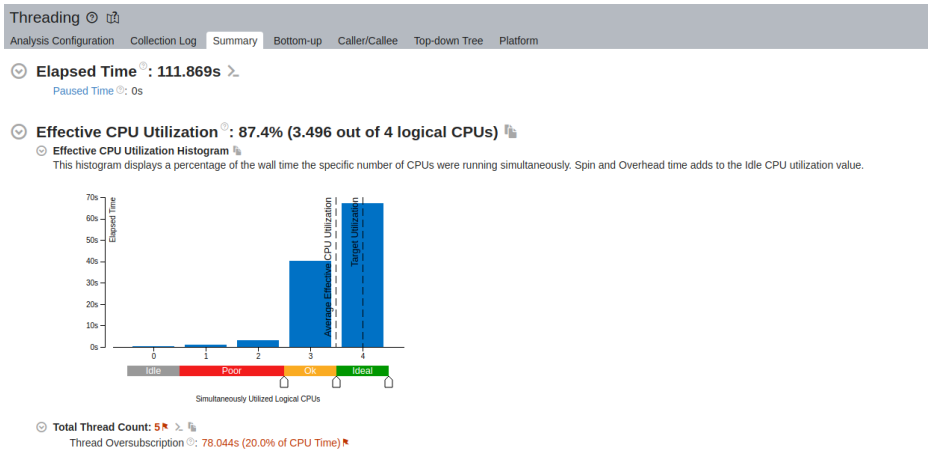


Figure 21 CPU UTILIZATION

8 Thread

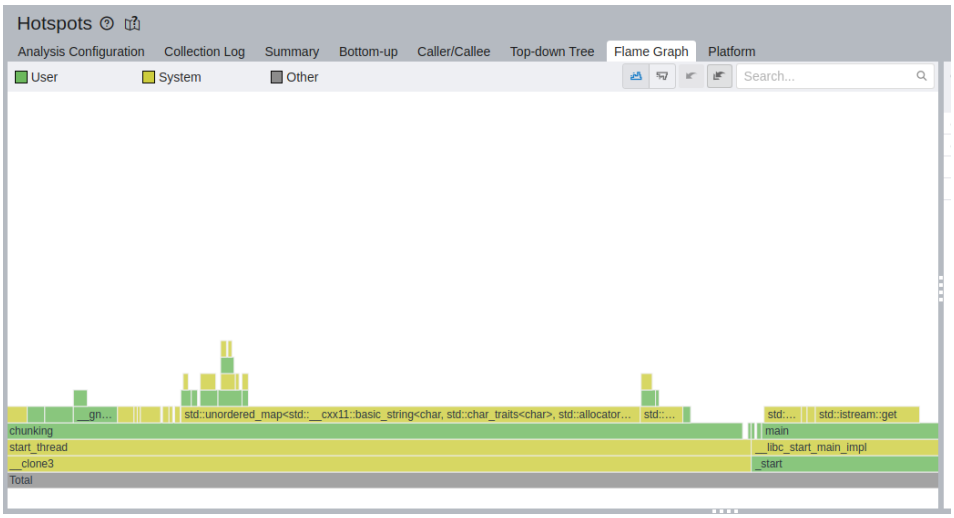


Figure 22 FLAME GRAPH

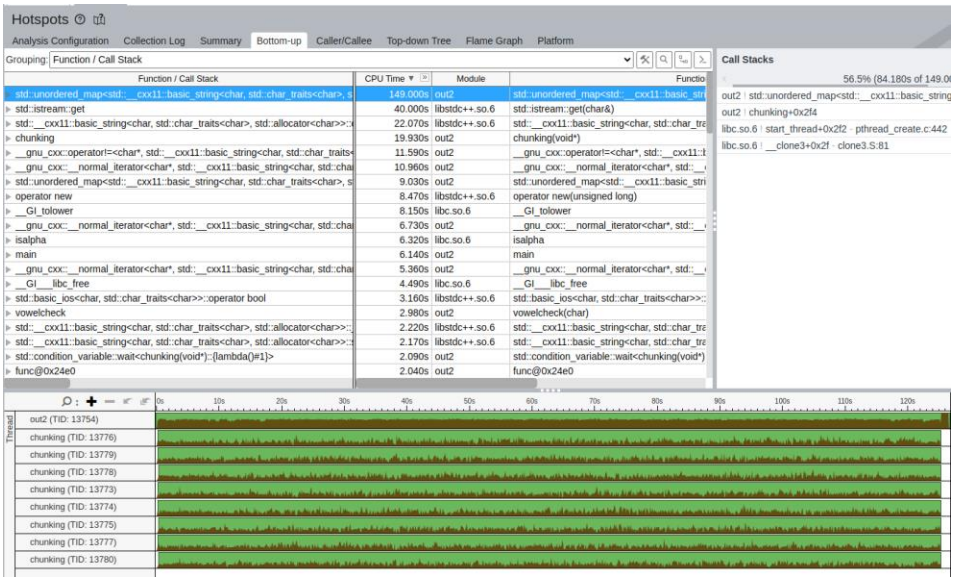


Figure 23 HOTSPOT ANALYSIS

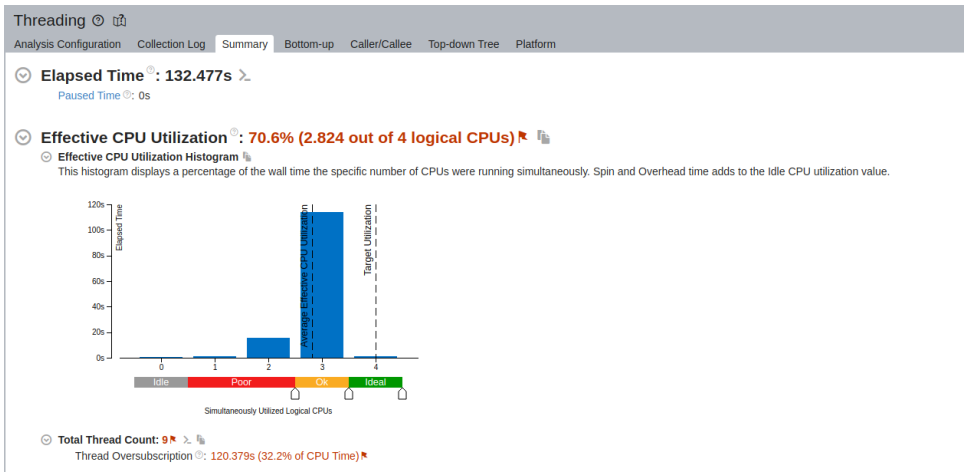


Figure 24 CPU UTILIZATION

Question 3 (Without Affinity)

1 Thread

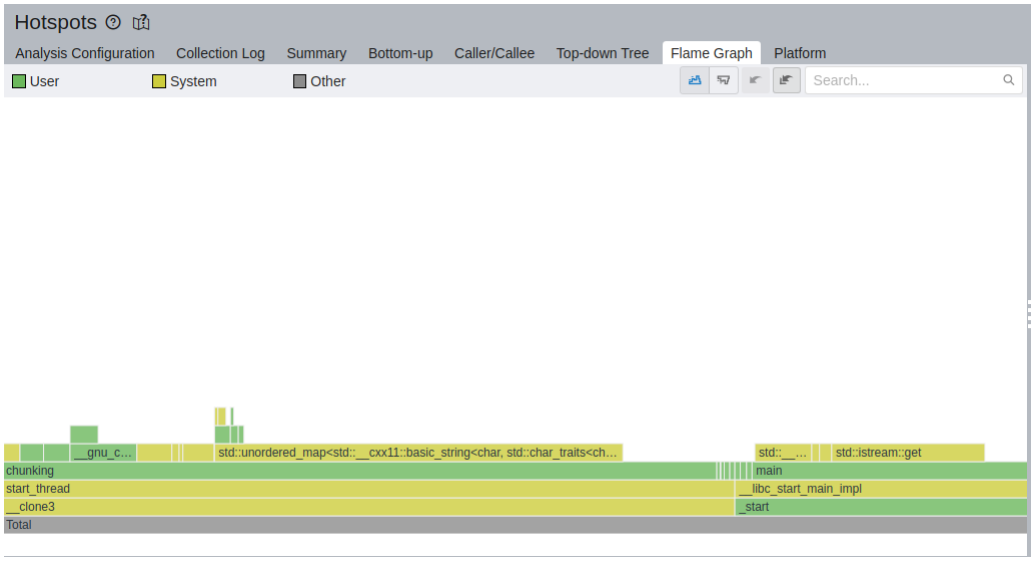


Figure 25 FLAME GRAPH

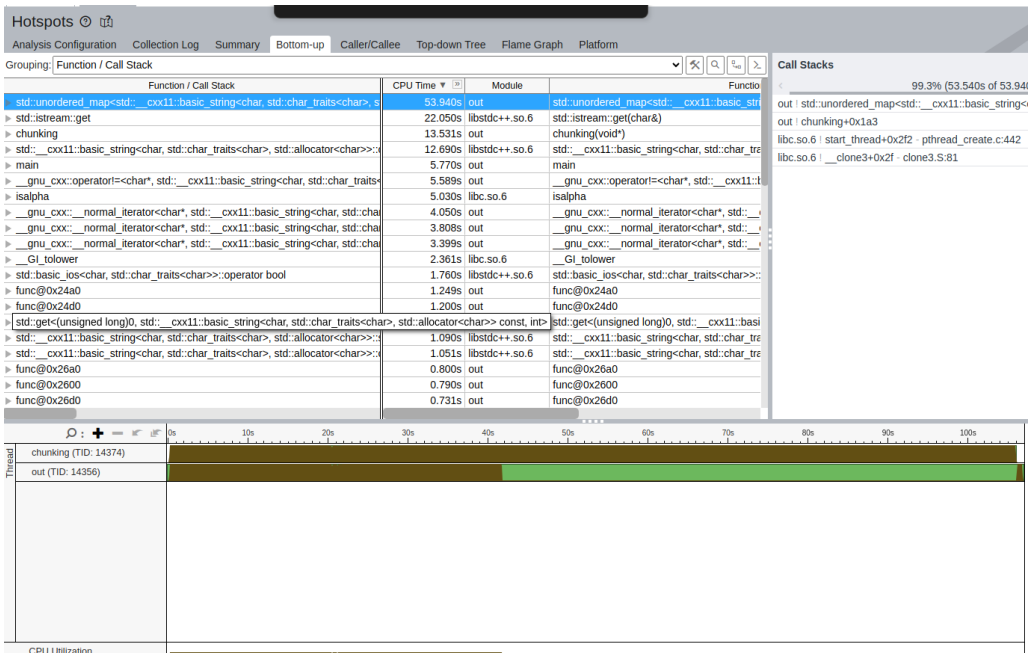


Figure 26 HOTSPOT ANALYSIS

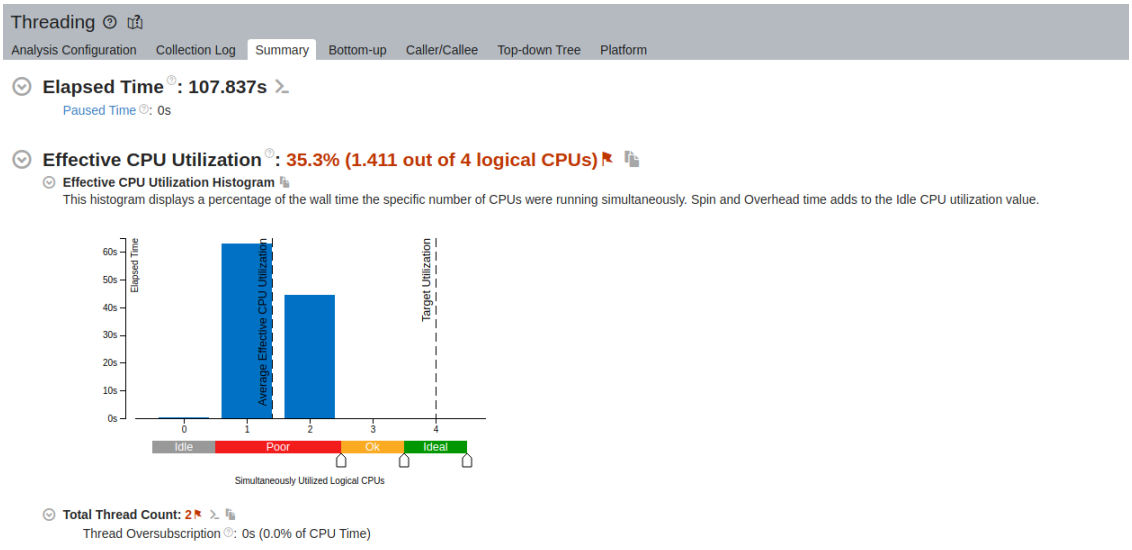


Figure 27 CPU UTILIZATION

2 Thread

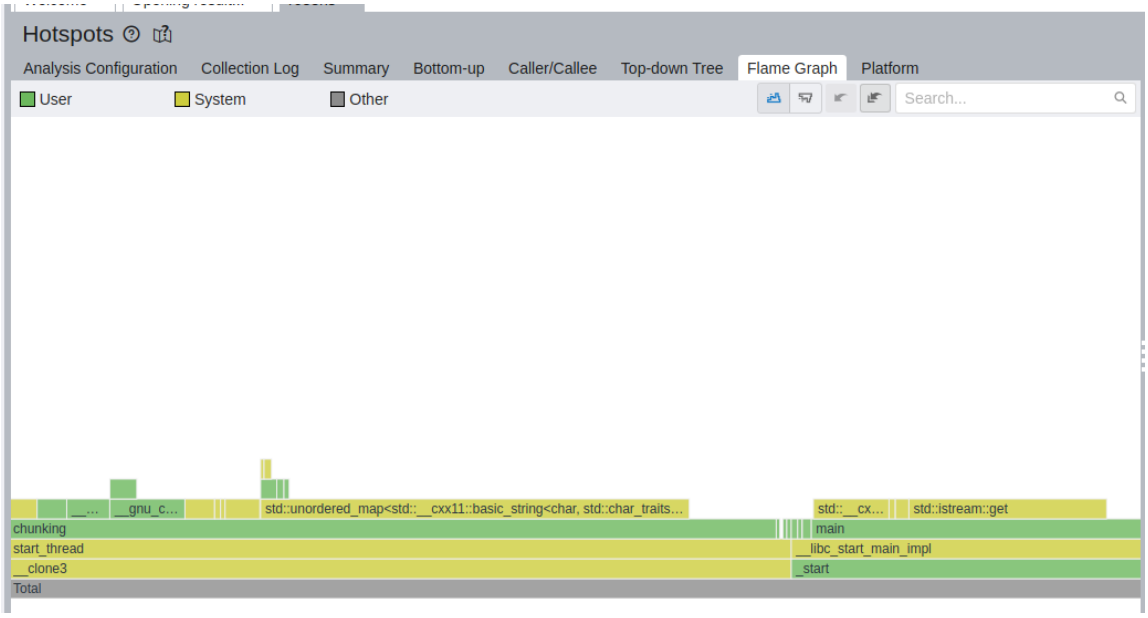


Figure 28 FLAME GRAPH

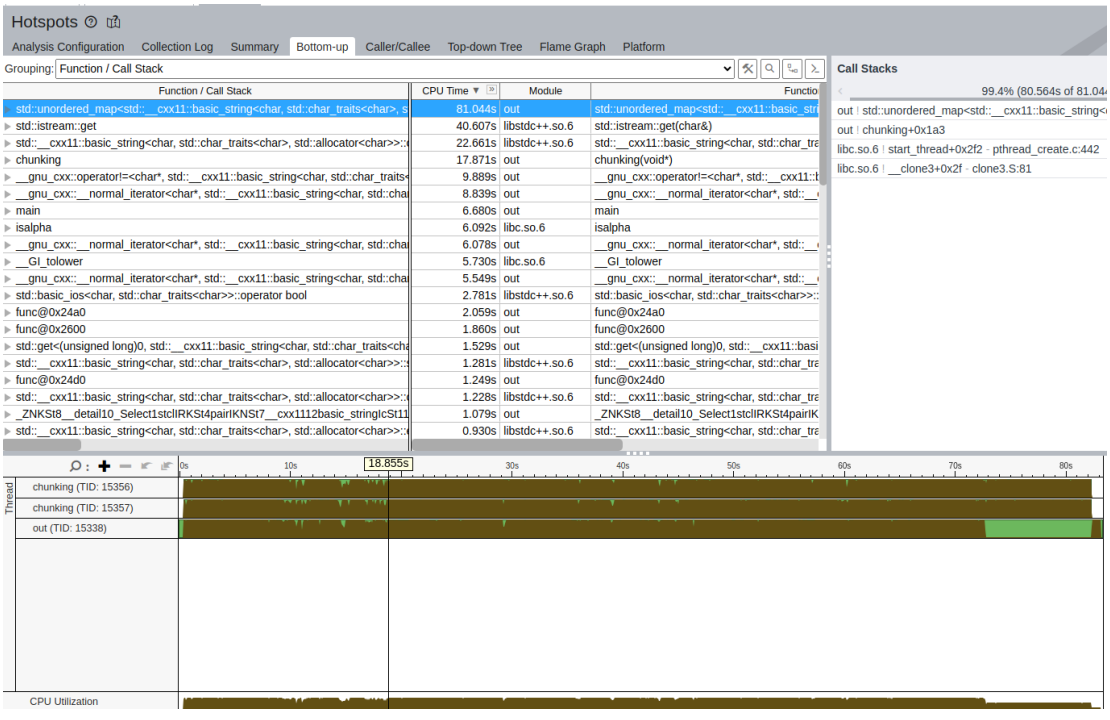


Figure 29 HOTSPOT ANALYSIS

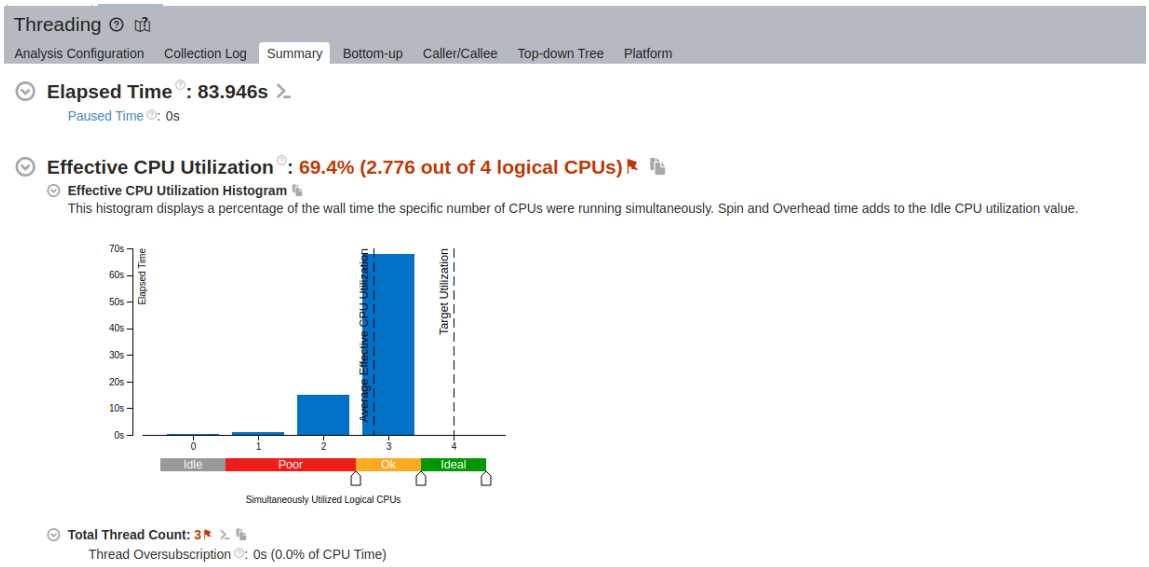


Figure 30 CPU UTILIZATION

4 Thread

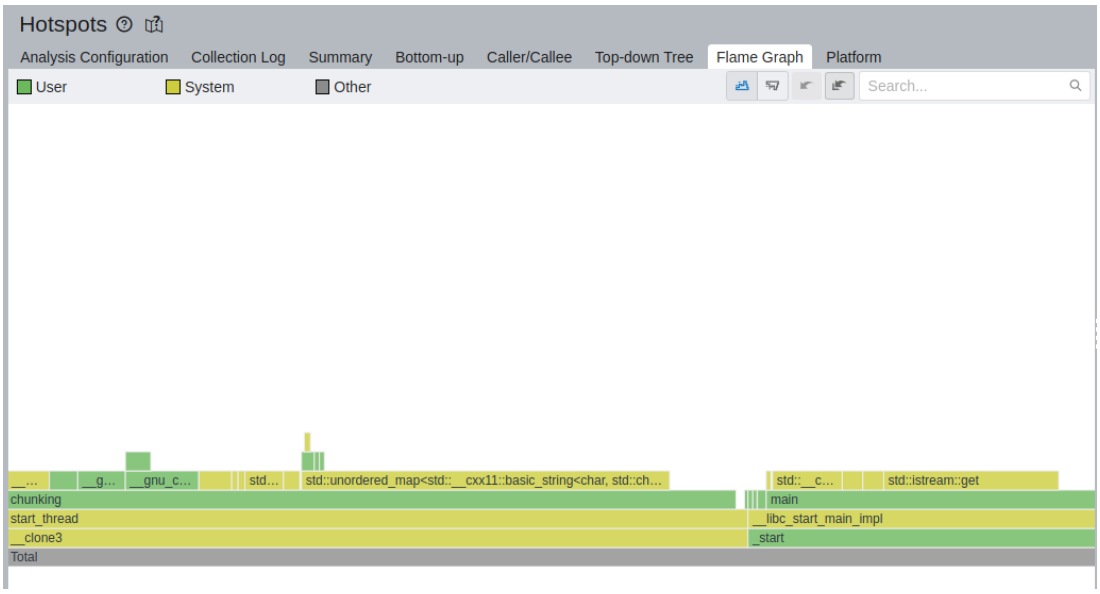


Figure 31 FLAME GRAPH

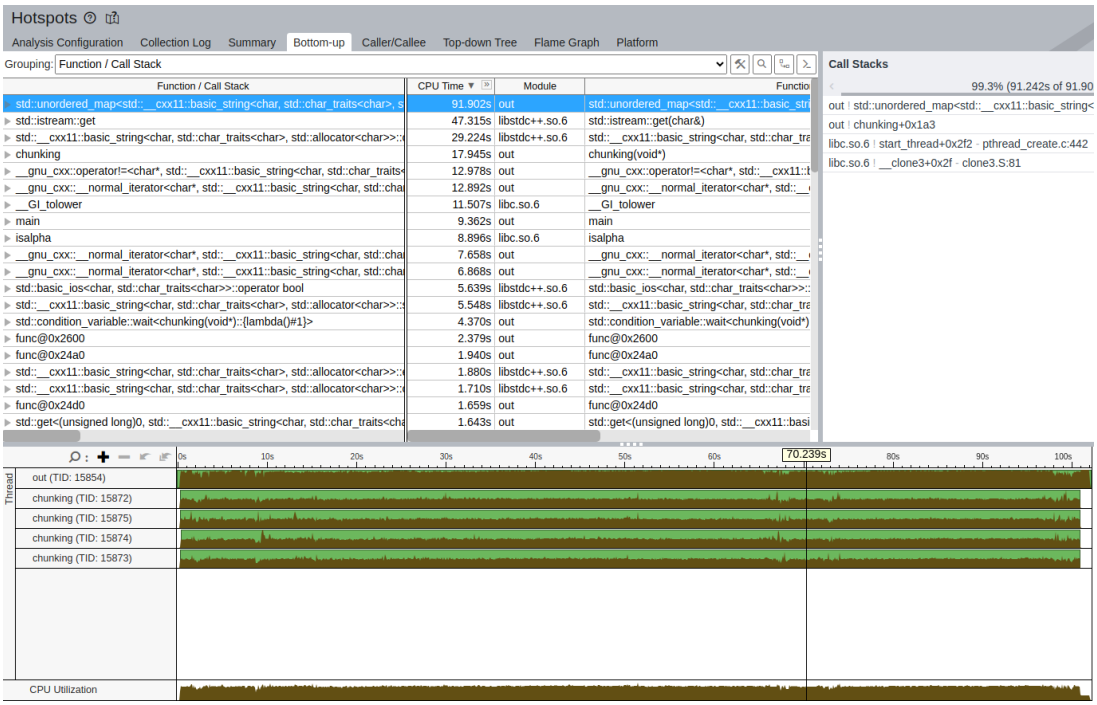


Figure 32 HOTSPOT ANALYSIS

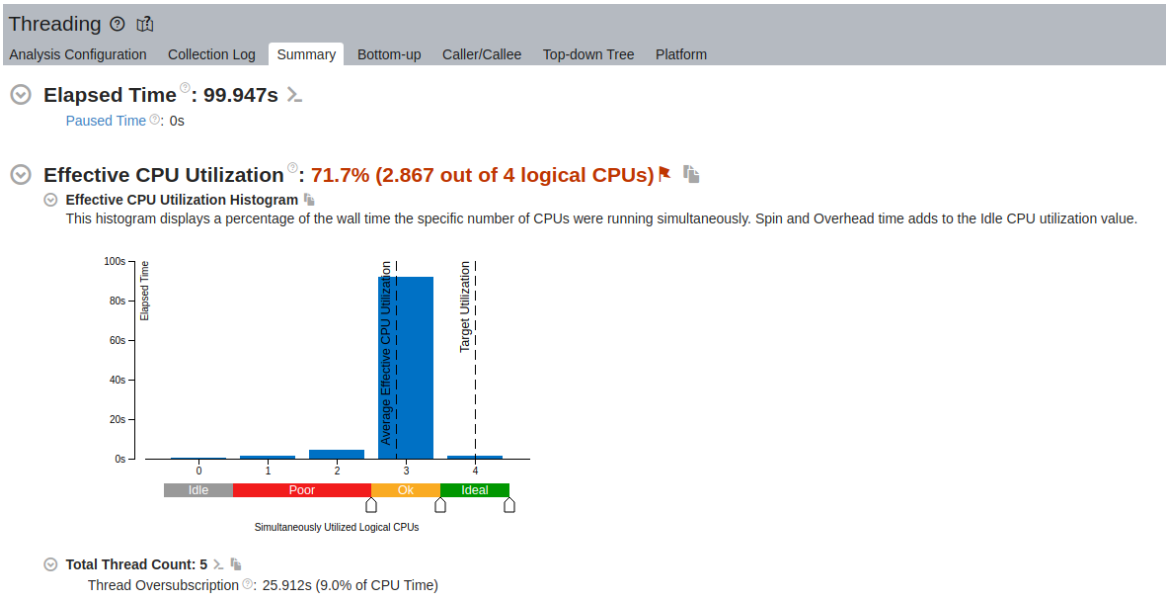


Figure 33 CPU UTILIZATION

8 Thread

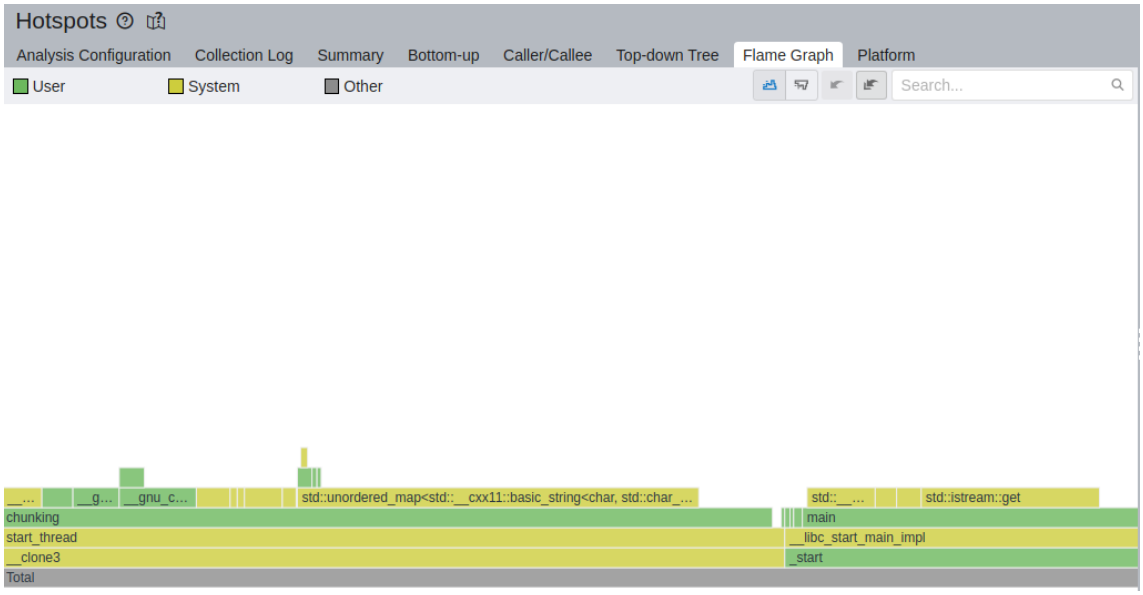


Figure 34 FLAME GRAPH

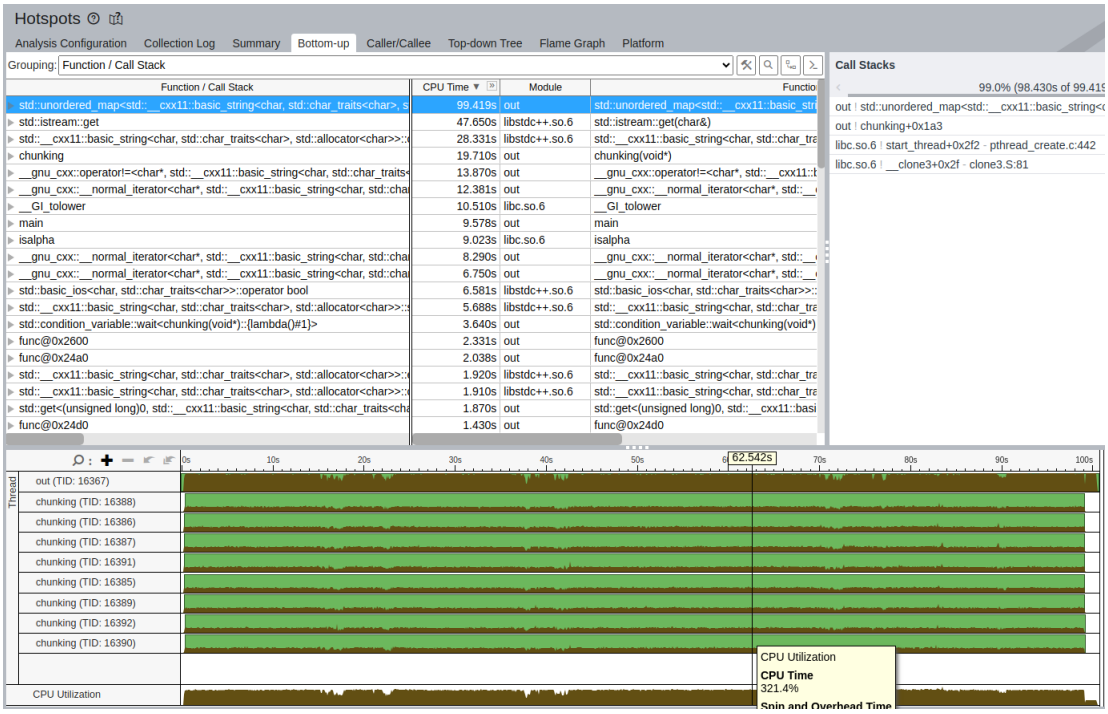


Figure 35 HOTSPOT ANALYSIS

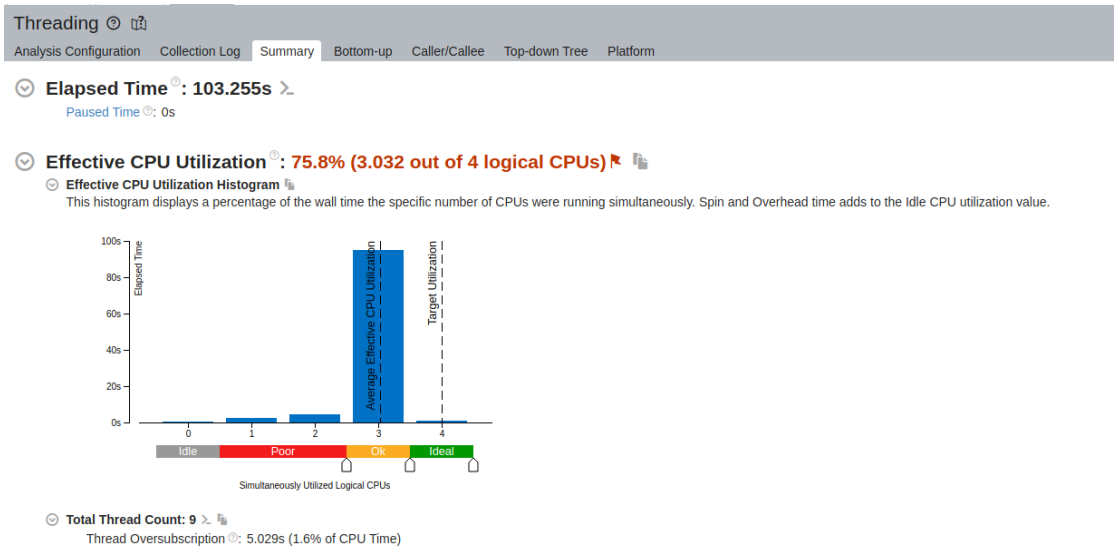


Figure 36 CPU UTILIZATION

Question 3 (With Affinity)

1 Thread

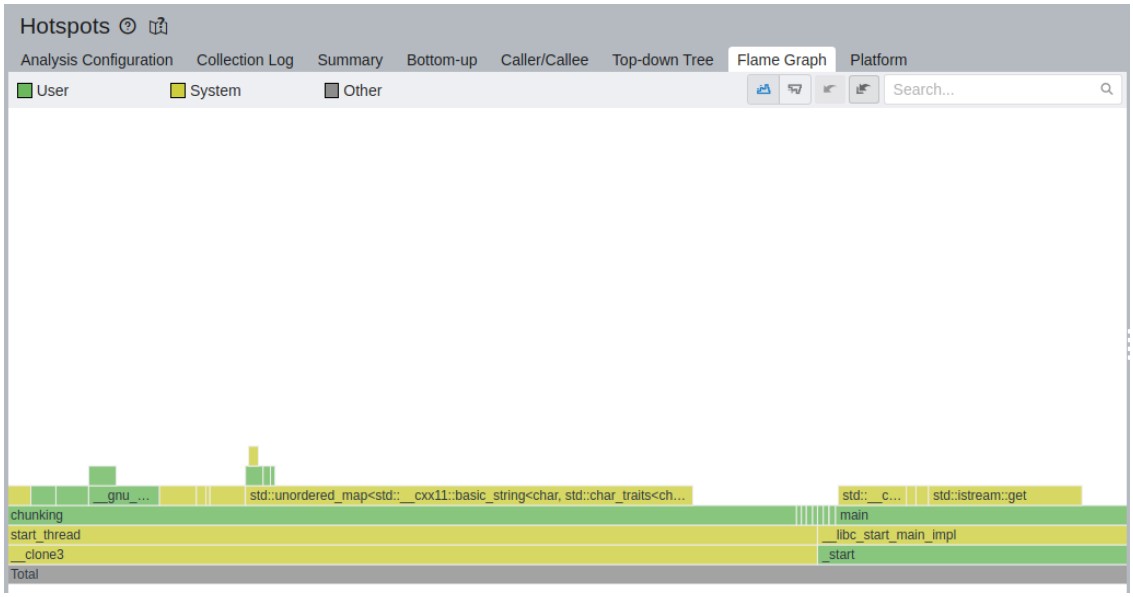


Figure 37 FLAME GRAPH

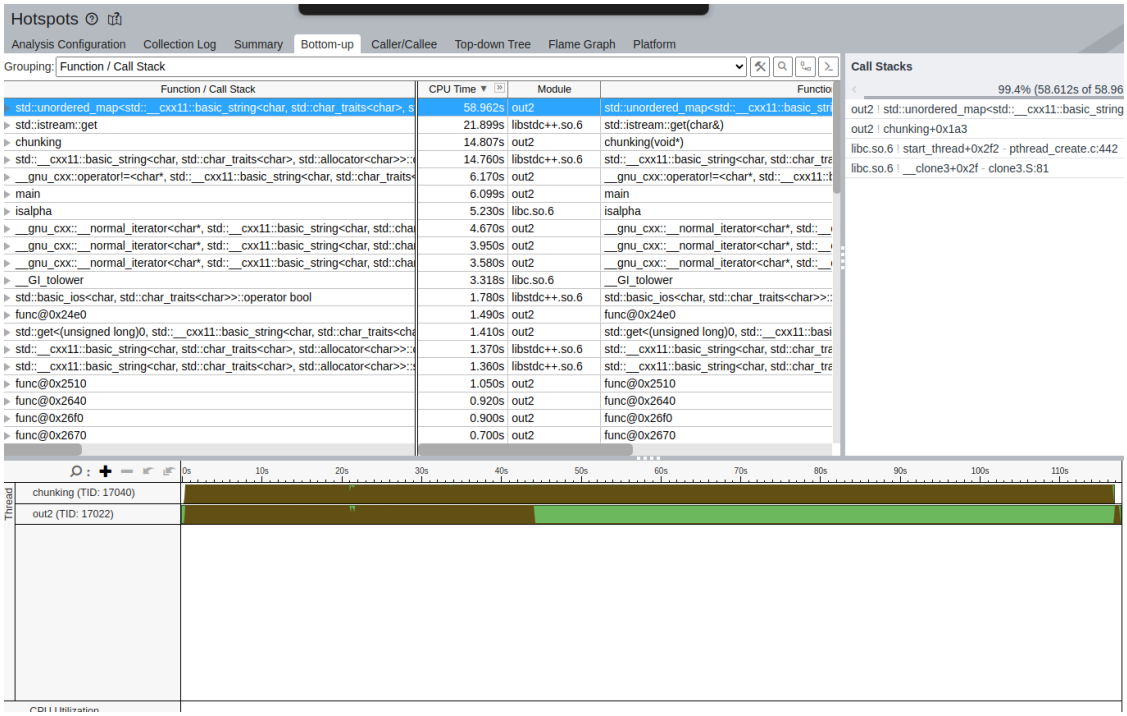


Figure 38 HOTSPOT ANALYSIS

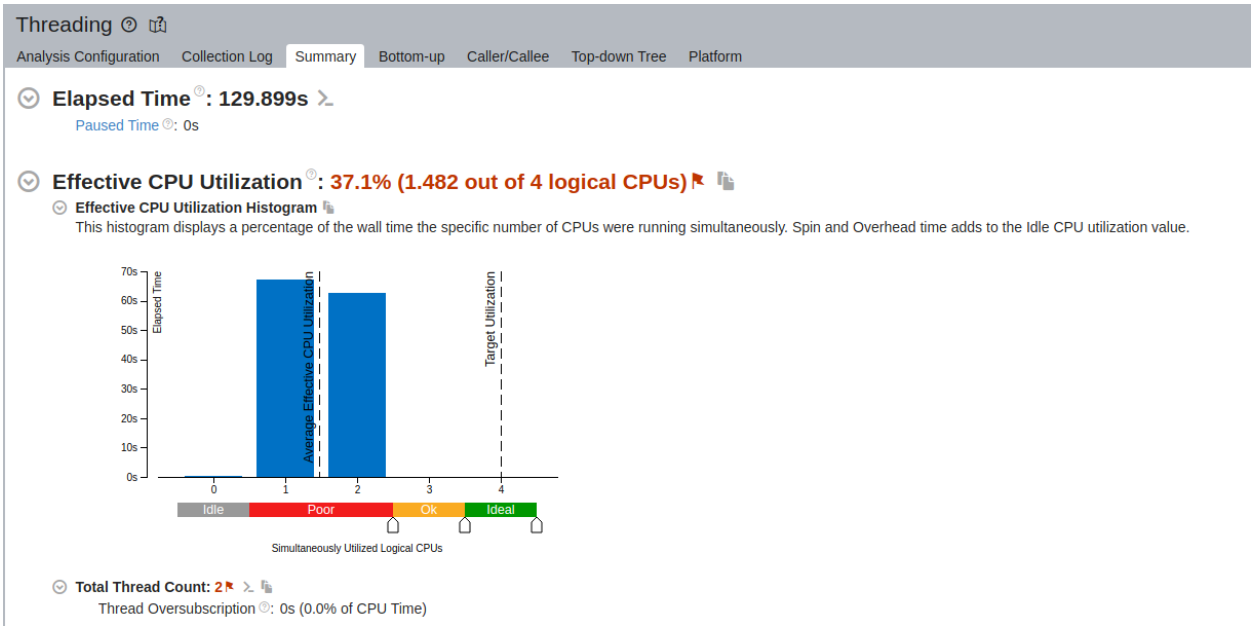


Figure 39 CPU UTILIZATION

2 Thread

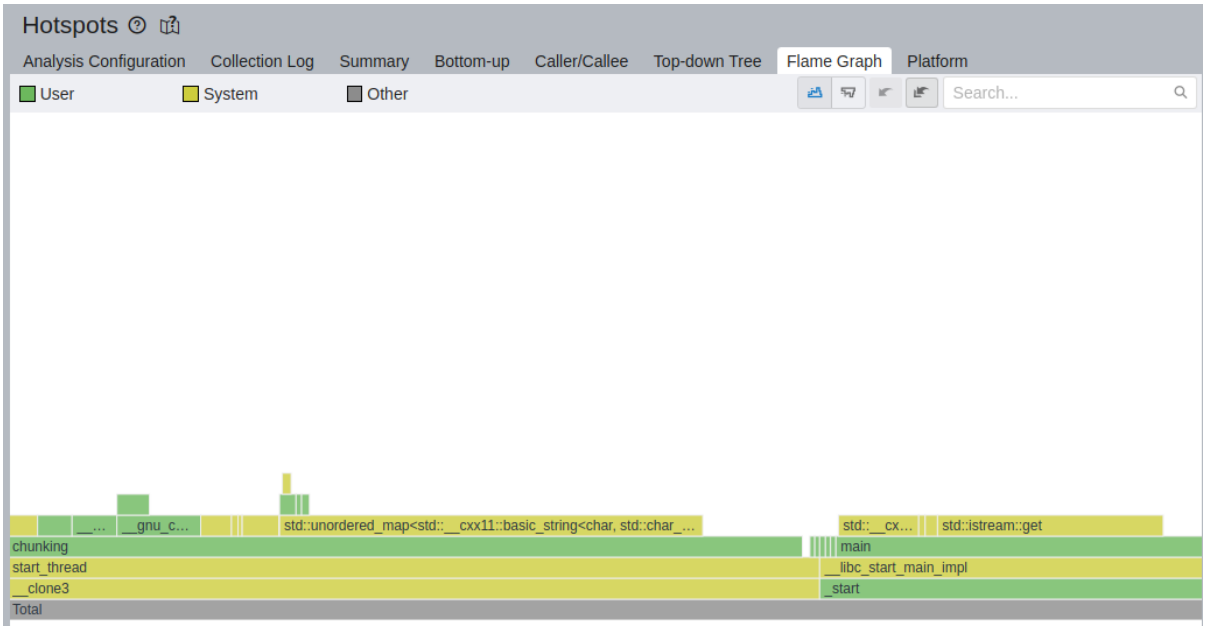


Figure 40 FLAME GRAPH

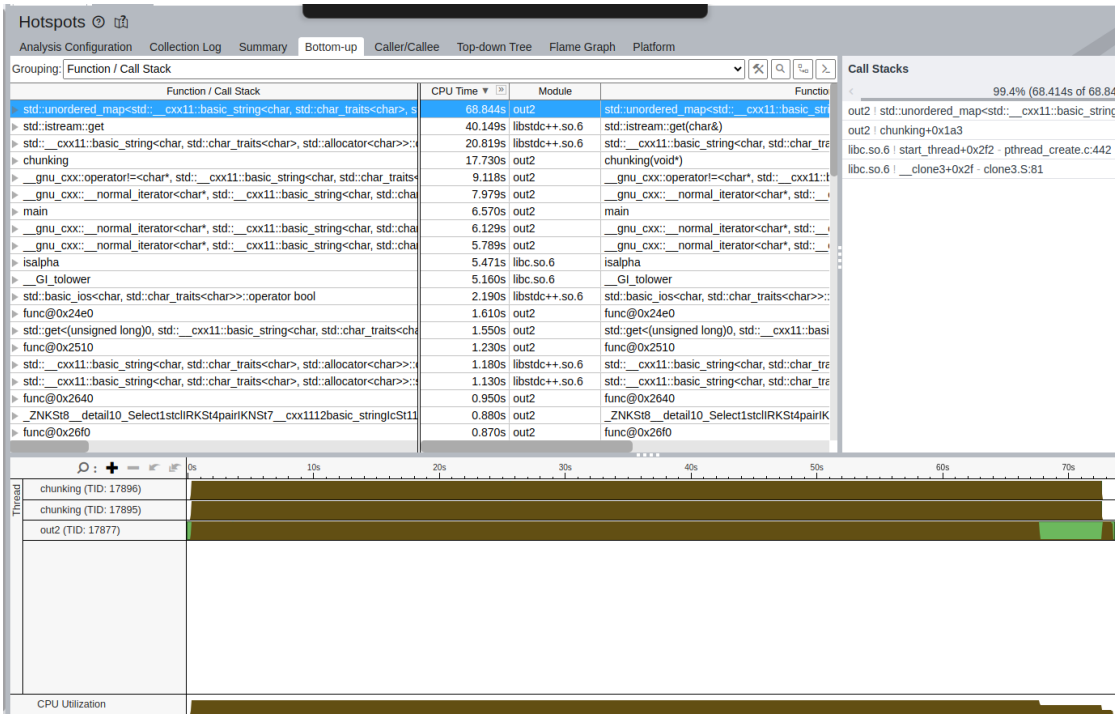


Figure 41 HOTSPOT ANALYSIS

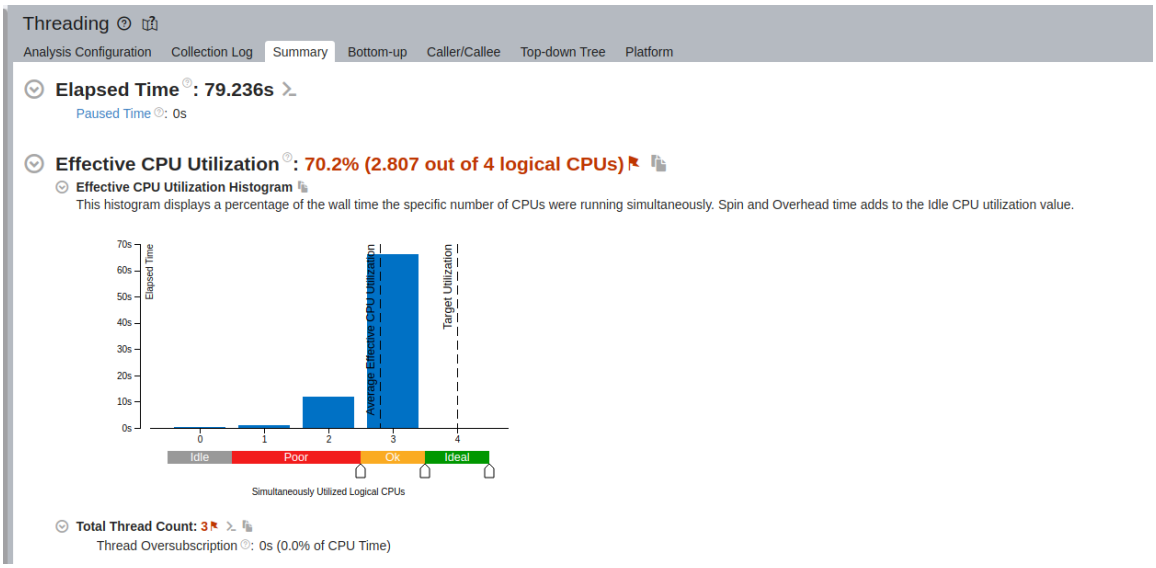


Figure 42 CPU UTILIZATION

4 Thread

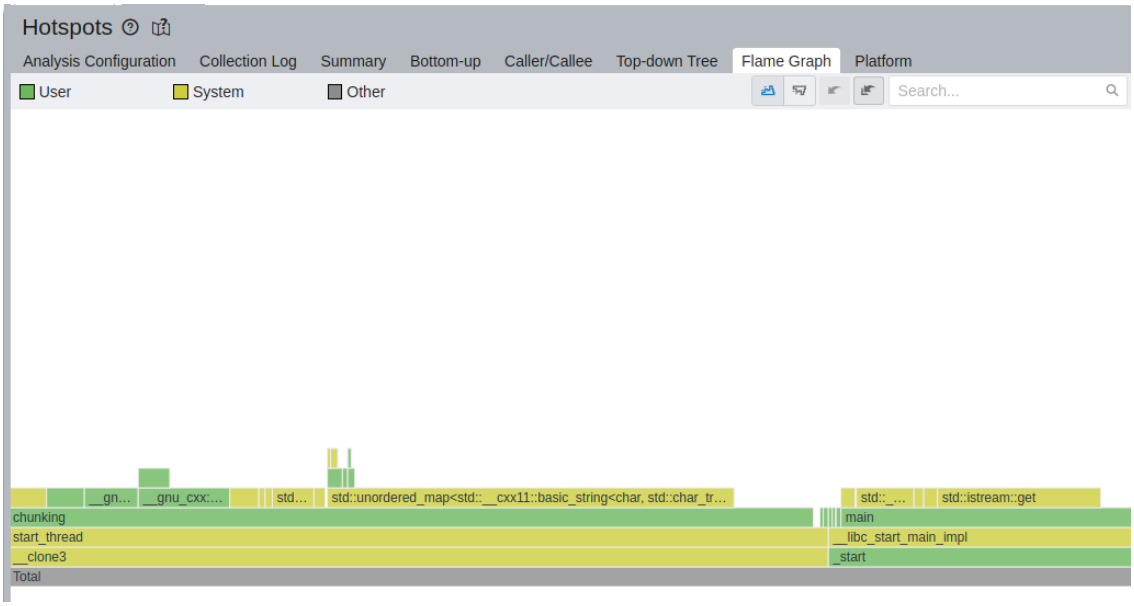


Figure 43 FLAME GRAPH

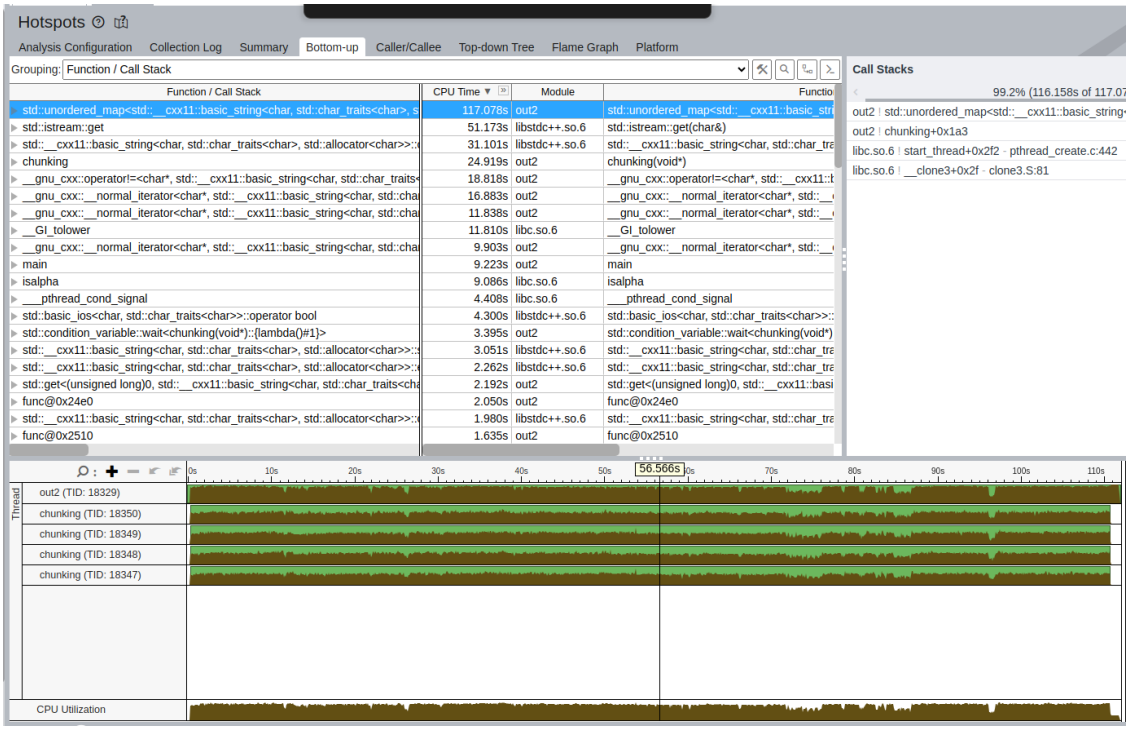


Figure 44 HOTSPOT ANALYSIS

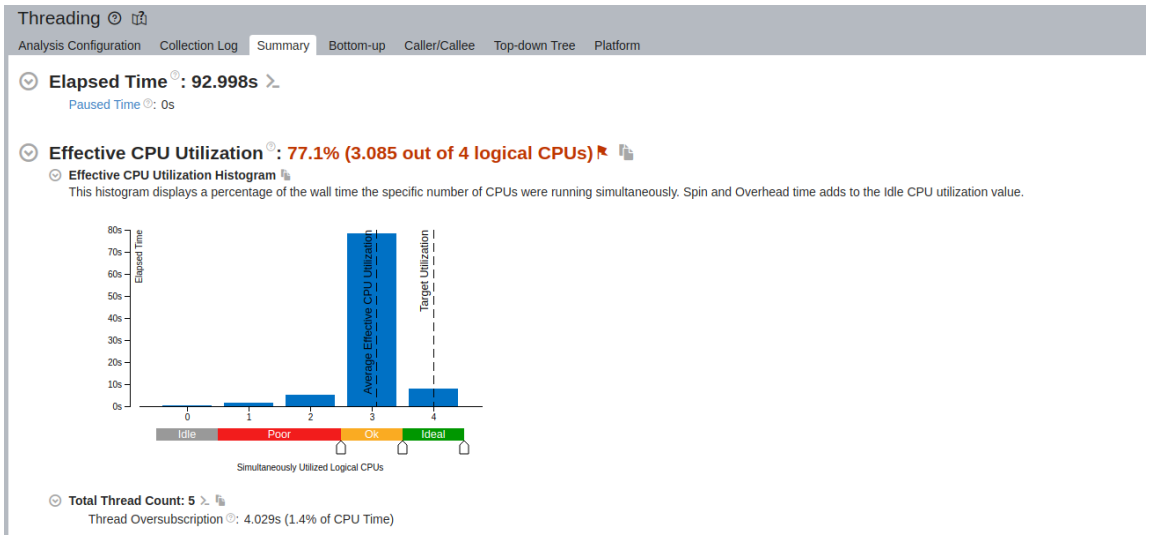


Figure 45 CPU UTILIZATION

8 Thread

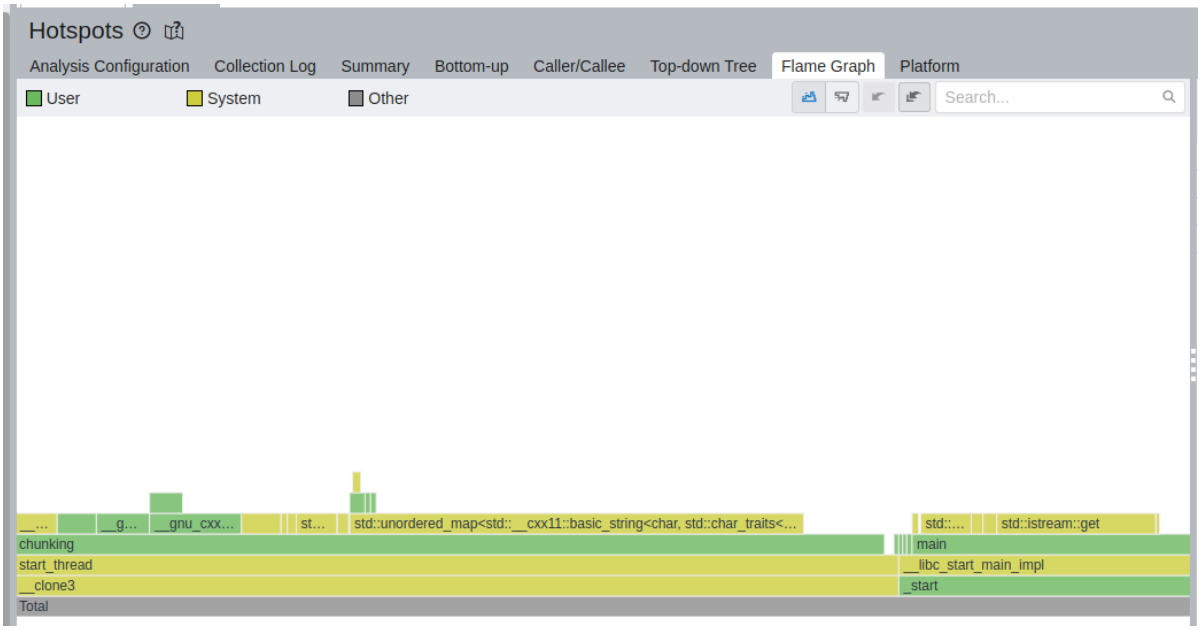


Figure 46 FLAME GRAPH

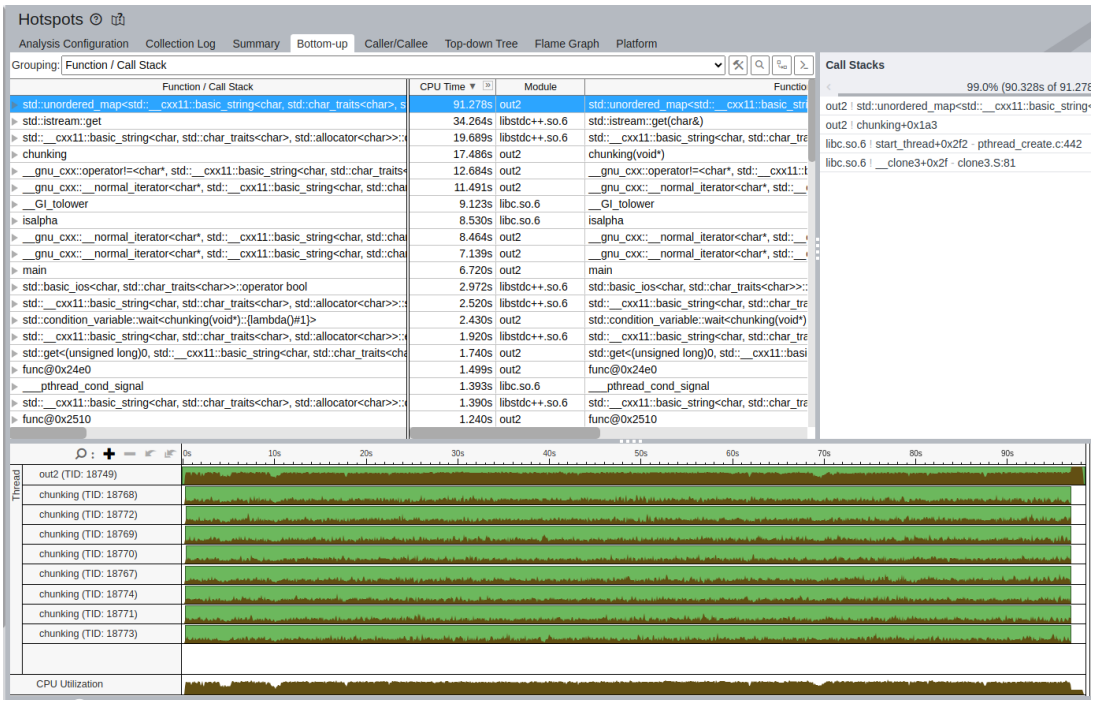


Figure 47 HOTSPOT ANALYSIS

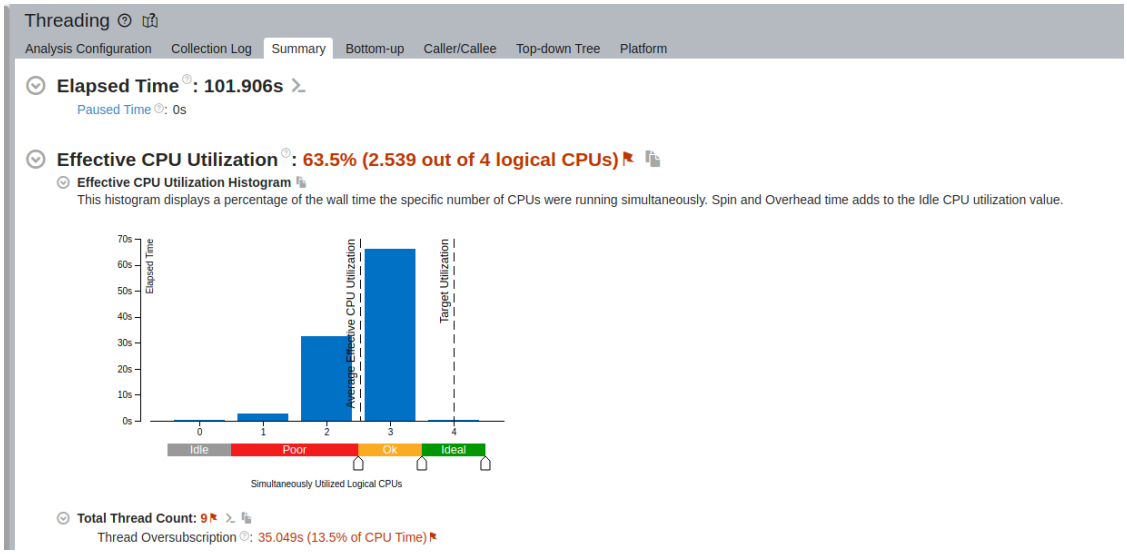


Figure 48 CPU UTILIZATION

Analysis of Multithreaded Execution

Impact of Affinity on Performance

CPU affinity determines how threads are scheduled on processor cores. By binding specific threads to specific cores, affinity aims to reduce cache misses, improve core utilization, and minimize context switching overhead.

a) Cache Locality

- Without affinity, threads are assigned dynamically to any available core, causing cache invalidation when a thread switches to a different core. This results in frequent memory access, increasing execution time.
- With affinity, each thread remains on a dedicated core, leading to better cache reuse and fewer memory stalls. However, this only improves performance when the workload benefits from cache locality.

b) Core Utilization

- Q2 (No Affinity) achieved 96.1% utilization at 4 threads but degraded to 90.2% at 8 threads, showing good parallelism but slight resource contention.
- Q2_Affinity had lower utilization (87.4% at 4 threads, 70.6% at 8 threads), indicating inefficiency in thread-core mapping due to imbalanced workload distribution.

- Q3 (No Affinity) performed better than Q3_Affinity, achieving a 71.7% utilization at 4 threads, while affinity degraded performance due to unoptimized memory access patterns.

Threads	Q2 (%)	Q2_Affinity (%)	Q3 (%)	Q3_Affinity (%)
1	31.2	31.3	35.3	37.1
2	62.2	67.6	69.4	70.2
4	96.1	87.4	71.7	77.1
8	90.2	70.6	75.8	63.5

c) Context Switching Overhead

- Threads running without affinity frequently migrate across cores, incurring context switching penalties.
- When affinity is applied, OS thread migration is prevented, but load balancing can be negatively affected, leading to underutilization of some cores.

Conclusion on Affinity

- Affinity benefits workloads with high cache reuse and minimal thread interaction.
- If threads rely on shared data, affinity can backfire, as some cores may remain idle while others become overloaded.
- Q2 suffered from affinity due to data dependencies, while Q3 was more balanced without affinity.

Analysis of Speedup Achieved by Thread Count

a) Speedup Table

The following table shows execution times for different threads across all implementations:

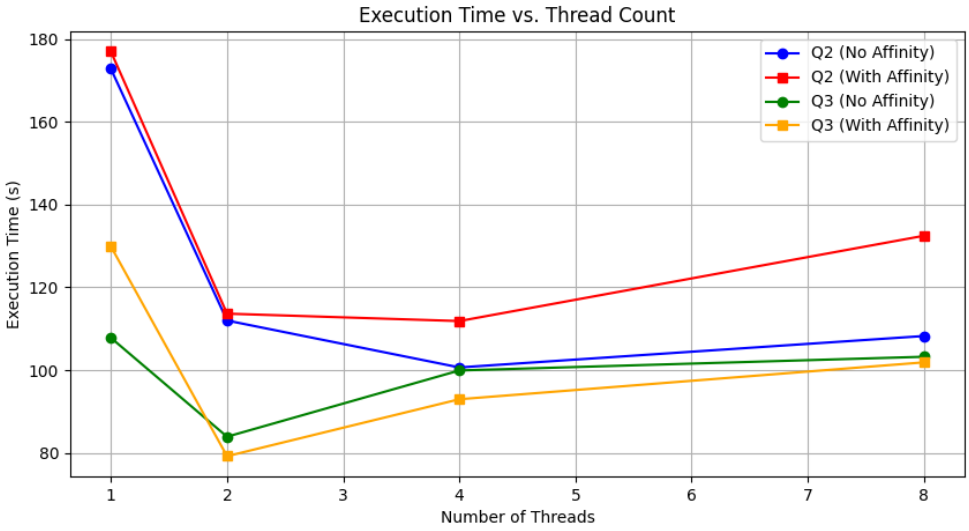
Threads	Q2 Time (s)	Q2 Affinity Time (s)	Q3 Time (s)	Q3 Affinity Time (s)
1	172.78	176.95	107.83	129.89
2	112.00	113.66	83.94	79.236
4	100.67	111.87	99.94	92.99

Threads	Q2 Time (s)	Q2 Affinity Time (s)	Q3 Time (s)	Q3 Affinity Time (s)
8	108.26	132.47	103.25	101.90

Speedup Calculation

Threads	Q2 Speedup	Q2_Affinity Speedup	Q3 Speedup	Q3_Affinity Speedup
1	1.00	1.00	1.00	1.00
2	1.54	1.56	1.28	1.64
4	1.71	1.58	1.07	1.40
8	1.59	1.34	1.04	1.27

b) Speedup Graph



c) Observations

1. Q2 achieves near-linear speedup to 4 threads but slows down at 8 threads due to thread contention.

2. Affinity negatively impacted Q2 at higher thread counts, likely due to poor memory access patterns.
3. Q3 scales well up to 4 threads but sees diminishing returns at 8 threads, indicating that it is not fully CPU-bound.
4. Affinity showed mixed results; it slightly improved Q3 at 2 threads but reduced performance at higher thread counts.

Hotspot Analysis

A hotspot analysis identifies which parts of the code consume the most execution time. Using Intel VTune Profiler, we analyzed both the serial and parallel versions.

a) Serial Version Hotspots

- File I/O operations (ifstream file.read()): High overhead due to large dataset size.
- Word extraction and frequency calculation: Expensive in single-threaded execution.

b) Parallel Version Hotspots

- Thread synchronization (mutex locks): Significant overhead when updating shared word frequency maps.
- Condition variable waiting: Threads often wait on tasks, leading to load imbalance.
- Memory bandwidth bottleneck: Increased cache misses at higher thread counts.

Challenges Faced & Solutions

a) Workload Distribution Imbalance

- Problem: Some threads finished faster than others, leading to idle CPU cycles.
- Solution: Implemented task queues instead of static chunk division to ensure better load balancing.

b) Mutex Contention on Global Word Frequency Map

- Problem: Multiple threads frequently locked the shared word map, reducing parallel efficiency.
- Solution: Used per-thread local maps, merging them only after all threads finished.

c) Performance Drop at Higher Thread Counts

- Problem: Speedup stagnated or degraded beyond 4 threads, especially with affinity.
- Solution: Optimized CPU affinity assignments and ensured better cache locality by reducing cross-core data sharing.

d) File I/O Bottleneck

- Problem: Reading large chunks of text serially before distributing to threads created an initial bottleneck.
- Solution: Used asynchronous file reads and efficient buffer management to reduce I/O wait times.

Final Takeaways

1. CPU affinity is beneficial for workloads with predictable memory access but can degrade performance if load balancing is affected.
2. Threading improves performance but beyond a certain point, memory bandwidth and synchronization overhead limit gains.
3. Careful workload balancing, minimizing synchronization overhead, and optimizing data locality are key to achieving efficient parallelism.