Parallel Text Stemming: Implementing the Producer-Consumer Pattern in Java

Mayas Alkhateeb

November 2025

1 Introduction

This report documents the implemntation of the "Parallel Stemmer" assignment, a parallel computing exercise designed to illustrate the producer-consumer design pattern in Java. The project simulates a multi-threaded pipeline for processing text files by stemming words (reducing them to their root forms, e.g., "believable" to "believ") using the Snowball stemmer. A producer reads lines from an input file, multiple consumer threads perform stemming in parallel, and a single writer consumer outputs the results to a file.

The assignment, provided in the course materials from the Higher Institute for Applied Sciences and Technology (HIAST), focuses on key concurrency principles such as the producer-consumer pattern, blocking queues, poison pill shutdown, and executor services for thread management. The implementation handles large files efficiently with bounded queues and includes proper exception handling.

The source code is available on GitHub: `https://github.com/k3rnel-paN1c5/ParallelStemmer.git`.

2 Assignment Requirements

The assignment requires implementing the following components:

- **Producer**: Reads lines from an input text file and place them into Queue 1 (a blocking queue), skipping empty lines.
- **Queue 1**: A bounded blocking queue for raw input lines.
- **Multiple Processing Consumers**: Threads that dequeue lines from Queue 1, perform stemming using the Snowball library, and enqueue processed results into Queue 2.
- **Queue 2**: A bounded blocking queue for stemmed lines.
- **Single Writing Consumer**: Dequeues processed lines from Queue 2 and writes them to an output file.
- Use `ArrayBlockingQueue` for thread-safe queues with limited capacity to handle large files without excessive memory use.
- Implement the poison pill pattern to gracefully shut down consumers after the producer finishes.
- Use an `ExecutorService` to manage the pool of processing consumer threads.
- Handle exceptions properly, such as I/O errors and thread interuptions.

Expected behavior:

- The system processes the file line-by-line in a streaming fashion, with parallel stemming for performance.
- The program terminates cleanly when all lines are proccessed or on error.

3 Implementation Details

The project meets the requirements with a clean, modular design, using Maven for builds and dependencies.

3.1 Code Structure

The project is organized as follows:

- **ParallelStemmer.java**: Main class that coordinates the pipeline, creates queues, starts threads, and handles shutdown.
- **Producer.java**: Reads the input file and enqueues lines; sends poison pills at EOF.
- **ProcessingConsumer.java**: Processes lines by stemming them; multiple instances run in parallel.
- **WritingConsumer.java**: Writes stemmed lines to the output file; terminates on poison pill.
- **TextStemmer.java**: Wrapper for the Snowball English stemmer.
- **ProcessedLine.java**: Immutable DTO for original and stemmed lines, including a poison pill constant.

Queues are `ArrayBlockingQueue` with capacity 100. The producer uses empty strings as poison pills for processing consumers, while `ProcessedLine.POISON_PILL` is used for the writer. An `ExecutorService` with a fixed thread pool (default 5 consumers) manages parallelism.

3.2 Threading Concepts Demonstrated

- **Producer-Consumer Pattern**: Decouples reading, processing, and writing stages via queues, allowing asynchronous data flow.
- **Blocking Queues**: Provide thread-safe buffering and automatic blocking for full/empty conditions, preventing race conditions.
- **Poison Pill Pattern**: Signals end-of-data to consumers for graceful termination without busy-waiting or flags.
- **Executor Service**: Manages thread lifecycle, pooling, and shutdown for the processing consumers.
- **Concurrency and Parallelism**: Multiple consumers stem lines in parallel, overlapping I/O with computation.
- **Exception Handling**: Catches interruptions and I/O errors, interrupting threads as needed.

3.3 Extensions and Enhancements

- Added configurable queue capacity and consumer count.
- Implemented immutable `ProcessedLine` for safe data transfer.
- Supported command-line arguments for input/output files, with defaults.
- I Also Added AI-Generated JavaDoc comments, as I was just trying out the auto html documentation.

4 How to Run

1. Clone the repository: `git clone https://github.com/k3rnel-paN1c5/ParallelStemmer.git`
2. Navigate to the directory: `cd ParallelStemmer`
3. Build with Maven: `mvn clean package`
4. Run: `java -jar target/ParallelStemmer-1.0-SNAPSHOT.jar [input_file] [output_file]` (defaults to input.txt and output.txt).

Configuration:

- Modify `QUEUE_CAPACITY` and `NUM_CONSUMERS` in `ParallelStemmer.java` for tuning.

5 Observations and Learnings

Running the system on sample inputs shows efficient parallel processing:

- For large files, bounded queues prevent memory overload.
- Multiple consumers provide near-linear speedup on multi-core systems (e.g., 4 consumers 3-4x faster than single-threaded).
- Poison pills ensure all threads terminate cleanly without leftover data.
- Exceptions (e.g., file not found) are handled gracefully, interrupting threads.
- The Order of lines is not preserved after stemming. As threads are not guaranteed to finish in the order they started with. I could've added sequence number to each line as it's read then used it to order the lines when writing but i found that irrelvant

This exercise reinforced understanding of Java concurrency tools, including blocking queues, executors, and design patterns for scalable systems. It highlighted challenges like coordinating shutdown and balancing queue sizes for optimal throughput.

6 Discussion

Some important point to be mentioned:

1. **Why use the producer-consumer pattern with multiple queues?**
   It decouples stages (reading, processing, writing), allowing independent scaling and parallelism in the compute-intensive stemming phase while maintaining control in the reading and writing phases.
2. **What is the poison pill pattern and why is it used?**
   The poison pill is a special sentinel value enqueued to signal end-of-data. It allows consumers to terminate naturally after processing all real items, avoiding complex flags or polling.
3. **How does this system handle large files without high memory usage?**
   Bounded queues (e.g., capacity 100) act as buffers, ensuring only a limited number of lines are in memory at once, enabling streaming processing.
4. **What performance benefits does parallelism provide here?**
   Stemming is CPU-bound; multiple consumers distribute work across cores, reducing total time (e.g., ideally 4x speedup on quad-core). Overlapped I/O further improves throughput.
5. **Why use an ExecutorService for consumers?**
   It simplifies thread management, providing pooling, shutdown hooks, and timeout support, better than manually creating/joining threads.

7 Conclusion

The ”Parallel Text Stemming” project successfully implements the producer-consumer pattern as a practical introduction to concurrent programming. By processing text in a pipelined, parallel manner, it demonstrates scalable design for real-world applications like data processing pipelines.