PARALLEL & DISTRIBUTED COMPUTING
ASSIGNMENT #01

Sahrish Mustafa 22i0977

Code Explanation Video:

Youtube Link

Google Drive Link (better quality)

Introduction

Understanding the problem

For this assignment, we were tasked to perform statistical analysis on datasets. These datasets are over 1GB in size, and performing computations on data this large will be slow and inefficient - if done serially.
The focus of this assignment, is, therefore, to improve performance of computation in terms of time and power, by parallelizing the process through multi-threading.

The pthread library is used to make separate units of execution, or threads, to perform the same task on different but related sets of data.

Generalized Solution

In order to make sure that multiple threads do not access the same memory simultaneously, synchronization techniques are used to make sure that shared memory is accessed serially; that is, one at a time. This portion of the code is called the “critical section”. The synchronization tools I used are mutexes, to lock that portion of the code.

```
Pthread_mutex()
```

is the function used to do this.

Each code has been tested with multiple threads, and the execution time for the parallelized computation has been recorded.
There are two versions of each task; one that maps each thread to a specific core, and the other which does not, and leaves the mapping to the Operating System.

```
pthread_setaffinity_np()
```

Is the command used to achieve this.

Laptop Specifications

TOTAL STORAGE	512 GB
RAM	8 GB
PROCESSOR	AMD RYZEN
PHYSICAL	6

Tasks

Task 5

This task involves a matrix of numbers, taken from the [LargeST: A Benchmark Dataset for Large-Scale Traffic Forecasting] (Liu #). The matrix represents the traffic data received from 8600 sensors in California in 2021. There are 35,040 samples. Therefore, the matrix has dimensions (35040,8600). The following tasks are performed on it in parallel:

- Total sum of all elements in the matrix.
- Maximum and minimum elements in the matrix.
- Row-wise and column-wise sums.

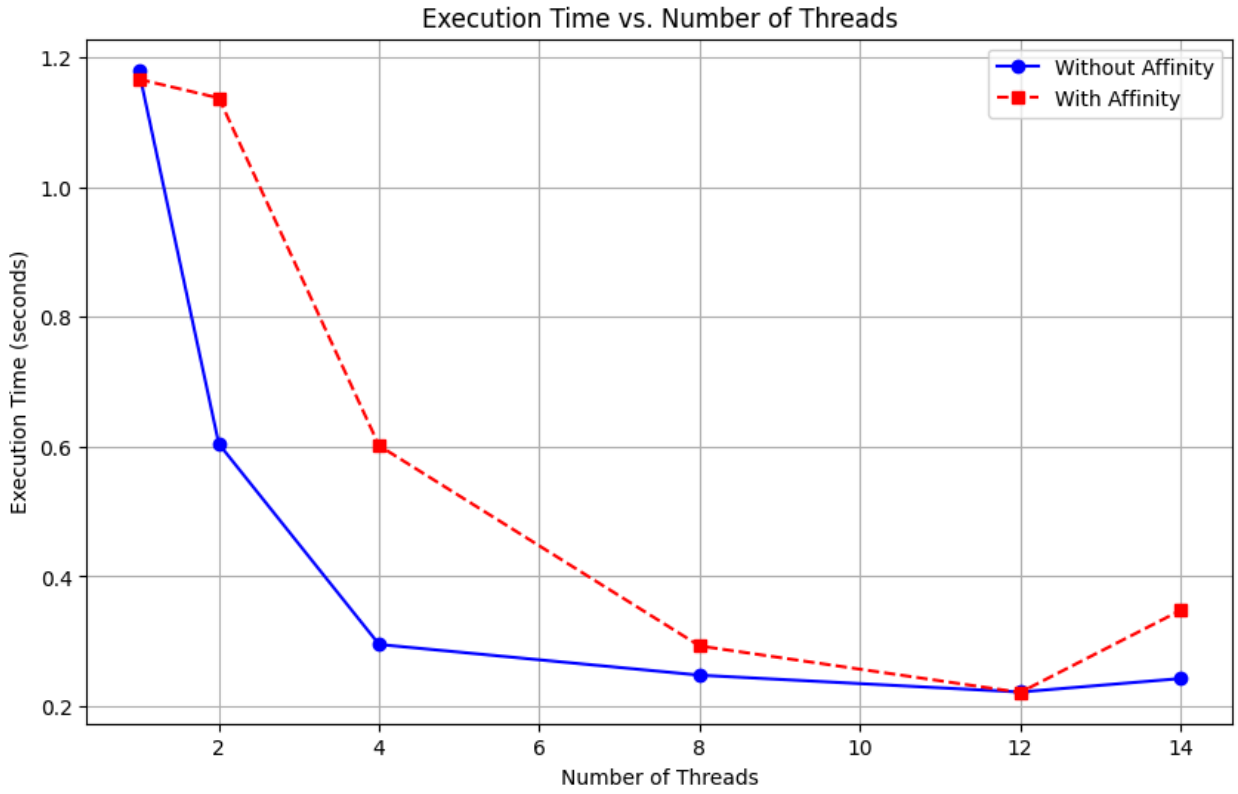
The data from the matrix is read into a 2D array serially by the main thread. The tasks are performed in parallel, using:

```
sum_matrix()
```

> Note: Since initialization of the shared memory (2D matrix) requires file reading by a single thread, most of the total execution time is spent on it. This will be demonstrated in the performance analysis section, using the AMD MicroProfiler.

Performance Analysis

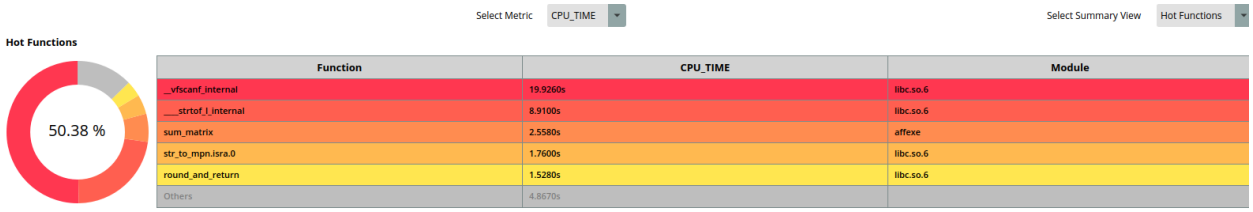
The following table and graph shows execution time:



Thread count	Without Affinity (s)	With Affinity (s)
1	1.180016	1.166024
2	0.604187	1.137065
4	0.295219	0.601395
8	0.247922	0.292903
12	0.221931	0.221333
14	0.242712	0.347625

Best time: 12 threads

Hotspot Analysis

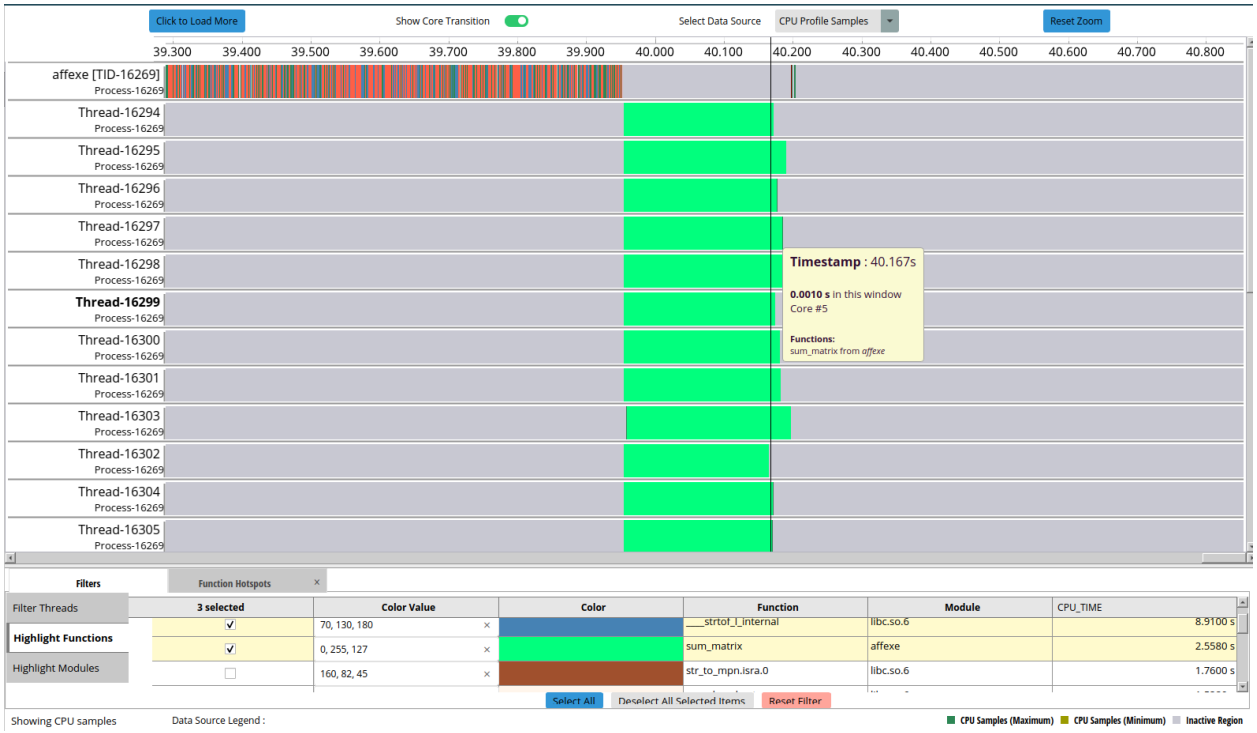


Function	CPU_TIME	Module
__vfscanf_internal	19.9260s	libc.so.6
___strtof_l_internal	8.9100s	libc.so.6
sum_matrix	2.5580s	affexe
str_to_mpn.isra.0	1.7600s	libc.so.6
round_and_return	1.5280s	libc.so.6
Others	4.8670s	

The first 2 hot (SYSTEM) functions are due to the file reading in the beginning, which takes the largest amount of time. Followed by sum_matrix (USER CODE - task), which is executed in parallel by 12 threads.

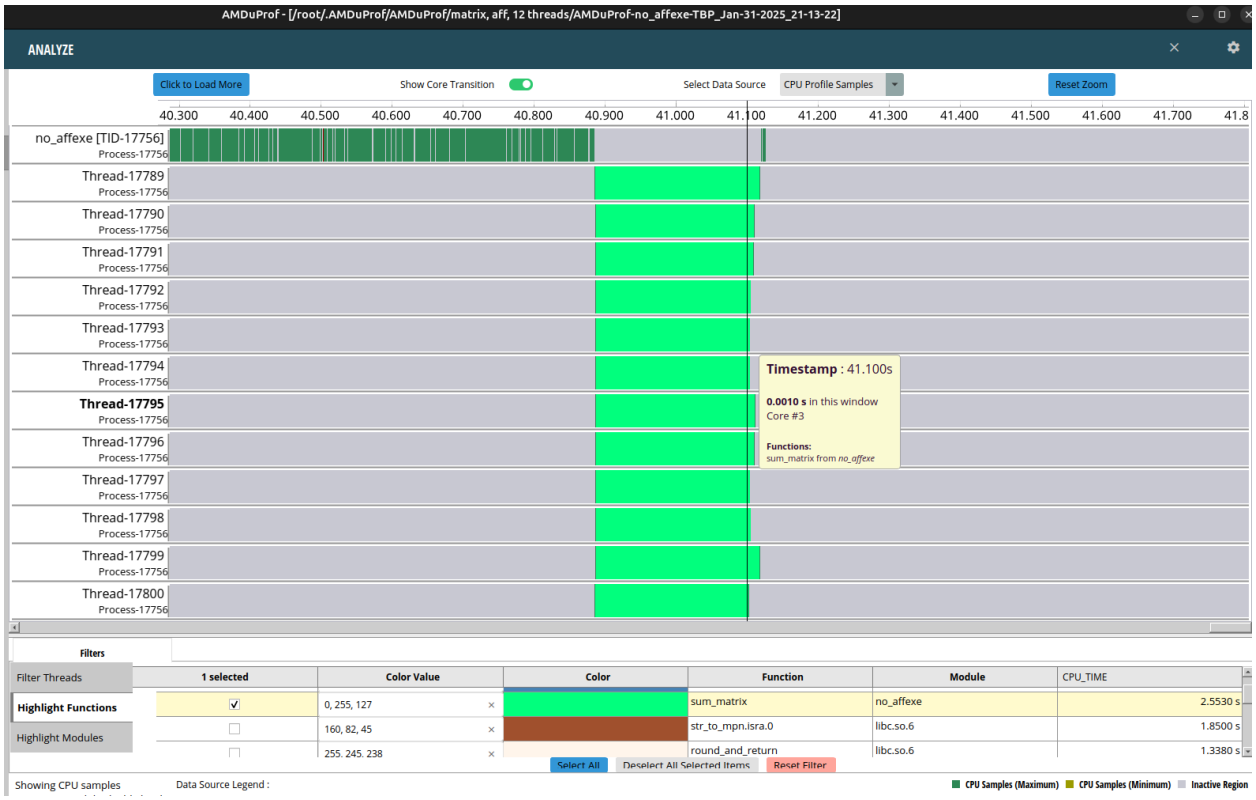
CPU utilization by threads

With Affinity



All threads being used in parallel, with no core switching, as all threads have been binded to a core (shows up in dark red - which is absent)

Without Affinity



As is evident from the profiler and the execution times, the thread affinity does not improve execution time significantly. This is because the operating system usually binds the threads to the core even without using the affinity command (since both profiles above lack core transitions).

Task 3

This task involves a dataset called the [MultiUN: A Multilingual Corpus from United Nation Documents] (Eisele and Chen #), involving 10 years of data stored in multiple .xml files in separate folders labelled by year.
I have used the English language portion. The files have been converted to .snt files and compiled into a singular folder called “combined_folder”. The following task is to be performed on the data:

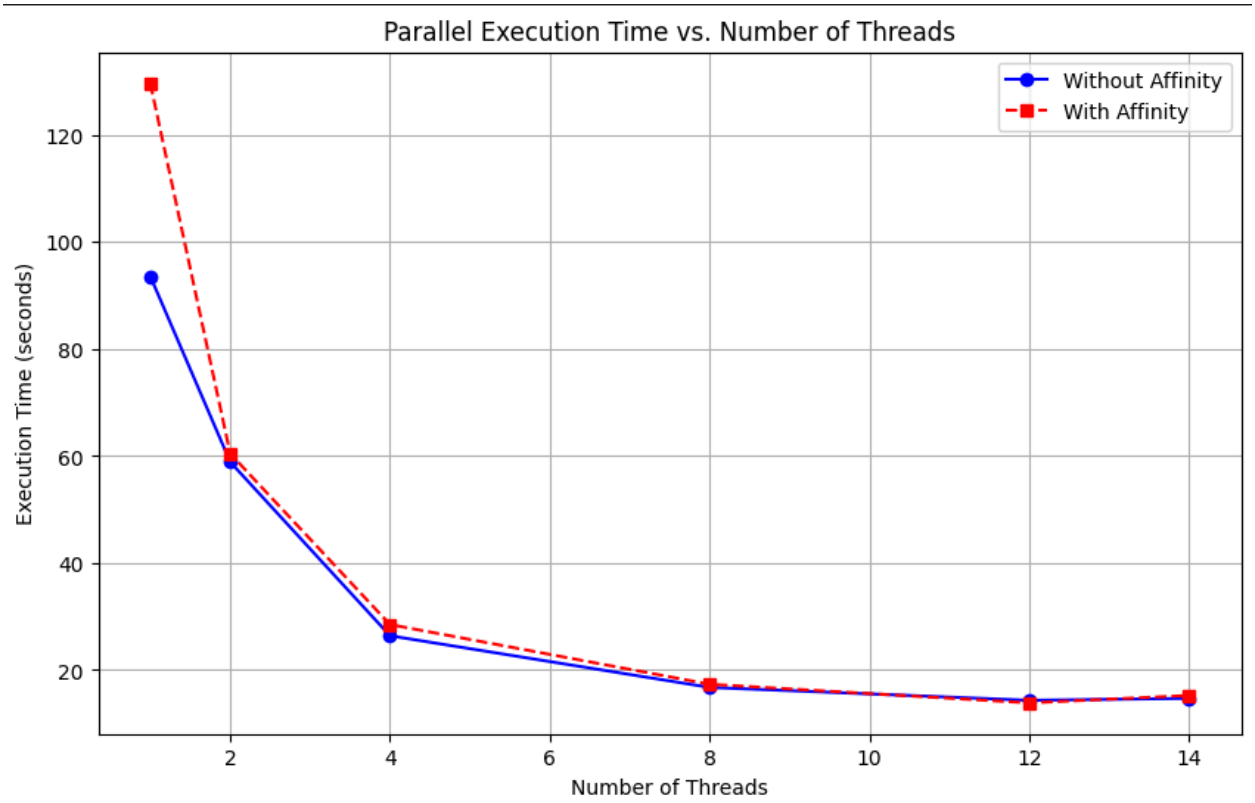
- Term frequency for each word in the file.
- Total number of unique words.

For this task, since the data is already divided into 100,535 files, the file reading is done in parallel by the threads before computation, in a cyclic manner.

Threads dynamically fetch files from a shared file list instead of being assigned fixed portions. Each thread reads and processes its allocated file, inserting words into a shared hash table.Synchronization is achieved using fine-grained mutex locking on individual hash table indices. To prevent race conditions, fine-grained mutex locks (hashMutex[i]) are applied only on the specific hash bucket.
This allows different threads to update different hash buckets simultaneously instead of locking the whole table.

Performance Analysis

The following table and graph shows execution time:



Thread count	Without Affinity (s)	With Affinity (s)
1	93.609348	129.687128
2	58.891999	60.456584
4	26.402951	28.499954
8	16.747526	17.332930
12	14.326771	13.852473
14	14.703398	15.244874

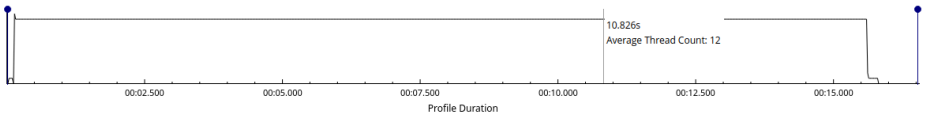
Best time: 12 threads

It is noticeable that the performance of the system with affinity is the worst when the code is mostly serial: using only one thread- this is because binding that singular threads to one core means it has to wait for that core to be free rather than being assigned to another core despite it being free.

Hotspot Analysis

The hottest user function (in the code) is the insertWord() function, which is what is used for recording word counts, followed by distinguishWord() function, which is used to identify and extract words out of the text. The hottest system functions include vfscanf_internal(), which is used by every thread for reading files, and the mutex lock, as every hashtable index has a separate mutex, thus used frequently.

With Affinity



Select Metric: Thread Concurrency
Duration (ms): 0.00 – 16579.00 Apply Reset

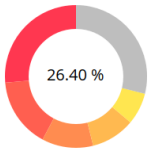
Select View: Time based hotspots | Value Type: Sample Count | System Modules: Include / Exclude | Search for functi... Search

Functions	Modules	CPU_TIME(s) ▼
insertWord	affexe	16.1770
distinguishWord	affexe	13.8860
isValidWord	affexe	10.0630
hashFunc	affexe	7.5700
processFile	affexe	5.0560

Number of hardware threads being utilized simultaneously

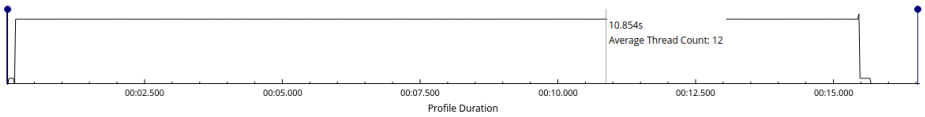
Select Metric: CPU_TIME | Select Summary View: Hot Functions

Hot Functions



Function	CPU_TIME	Module
__vfscanf_internal	36.5890s	libc.so.6
pthread_mutex_lock@@GLIBC_2.2.5	22.1000s	libc.so.6
insertWord	16.1770s	affexe
distinguishWord	13.8860s	affexe
isValidWord	10.0630s	affexe
Others	39.7680s	

Without Affinity

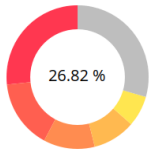


Select Metric: Thread Concurrency
Duration (ms): 0.00 – 16581.00 Apply Reset

Select View: Time based hotspots | Value Type: Sample Count | System Modules: Include / Exclude | Search for functi... Search

Functions	Modules	CPU_TIME(s) ▼
insertWord	noaffexe	16.6390
distinguishWord	noaffexe	13.7030
isValidWord	noaffexe	9.7840
hashFunc	noaffexe	7.6790
processFile	noaffexe	4.9810

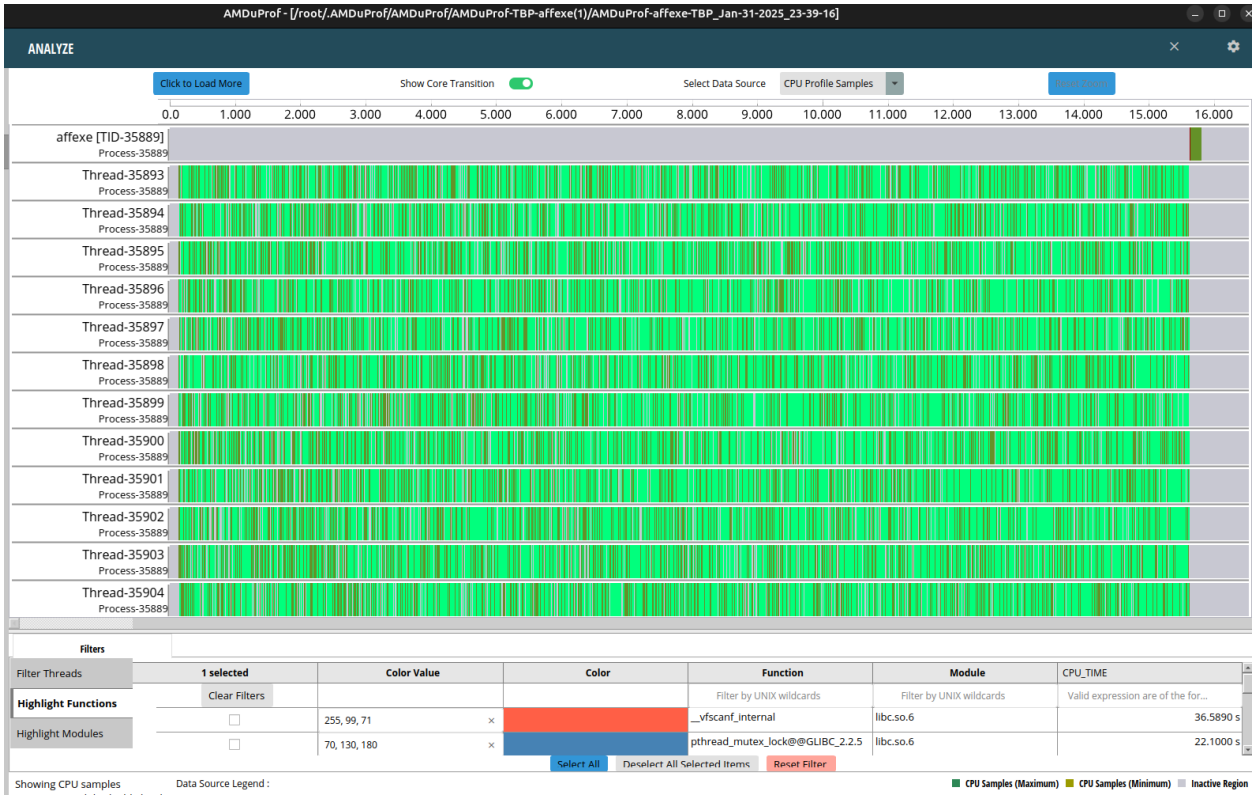
Hot Functions



Function	CPU_TIME	Module
__vfscanf_internal	37.4320s	libc.so.6
pthread_mutex_lock@@GLIBC_2.2.5	21.9360s	libc.so.6
insertWord	16.6390s	noaffexe
distinguishWord	13.7030s	noaffexe
isValidWord	9.7840s	noaffexe
Others	40.0830s	

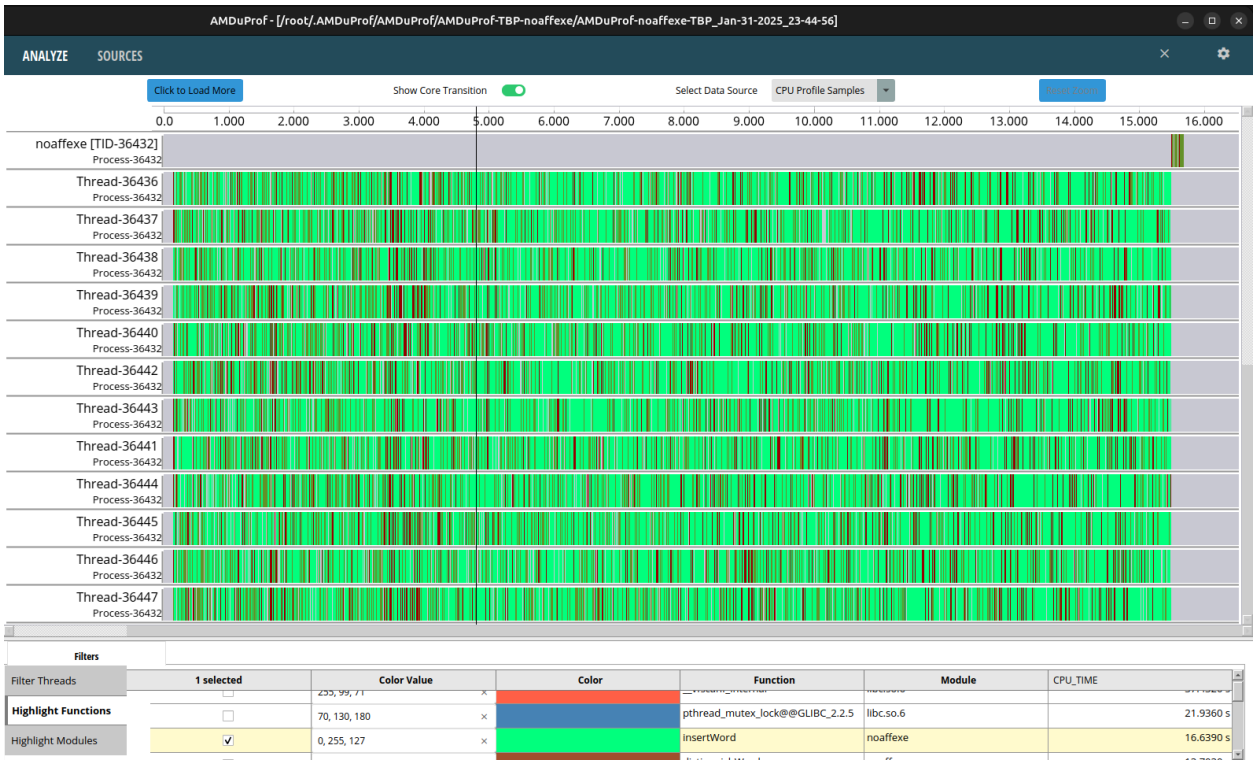
CPU Utilization by threads

With Affinity



All 12 threads are being utilized simultaneously, being used for insertWord() (visible in light green). No core transitions(in dark red) visible as each thread is bound to a specific CPU.

Without Affinity



All threads are being utilized simultaneously, with core switches(visible in dark red).

Thread Affinity & Affects

Without using thread affinity, when a core switch causes the thread to move to a different thread, it will require a cache reload, as opposed to that, when thread affinity is used, cache reuse is improved as memory is accessed a lot faster. It reduces context switching overhead, but the cores may be underutilized, as was evident form the worst case of task 3. The OS scheduler already does a good job of distributing work amongst threads, so using affinity has not presented a significant advantage.