BGU 2026-1 SPL Assignment 2

TAs: Amit Hendin, Gur Elkin, Dan Zlotnikov

December 9, 2025

Contents

1 Introduction

In this assignment you will implement a *Linear Algebra Engine* (LAE) that performs linear algebra computations using a custom thread pool, while managing shared resources safely. The goal is to gain hands-on experience with multi-threading, synchronization, and correct usage of shared data structures.

The assignment is to be done in pairs and written in Java. See the course website for material on compiling and running Java, working with threads, and handling shared resources.

1.1 Environment Setup

To ensure that your code compiles and runs as expected, follow the environment setup document on the course website. Once completed, add the skeleton project files provided there to the root directory of your project.

1.1.1 Maven

This project is organized as a Maven-based Java application. Maven is a build automation and dependency management tool that uses a standard directory structure and a configuration file named `pom.xml`.

Maven operates by executing predefined build phases such as `compile`, `test`, and `package`. Each phase may compile Java code, download dependencies, run tests, and produce output files such as JAR archives. Maven retrieves required libraries and plugins from remote repositories and caches them under `~/.m2/repository` for reuse.

The state and behavior of the project are defined in `pom.xml`, which specifies the project name, Java version, dependencies, and plugin configuration.

The standard layout is:

```
src/main/java/        Application source code
src/test/java/        Unit tests
target/               Build output (generated automatically)
pom.xml               Maven project configuration
```

All program source files go under `src/main/java/`. Test files go under `src/test/java/`.

Before working with the project, ensure that:

- Java 21 or a compatible version is installed, and the `java` and `javac` commands are in the system path.
- Maven is installed. Running `mvn -version` should display version information and show that Maven detects the correct Java installation.

To compile all Java source files:

```
mvn compile
```

This command:

- Reads configuration from `pom.xml`.
- Ensures all dependencies and plugins are available.
- Compiles code under `src/main/java/` to `target/classes/`.

To run unit tests:

```
mvn test
```

Maven will:

- Compile test code.
- Execute tests using the configured framework.
- Print a test report in the console.

To build a JAR file:

```
mvn package
```

This command:

- Compiles main and test code.
- Runs tests (unless disabled).
- Packages compiled classes into a JAR under `target/`.

After packaging, the JAR file is located in:

```
target/lga-1.0.jar
```

1.1.2 Running the JAR File

Start the program with:

```
java -jar target/lga-1.0.jar
```

If the program expects command-line arguments:

```
java -jar target/lga-1.0.jar arg1 arg2 arg3
```

These values are accessible in the `main` method via `String[] args`.

1.2 Unit Tests

Unit testing is essential for correctness and maintainability. In this assignment, unit tests must verify that each component of your LAE behaves as intended in isolation, and help detect regressions when code changes.

You are required to write comprehensive unit tests for all methods that contribute to the correctness of your program, including:

- Matrix and vector operations (addition, multiplication, transpose, negation).
- Dimension checks and error-handling logic.
- Task creation and decomposition logic.
- Any helper methods whose correctness is necessary for proper LAE behavior.

Tests should cover standard and edge cases, including invalid inputs and small matrices where results are easy to verify. Submissions without meaningful passing tests will be penalized.

2 Linear Algebra Engine

The program you will implement is called a Linear Algebra Engine (LAE). Its goal is to perform linear algebra computations efficiently by exploiting the parallel nature of the operations and the available processors.

The LAE reads three command-line arguments:

```
<number of threads> <path/to/input/file> <path/to/output/file>
```

For example:

```
java -jar target/lga-1.0.jar \
    10 \
    ./input_files/example1.json \
    ./output_files/example1.out.json
```

2.1 Parsing Input Files

All computations are over the field of real numbers $\mathbb{R}$. The LAE must support:

- $+$ Matrix addition
- $*$ Matrix multiplication
- T Transpose
- $-$ Negation

Unary operators accept exactly one operand; if they receive more, you must treat this as an error. Binary operators must have at least two operands; if more than two are given, evaluate left-associatively, e.g. $*(A, B, C)$ is $(A * B) * C$.

Matrices. A matrix in $\mathbb{R}^{3\times 3}$ is represented as an array of arrays. For example:

$$\begin{bmatrix} 1 & 0 & 2 \\ 3 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{3\times 3}$$

is encoded as:

```
[
    [1, 0, 2],
    [3, 1, 4],
    [0, 0, 1]
]
```

Single Operations. Operations are encoded using an `"operator"` and an `"operands"` array:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```
{
    "operator": "+",
    "operands": [
        [
            [1, 2],
            [3, 4]
        ],
        [
            [5, 6],
            [7, 8]
        ]
    ]
}
```

Nested Operations. More complex computations are formed by nesting operators:

$$\left(\begin{bmatrix} 1 & 1 & 1 & 1 & 5 \\ 1 & 1 & 1 & 1 & 5 \\ 1 & 1 & 1 & 1 & 5 \end{bmatrix} + \begin{bmatrix} 6 & 1 & 1 & 1 & 1 \\ 6 & 1 & 1 & 1 & 1 \\ 6 & 1 & 1 & 1 & 1 \end{bmatrix} \right) \begin{bmatrix} 3 & 2 & 1 \\ 1 & 2 & 3 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \\ 1 & 2 & 3 \end{bmatrix}$$

```json
{
    "operator": "*",
    "operands": [
        {
            "operator": "+",
            "operands": [
                [
                    [1,1,1,1,5],
                    [1,1,1,1,5],
                    [1,1,1,1,5]
                ],
                [
                    [6,1,1,1,1],
                    [6,1,1,1,1],
                    [6,1,1,1,1]
                ]
            ]
        },
        [
            [3,2,1],
            [1,2,3],
            [2,1,3],
            [3,2,1],
            [1,2,3]
        ]
    ]
}
```

2.2 LAE Orchestration Logic

The computation described by the input JSON is represented as a tree of `ComputationNode` instances. Each node has an operator and zero, one, or more operands.

The `LinearAlgebraEngine` (LAE) class orchestrates execution of this tree. Its responsibilities are:

- Iteratively locate the next *resolvable* node: a node whose operands are already concrete matrices.
- Load the operand matrices of that node into two shared matrices M_1 and M_2, which are instances of `SharedMatrix`.
- Decompose the operation into a collection of `Runnable` tasks that work on individual `SharedVector` objects (rows or columns) inside M_1 and M_2.
- Submit all tasks for the current node to the `TiredExecutor` thread pool, and block until all tasks in the batch have completed.
- After completion of the batch, read the result from the left shared matrix (M_1) and attach it back to the corresponding node in the computation tree.

If the root node is itself a matrix (i.e., a leaf node representing a concrete matrix), the LAE writes that matrix directly to the output file without scheduling worker tasks.

2.3 Scheduling Computation for Execution

The LAE breaks each operation into tasks (see Section 2.3.1) and distributes them among worker threads (see Section 2.3.2). The process of breaking down and distributing work is called *scheduling*.

Your goal is to maximize thread utilization while minimizing resource contention and idle time. Implementations that show poor performance compared to a serial implementation, or that cause workers to remain idle unnecessarily, will lose points.

2.3.1 Tasks

A *task* is a unit of work that is ready to be executed by a worker thread. Tasks operate on `SharedVector` and `SharedMatrix` objects and are represented as standard `Runnable` instances.

When breaking a computation into tasks, you must:

- Validate that the operation is legal (e.g., check matrix dimensions before multiplication).
- If an operation is illegal, write an error message to the output file and halt execution.

Wide but Shallow Example.

```
{
    "operator": "+",
    "operands": [
        {
            "operator": "*",
            "operands": [
                [ [1,2], [3,4] ],
                [2, 0]
            ]
        },
        {
            "operator": "*",
            "operands": [
                [ [0,1], [1,0] ],
                [1, 1]
            ]
        },
        {
            "operator": "-",
            "operands": [
                [5, 5],
                [1, 2]
            ]
        },
        {
            "operator": "+",
            "operands": [
                [10, 0, 1],
                [0, 1, 0]
            ]
        }
    ]
}
```

Large Matrices Example.

```
{
    "operator": "*",
    "operands": [
        [
            [1, 2, 3, ..., 50],
            [0, 1, 0, ..., 0],
            [0, 0, 1, ..., 0],
            ...,
            [0, 0, 0, ..., 1]
        ],
        [
            [1, 0, 0, ..., 0],
            [0, 1, 0, ..., 0],
            [0, 0, 1, ..., 0],
            ...,
            [50, 49, 48, ..., 1]
        ]
    ]
}
```

Tasks should typically operate at the granularity of vector operations: for example, computing one row of the result matrix as a sum of one row from M_1 and appropriate row from M_2.

2.3.2 Thread Pool and Worker Threads

When the LAE starts, it creates a thread pool with n worker threads. The pool is implemented by the `TiredExecutor` class, and each worker is an instance of `TiredThread`. The only way to execute work on the pool is by submitting standard `Runnable` objects to the executor.

Thread-pool implementation restriction. For this assignment, the thread pool and worker-scheduling logic must be implemented using **only** Java's built-in monitor primitives: `synchronized` blocks together with the `wait()`, `notify()`, and `notifyAll()` methods on your own objects. The only exception may be locks and blocking data structures provided in the skeleton classes.

The executor maintains:

- An array of worker `TiredThread` instances.
- A `priority-queue`–like structure of *idle* workers, ordered by increasing `fatigue`, so that the least-fatigued worker receives the next task; however, since the fatigue factor is unknown in advance, you may not assume that the fatigue order remains valid between successive operations, therefore this ordering must be recomputed dynamically after each task execution.
- A counter (for example, an `AtomicInteger`) that tracks the number of currently running tasks.

To submit a task, the executor:

- Selects the least-fatigued idle worker from the priority queue (blocking if none are currently idle).
- Places the `Runnable` into the worker's one-element *handoff* queue.

When all computation is complete, the executor shuts down by sending a designated "poison pill" through each worker's handoff queue, instructing it to exit its run loop.

TiredThread. Each `TiredThread`:

- Has a hidden fatigue factor, a random value in the range `0.5–1.5`.
- Measures how much CPU time it spends executing tasks, accumulating this in `timeUsed`.
- Separately tracks idle periods as `timeIdle`.
- Computes its fatigue as

$$\text{fatigue} = \texttt{fatigueFactor} \times \texttt{timeUsed}.$$

You must not change the logic inside `TiredThread` that measures time and computes fatigue.

Responsibilities. The main thread (which owns the `LinearAlgebraEngine`) is responsible for:

- Parsing the input file and writing the output file.
- Building the computation tree of `ComputationNode` objects.
- Loading operand matrices into the two `SharedMatrix` instances M_1 and M_2.
- Creating and submitting `Runnable` tasks to `TiredExecutor`.
- Waiting until all tasks for the current node have completed.
- Shutting down the executor cleanly once the entire computation finishes.

Worker threads are responsible only for:

- Concurrently reading from and writing to `SharedMatrix` and `SharedVector` instances using the locking discipline of `SharedVector`.
- Performing vector-level and row/column-level computations as described by their assigned tasks.

2.3.3 Shared Memory: SharedMatrix and SharedVector

The LAE exposes two shared matrices, M_1 and M_2, as instances of the `SharedMatrix` class. Each `SharedMatrix` manages its data as an array of `SharedVector` objects. The orientation of these vectors (rows or columns) determines how the logical matrix is interpreted.

- **SharedMatrix** does *not* implement any locking. It must not use `synchronized` or explicit lock objects in its methods. It is responsible only for organizing and exposing arrays of `SharedVector` instances and for providing convenience methods (such as accessing a row or column).
- **SharedVector** wraps a `double[]` that stores the numeric data, plus metadata describing whether it represents a row or a column. Each `SharedVector` owns a `ReentrantReadWriteLock` or equivalent that may be used to protect concurrent access to its underlying array when necessary (in which case, the relevant scenario must be explained in a code comment).

Multiple threads may hold the read lock simultaneously to read a `SharedVector`, but writes require exclusive access via the write lock. `SharedVector` must provide in-place methods for basic operations such as: Adding vectors, negating vectors, etc.

Each task submitted to the executor operates by acquiring the appropriate read and/or write locks on the relevant `SharedVector` instances, performing numeric operations, and releasing the locks.

2.3.4 Data Flow and Matrices M_1, M_2

The output of every task must be written to the left matrix M_1, by updating the appropriate `SharedVector` instances in place. Subsequent operations can reuse M_1 and M_2 by unloading their contents back into the computation tree, and then reloading new operand matrices as needed. A single operation can be broken into many tasks.

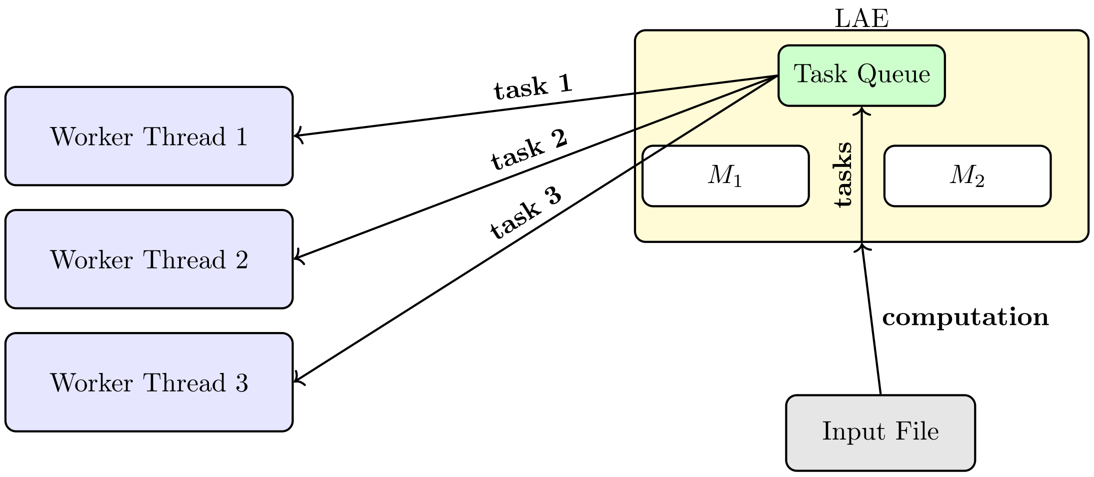

Figure 1: Flow of computation into the LAE, then into tasks and worker threads.

2.3.5 Fairness and Fatigue

At the end of execution, the quality of your scheduling logic is evaluated by how fairly the workload is distributed across workers. Fairness is measured by the sum of squared deviations from the average fatigue:

$$\sum_i (\texttt{fatigue}_i - \overline{\texttt{fatigue}})^2.$$

A lower score means a more uniform distribution of work.

Your executor must use the workers' `getFatigue()` values to drive scheduling decisions so that tasks tend to be assigned to less-fatigued threads first, balancing total work.

2.3.6 Handling Output Files

Output must be formatted in the same JSON format as the input, meaning JSON arrays. Output files will contain one object with either "result" property or "error" property. The result property will only have a JSON array in it that is the result fo the computation. The error property will have a single string containing the error message if one occured. The output file cannot have

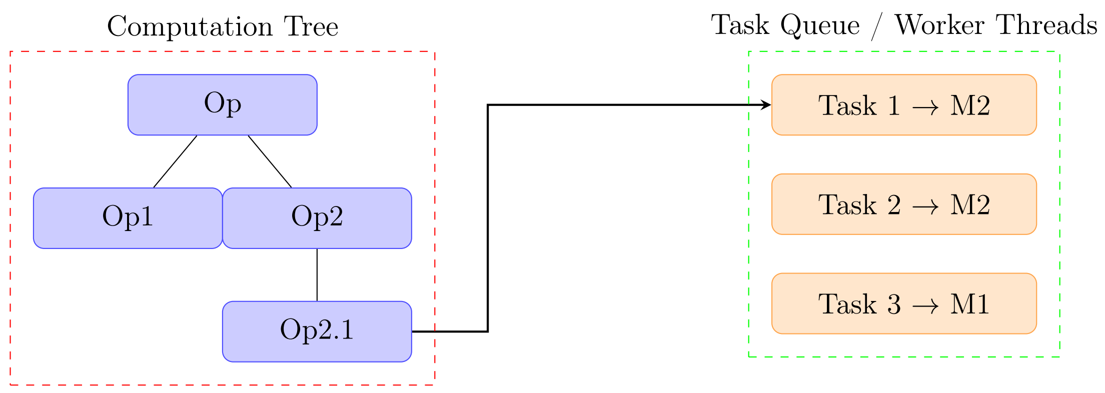

Figure 2: Conversion of a computation tree into tasks with destinations in the shared matrices.

both result and error, put error if the computation failed and put result if it succeeded. For example:

Input file

```
{
    "operator": "+",
    "operands": [
        [
            [1, 2],
            [3, 4]
        ],
        [
            [5, 6],
            [7, 8]
        ]
    ]
}
```

Output file

```
{"result": [
    [6, 8],
    [10, 12]
]}
```

Input file

```
{
    "operator": "*",
    "operands": [
        [
            [1, 2],
            [3, 4]
        ],
        [
            [5, 6, 7],
            [7, 8, 8],
            [6, 6, 1]
        ]
    ]
}
```

Output file

```
{"error": "Illegal operation: dimensions mismatch"}
```

The LAE (main thread) is responsible for writing this output file after the root computation has been resolved.

3 Conclusion

Your task is to write a program that lets users compute linear algebra expressions while utilizing the full parallel computing power available on their system. First, ensure that your linear algebra operations are correct and thoroughly tested. Then, focus on parallelizing them with careful use of fine-grained locks and a fair scheduling policy to avoid unnecessary contention and idle time.

Example input and output files are provided in the skeleton project, but they do not cover all cases. You must add and test your own inputs to validate correctness and performance.

3.1 Grading

Your work will be evaluated on correctness and efficiency.

- **Correctness.** All computations must return correct results. Incorrect results will incur significant penalties.
- **Efficiency.** Implementations that fail to demonstrate effective parallelism, show excessive worker idle time, will lose points.
- **Use of Skeleton.** The skeleton includes several classes. You must use these classes to implement your solution. Do not add fields to these classes. You must implement the provided empty methods without changing their signatures and use them in your design.

3.2 Submission Instructions

When you are finished:

- Create a `.zip` file containing all your source code files, including any skeleton files from the course website that you modified.
- Include a plain text `README` file containing your names, ID numbers, and the submission date.

Upload your `.zip` archive via the course submission system as instructed.