**Lebanese university**

الجامعة اللبنانية

كلّيّة الهندسة

الفرع الثالث

**Faculty of engineering**

**Branch 3**

**Concurrent Programming project report**

**2024-2025**

Presented for:

Doctor Mohammad Aoude

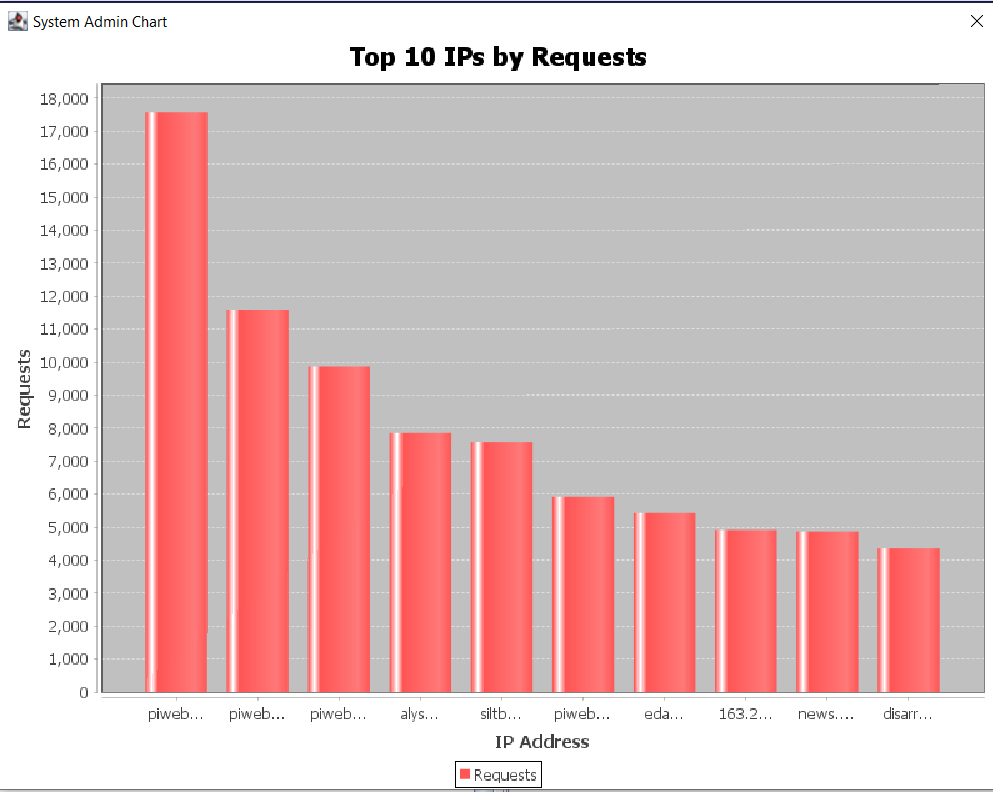
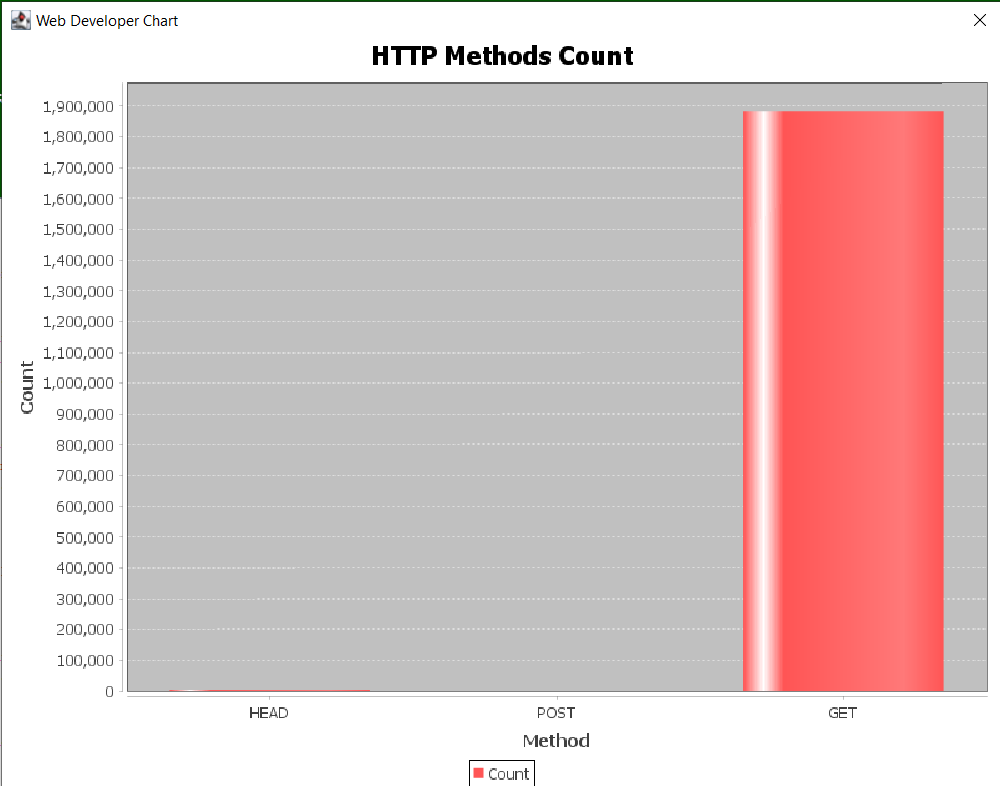
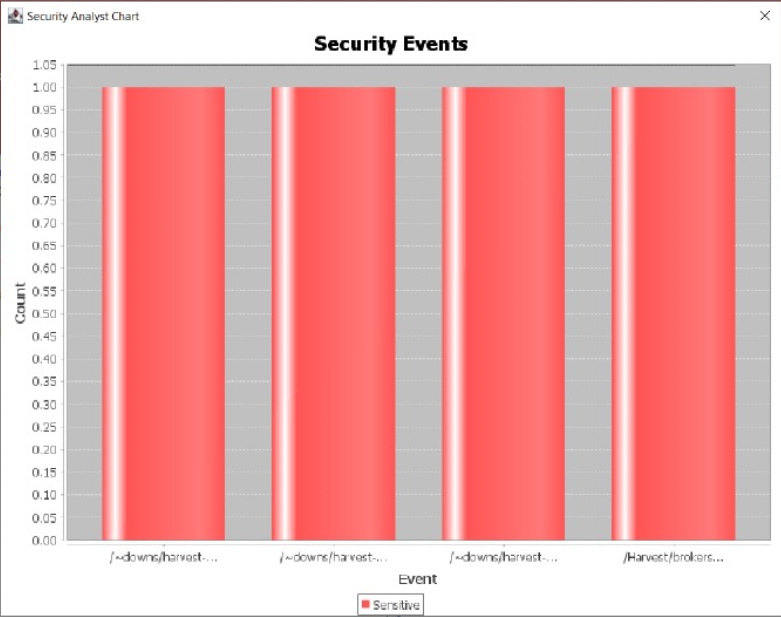
Prepared by:

* Fatima Jawad (6496)
* Fatima Moselmani (6468)

1.Motivation and Problem:  
  
In today's highly connected digital world, log files are automatically generated by almost every digital system. They are produced in real-time from hardware to applications. These log files are essential for diagnostics, monitoring, and analytics of the performance of the system.  
  
Examples of systems that generate these log files are such as web servers, cloud platforms, operating systems, network devices and many others..  
  
In earlier computing eras, administrators and developers often relied on manual inspection of log files in a slow and un-scalable methods. Although it is functional for small scale systems, manual log analysis quickly becomes impractical in modern environments. Due to lack of visualization, human errors and biasing and incapability of following up with real time dependent procedure, the need of automatically generated log files has become an essential need.  
  
This growing complexity and volume of logs has made manual analysis obsolete, leading to the need for automatically generated insights using concurrent programming techniques to process data efficiently and in parallel.

2. Design

2.1 Algorithms

* Sequential Algorithm
  + - The sequential approach reads the log file line by line using a BufferedReader.
    - Each line is parsed into a structured LogEntry object using a regular expression pattern.
    - A list of all parsed entries is then analyzed according to the selected user role:
      * System Admin: counts server errors, failed logins, unique IPs. 
      * Web Developer: analyzes HTTP methods and top visited URLs. 
      * Security Analyst: detects sensitive endpoint access and suspicious methods. 
* Parallel Algorithm
* The parallel implementation uses a fixed thread pool via ExecutorService.
* The log file is first read completely into a List<String> of lines.
* This list is then split into chunks (batches of lines), and each chunk is submitted as a separate task to the thread pool.
* Each thread parses its chunk independently into List<LogEntry>.
* After all tasks finish (Future.get()), the partial results are merged into a final combined list.
* This unified list is passed to the same analysis logic as in the sequential mode.
* Justification
  + - Since log entries are independent and order doesn't affect parsing or analysis, this chunk-based parallelism is safe and highly effective.
    - ExecutorService allows easy control over thread count, supports reuse of threads, and handles task queuing efficiently.
    - Using a fixed thread pool limits CPU overuse while achieving high performance through concurrency.

2.2 Data Structures

* Storing parsed log entries: using *list<LogEntry>* for dynamic size with ordered iteration and easy merging
* Count IPs, Methods, URLs : using *HashMap<String, Integer>* for fast aggregation, constant-time insert and lookup.  
  Track unique IPs: *HashSet<String>* for efficient deduplication of IPs.
* Count events (errors, logins): using *AtomicInteger* for thread-safe counting when shared in future extensions.
* Track speedups per thread: using *Map<Integer, Double>* for simple association of thread count to observed speed-up.

3. Implementation Notes — tricky parts, obstacles, how solved.

* **Challenge:**

Swing components must only be updated on the Event Dispatch Thread (EDT). Mixing background threads and GUI updates can cause crashes or unpredictable behavior.

* **How It Was Solved:**

All background processing (sequential and parallel) was wrapped in SwingWorker tasks

The results and visual updates (text output, charts, etc.) were posted to the EDT using SwingUtilities.invokeLater().

This ensured thread safety while keeping the UI responsive.

* **Challenge:**  
  Different user roles (System Admin, Web Developer, Security Analyst) require different types of analysis logic and presentation.
* **How It Was Solved:**

The GUI includes a role selector (JComboBox) that switches analysis logic dynamically.

Three dedicated methods (doSystemAdminAnalysis, doWebDeveloperAnalysis, doSecurityAnalystAnalysis) were created for clean separation.

Each method calculates and displays relevant metrics, and generates role-specific charts using JFreeChart.

4.Testing Methodology

4.1 Correctness Testing  
To ensure that both the sequential and parallel processing modes produce accurate and consistent results, the following methodology was used:

* Baseline Comparison:

1. After processing the same log file, the output from the parallel processor (for each thread count) was compared line-by-line and statistically (e.g., total entries, failed logins, top URLs, suspicious IPs).
2. Results were considered correct if the key metrics (e.g., counts, analysis summaries) exactly matched between sequential and parallel runs.

* Role-Specific Output Validation:  
  For each role (Admin, Developer, Analyst), the output was reviewed to ensure that:

1. The correct analysis logic was triggered.
2. Charts and statistics were updated accordingly.
3. Switching roles dynamically reflected correct and isolated results.

4.2 Performance Testing

1. Thread-Based Performance Evaluation

* The same large log file (e.g., NASA access logs) was processed multiple times using different thread counts: 4, 8, 12, and 16.
* For each test case:
  + The sequential execution time was recorded.
  + The parallel execution time was measured using a fixed-size thread pool.
  + The performance gain (speed-up): T parallel/T seq

1. Result Logging and Visualization

* The measured speed-up values were stored in a CSV file using SpeedupCSVWriter
* A line chart was created with JFreeChart, showing the relationship between thread count and speed-up.
* This allowed visual confirmation of how increased parallelism affected overall processing efficiency.

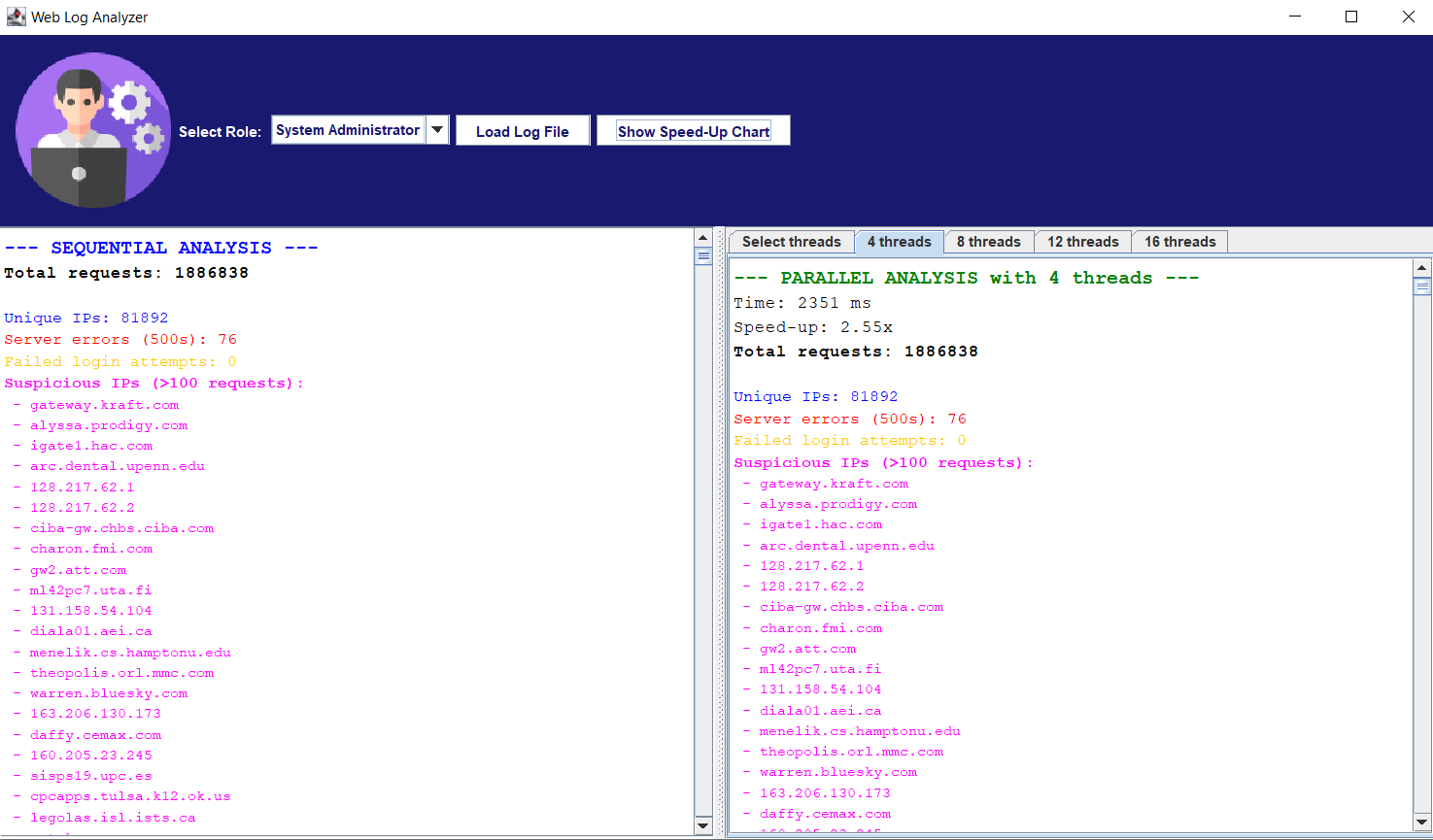
1. System Configuration

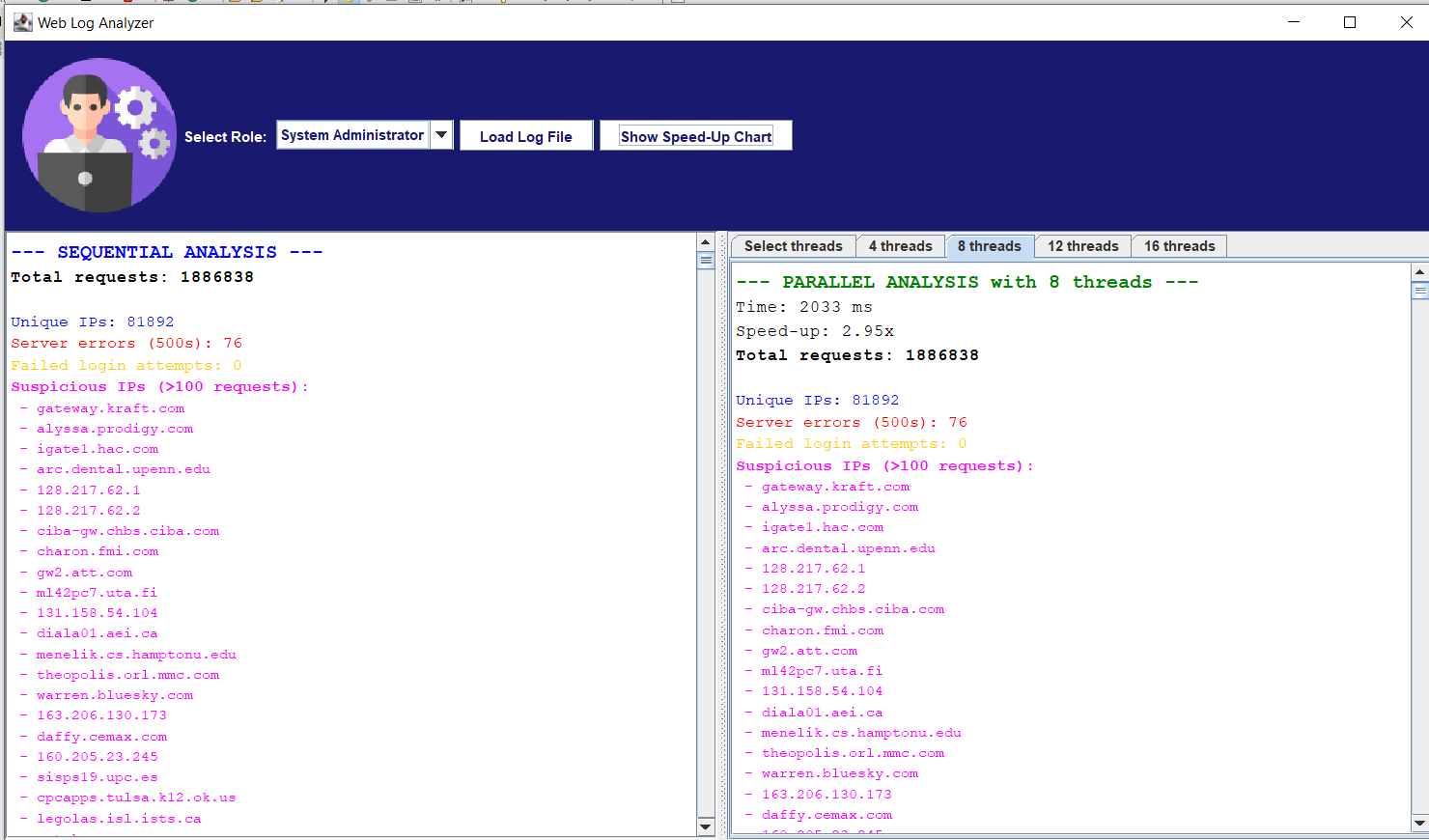
* All performance tests were conducted on a machine with a multi-core CPU (capable of at least 16 parallel threads).
* The Java application was executed using JDK 17 inside Eclipse on a Windows environment.

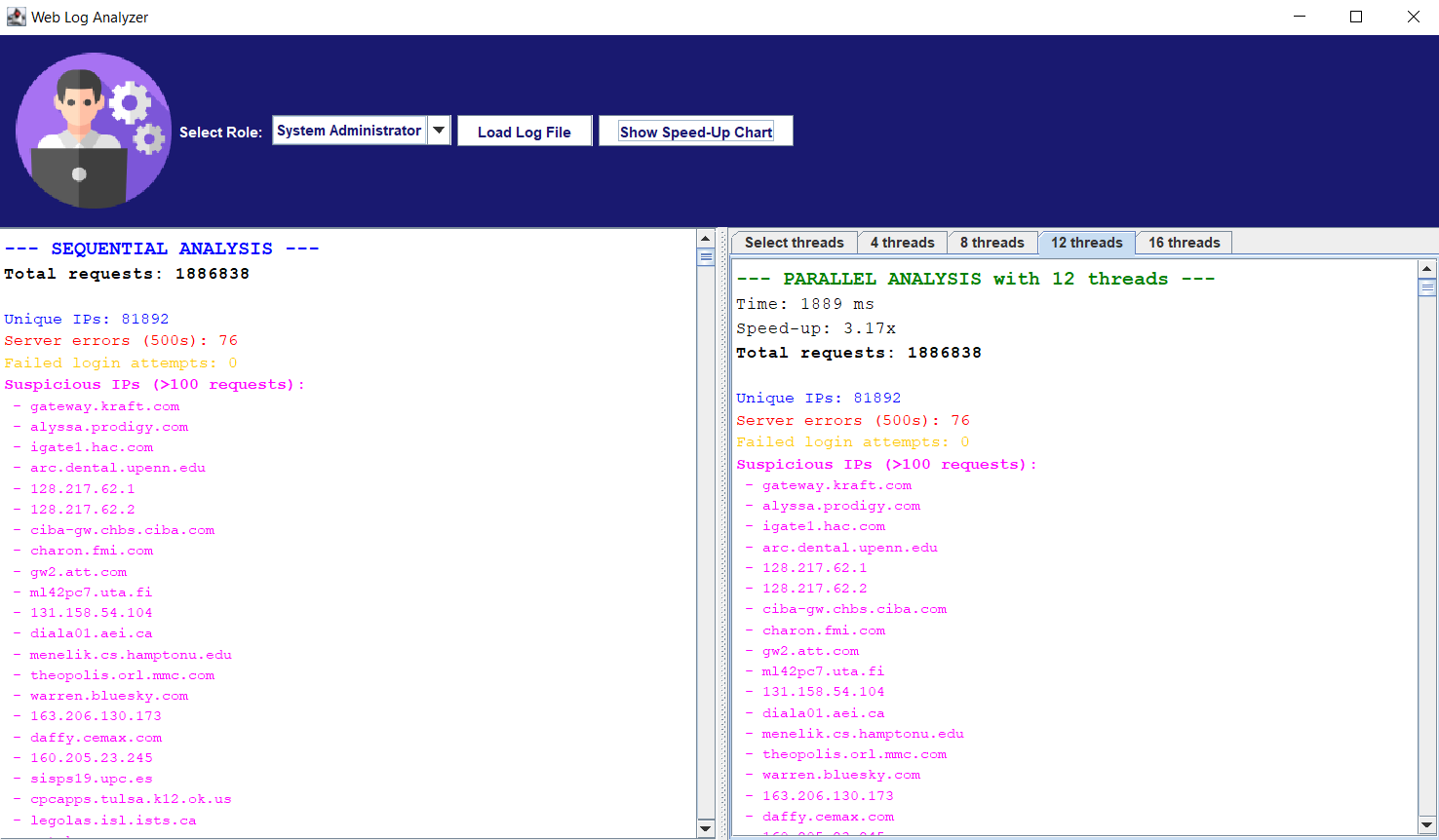
5. Results

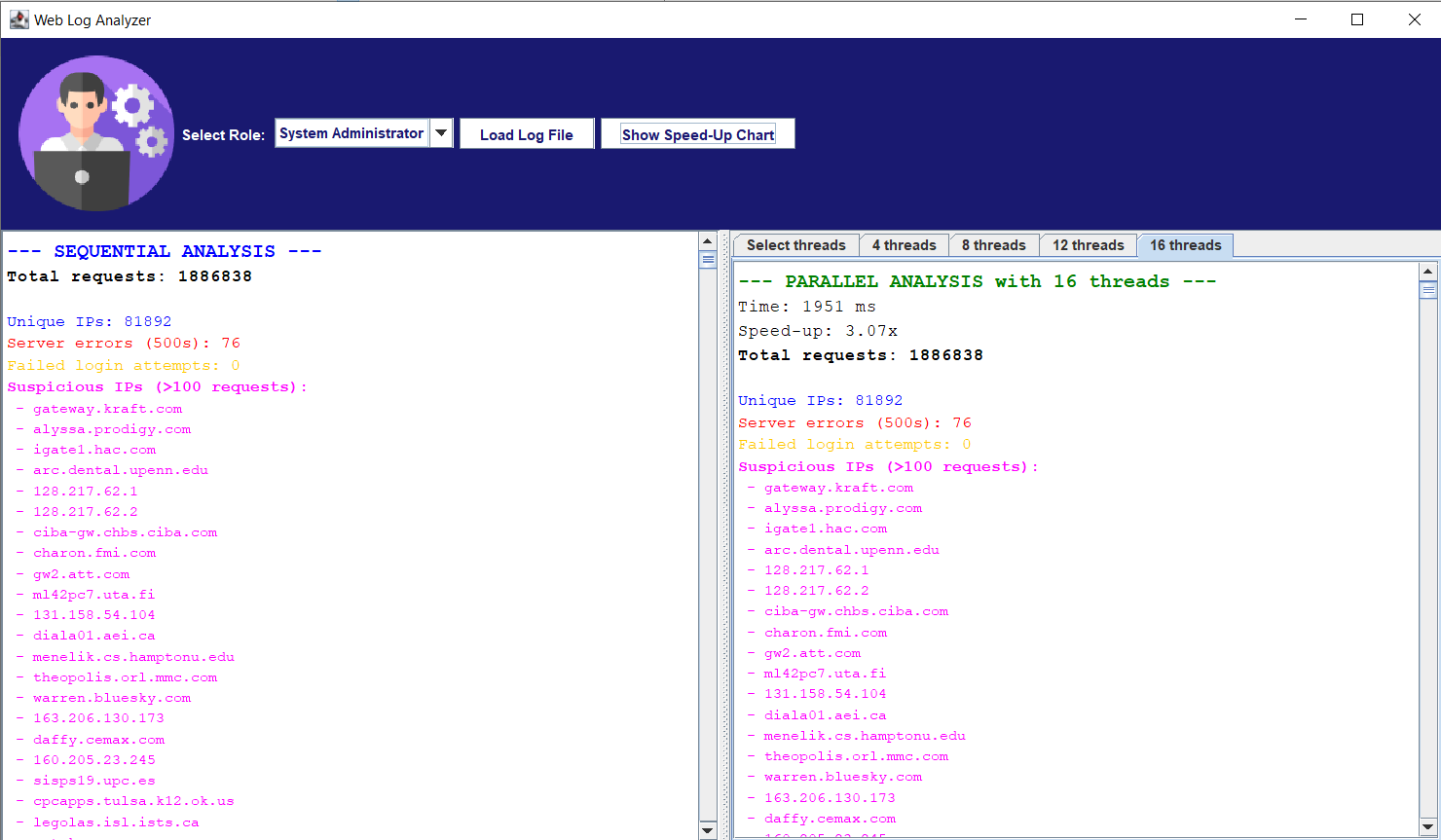
5.1 Speed-Up and Efficiency:

* To evaluate performance, the same large log file was processed using the sequential approach and the parallel approach with varying thread counts: 4, 8, 12, and 16. The total processing time (in milliseconds) was recorded for each configuration.

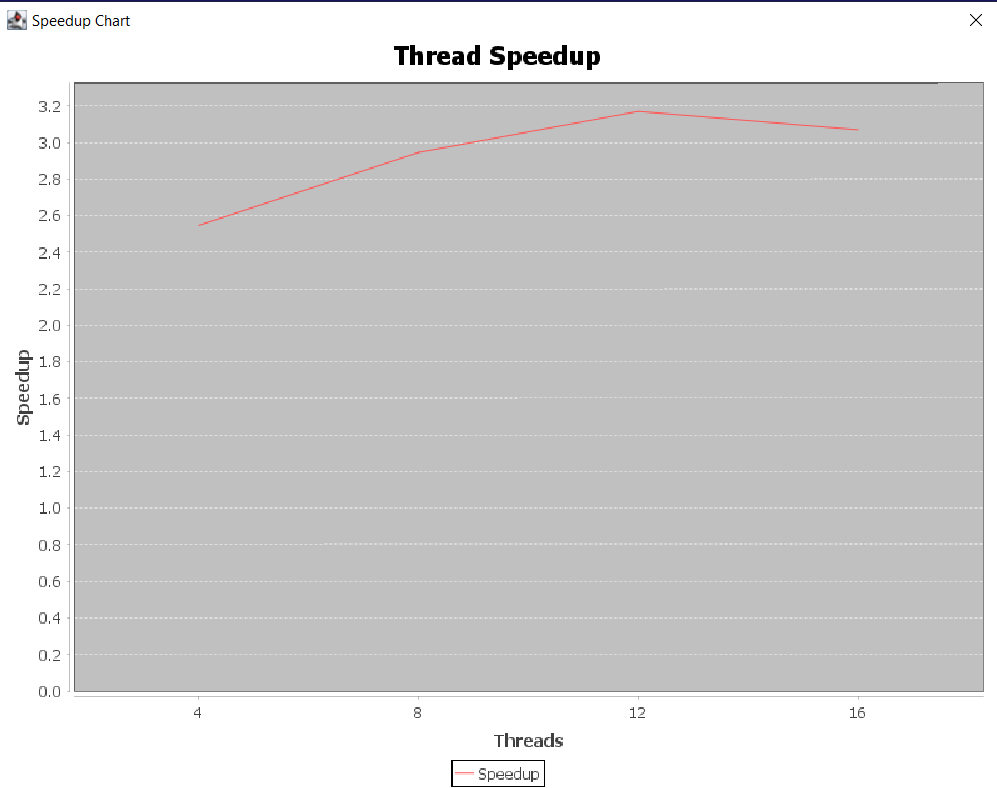
4 threads: 

8 threads: 

12 threads: 

16 threads: 

5.2 Graphical Visualization

A speed-up chart was automatically generated using JFreeChart based on the collected data. The chart displayed: 

x- axis: number of threads

y –axis: speedup

5.3 Scaling Analysis

* Speed-up increases rapidly from 4 to 8 threads.
* From 12 to 16 threads, speed-up gains begin to saturate
* The trend follows Amdahl’s Law, where speed-up is limited by the sequential portion of the task (e.g., file reading, merging results).
* The system scales well up to 8–12 threads, after which I/O overhead and thread coordination costs limit further gains.

5.4 Bottlenecks Identified

* File I/O Time: Reading the entire log file into memory is done in a single thread (even in parallel mode), creating a bottleneck before task distribution.
* Thread Management Overhead: For very high thread counts (16+), the cost of thread scheduling, task submission, and result merging reduces the effectiveness of additional parallelism.
* Memory Usage: Large files loaded fully into memory may strain lower-spec machines; partial or stream-based parsing could reduce footprint.

6. Comparison with sequential -Wins & Trade-offs

6.1 Comparison with Sequential

Parallel processing significantly outperforms sequential processing up to a certain number of threads.

In the tested results, maximum performance gain was observed around 8 to 12 threads, with speed-up reaching up to \*3.17 compared to the sequential execution.

Tasks that are CPU-bound (like parsing large files) benefit heavily from multithreading, especially on multi-core processors.

6.2 Trade-Offs and Limitations

* + At 16 threads, the performance decreased slightly, showing diminishing returns.

This is due to:

* Thread management overhead (context switching, queuing, scheduling).
* Increased memory pressure and garbage collection time.
* The fact that Java’s ExecutorService does not guarantee perfect linear scalability.
  + Unlike sequential mode, which has predictable behavior, the parallel mode introduces complexity:
* Task chunking and merging require additional steps.
* Debugging becomes harder due to concurrency.
* Debugging becomes harder due to concurrency.

7. Conclusion

This project presented a fully functional, role-adaptive Java-based web log analyzer with both sequential and parallel processing modes. It successfully demonstrated how performance can be improved using multithreading with ExecutorService, without compromising accuracy or usability.  
***Key achievements include:***

* A user-friendly GUI tailored to different user roles: System Administrator, Web Developer, and Security Analyst, each with dedicated analysis logic and charts.
* A robust log parsing system capable of handling real-world log formats (in our case Nasa log files)
* A scalable parallel processing implementation that leverages multi-core CPUs to significantly reduce execution time.
* Integrated performance visualization, showing how speed-up improves with thread count, while also revealing the limits of scalability due to thread management overhead.

8. Individual Contributions  
 Fatima Moselmani (6468):

* Designed and implemented the Java Swing GUI, including the role selector, dynamic styling, and integration of charts.
* Developed analysis logic for System Administrator and Security Analyst roles.
* Handled JFreeChart integration for role-based visualizations.
* Participated in testing, interface polishing, and report writing.

Fatima Jawad(6469):

* Implemented the log parsing system, including the LogEntry model and regex-based parser.
* Built both sequential and parallel log processors using ExecutorService.
* Designed the performance evaluation framework, measured speed-up, and created speed-up charts.
* Contributed to data analysis, backend logic, and documentation.