Project 1 – Parallel Matrix Multiplication (Fork/Join)

1. Project Overview

In this project, you will build a parallel matrix multiplication engine in Java using the Fork/Join framework, and compare it with a sequential baseline. You will design an object-oriented solution, implement both versions, and benchmark performance on different matrix sizes.

2. Learning Goals

By the end of this project you should be able to:

- Implement matrix multiplication correctly and efficiently.
- Use the Fork/Join framework (RecursiveTask, ForkJoinPool) to parallelize a computational problem.
- Design an object-oriented architecture for a parallel algorithm.
- Measure and interpret performance metrics (runtime, speedup).

3. Technical & OOP Requirements (All Required)

Core Functional Requirements

1. Sequential Matrix Multiplication: Multiply matrices A (m×n) and B (n×p) to produce C (m×p). Validate dimensions and handle invalid input gracefully.
2. Parallel Matrix Multiplication (Fork/Join): Implement a Fork/Join solution using a divide-and-conquer strategy. Support at least one decomposition strategy (row-range based or block-based). Use a configurable threshold to control when to stop splitting.
3. Benchmarking & Comparison: Implement a benchmark runner that generates random matrices of several sizes (for example, 256×256, 512×512, 1024×1024 depending on your machine), runs sequential and parallel implementations multiple times, and prints average runtime and speedup.

Object-Oriented Design Requirements

Your solution must follow an object-oriented design. A suggested starting structure is:

- class Matrix – holds the 2D data and provides basic operations (such as getRowCount, getColCount).
- interface MatrixMultiplier – defines Matrix multiply(Matrix a, Matrix b).
- class SequentialMatrixMultiplier implements MatrixMultiplier – sequential implementation.
- class ForkJoinMatrixMultiplier implements MatrixMultiplier – uses an inner RecursiveTask (for example, MultiplyTask) and a ForkJoinPool.
- class MatrixBenchmark – generates matrices, runs tests, and prints results.

You may adjust this design, but avoid placing all logic in a single class. Separate data representation, multiplication logic, and benchmarking responsibilities.

4. Suggested Implementation Roadmap

1. Implement Matrix and SequentialMatrixMultiplier, and test correctness with small matrices (2×2, 3×3).
2. Implement ForkJoinMatrixMultiplier with a clear splitting strategy (row-range or block-based) and a configurable threshold.
3. Implement MatrixBenchmark to generate random matrices, run both implementations multiple times, and measure runtime.
4. Collect results (tables or simple plots) and analyze where parallelization helps.

5. Write a short report summarizing design decisions and performance results.

5. Deliverables

- Source Code: Matrix, MatrixMultiplier, SequentialMatrixMultiplier, ForkJoinMatrixMultiplier, MatrixBenchmark.
- Report (approximately 3–4 pages): problem description, design (including OOP structure and decomposition strategy), benchmark results (tables/plots), and a discussion of findings.
- Short Team Presentation (5–8 minutes): architecture overview, performance results, and lessons learned.

Bonus (up to +5 points):

- Up to +3 points – GUI or visualization (for example, UI to choose matrix sizes and run tests, or visualization/logging of task splitting).
- Up to +2 points – Extra features such as implementing and comparing a second decomposition strategy (row-based vs block-based) or simple automatic threshold tuning experiments.

Project 2 – Parallel Merge Sort with Fork/Join

1. Project Overview

In this project, you will implement sequential and parallel merge sort in Java and compare them with Java's built-in sorting methods. You will use the Fork/Join framework for parallelization and structure your solution using object-oriented principles.

2. Learning Goals

By the end of this project you should be able to:

- Implement the merge sort algorithm correctly.
- Transform a recursive algorithm into a Fork/Join solution.
- Compare custom sorting implementations with Arrays.sort and understand performance differences.
- Apply object-oriented design to structure your sorting code.

3. Technical & OOP Requirements (All Required)

Core Functional Requirements

1. Sequential Merge Sort: Implement merge sort for int[], handling edge cases such as empty arrays and already sorted arrays.
2. Parallel Merge Sort (Fork/Join): Implement a RecursiveAction or RecursiveTask that splits the array segment into halves, recursively sorts each half in parallel, and merges the sorted halves. Use a threshold so that small segments are sorted sequentially.
3. Benchmarking & Comparison: Compare sequential merge sort, parallel merge sort, and Arrays.sort (and optionally Arrays.parallelSort). Test on several array sizes (e.g., 10,000; 100,000; 1,000,000) and at least two input patterns (random and reverse sorted).

Object-Oriented Design Requirements

Your design should be modular and object-oriented. A suggested starting structure is:

- interface SortAlgorithm – defines void sort(int[] array).
- class SequentialMergeSort implements SortAlgorithm – sequential merge sort implementation.
- class ParallelMergeSort implements SortAlgorithm – parallel implementation using an inner MergeSortTask that extends RecursiveAction.
- class SortBenchmark – responsible for generating input arrays, running each sort algorithm, and measuring execution time.

You may extend this structure (for example, with ArrayGenerator or input pattern helpers), but avoid putting all logic in a single class.

4. Suggested Implementation Roadmap

1. Implement SequentialMergeSort and test it thoroughly with small arrays and edge cases.
2. Design and implement ParallelMergeSort using Fork/Join, with a reasonable threshold for switching to sequential sort.
3. Implement SortBenchmark to generate arrays for different sizes and patterns, execute each algorithm multiple times, and record average runtime.
4. Collect results into tables or plots and analyze when parallelization helps and when it does not.
5. Prepare a short report and slides summarizing your design and findings.

5. Deliverables

- Source Code: SortAlgorithm, SequentialMergeSort, ParallelMergeSort, SortBenchmark (and any helper classes).
- Report (approximately 3–4 pages): explanation of merge sort, description of your parallel design and threshold choice, benchmark results and analysis.
- Short Team Presentation (5–8 minutes): explanation of the algorithm, design, and performance comparisons.

Bonus (up to +5 points):

- Up to +3 points – GUI or visualization (for example, animating the divide-and-conquer process or visualizing merges).
- Up to +2 points – Extra features such as sorting custom objects with Comparator, or exploring additional input patterns and metrics (e.g., comparison counts).

Project 3 – Monte Carlo Estimation of π with Executors & Futures

1. Project Overview

In this project, you will estimate the value of π (pi) using a Monte Carlo simulation. You will first build a sequential estimator and then a parallel estimator using ExecutorService, Callable, and Future. You will design your solution using object-oriented principles and run experiments to compare runtime and accuracy.

2. Learning Goals

By the end of this project you should be able to:

- Understand and implement a Monte Carlo estimation method.
- Use ExecutorService, Callable, and Future to run tasks in parallel.
- Design a clean object-oriented structure for simulations.
- Analyze accuracy versus number of samples and parallel speedup.

3. Technical & OOP Requirements (All Required)

Core Functional Requirements

1. Sequential π Estimator: Estimate π by generating N random points inside a square and counting how many fall inside the inscribed circle. Return a π estimate for a given N.
2. Parallel π Estimator (ExecutorService + Futures): Use a fixed-size ExecutorService. Divide the total N points into M tasks. Each Callable<Long> simulates its share of points and returns the number of hits. Aggregate results, compute π, and use ThreadLocalRandom.current() inside tasks.
3. Experiments: For several values of N (e.g., 10^5, 10^6, 10^7 as time allows), measure sequential and parallel runtimes and π estimates, and compute absolute error |estimate − Math.PI|. Experiment with at least two different thread pool sizes.

Object-Oriented Design Requirements

Your design should separate configuration, estimation, and experimentation. A suggested starting structure is:

- class SimulationConfig – holds parameters such as long totalPoints, int numTasks, int numThreads.
- interface PiEstimator – defines double estimatePi(SimulationConfig config).
- class SequentialPiEstimator implements PiEstimator – sequential implementation.
- class ParallelPiEstimator implements PiEstimator – parallel implementation using ExecutorService, Callable, and Future.
- class PiExperimentRunner – runs experiments for different configurations and prints or records results.

4. Suggested Implementation Roadmap

1. Implement SequentialPiEstimator and verify that it produces reasonable estimates for π.
2. Implement ParallelPiEstimator using a fixed thread pool, Callable tasks, and Futures to aggregate results.
3. Implement PiExperimentRunner to vary totalPoints and number of threads, measure runtime and error for each configuration.
4. Summarize experimental results in tables or simple plots and discuss how performance and accuracy change.
5. Write a short report describing your method, design, and findings.

5. Deliverables

- Source Code: SimulationConfig, PiEstimator, SequentialPiEstimator, ParallelPiEstimator, PiExperimentRunner.
- Report (approximately 3 pages): explanation of the Monte Carlo method, description of your object-oriented and parallel design, experimental results with analysis.

- Short Team Presentation (5–8 minutes): overview of the method, design, and key results.

6. Grading (20 points + up to 5 bonus points)

Project 4 – Parallel K-Means Clustering with Fork/Join

1. Project Overview

In this project, you will implement the K-means clustering algorithm in Java and then parallelize it using the Fork/Join framework. Your program will cluster data points into K groups, and you will compare sequential and parallel performance and clustering quality.

2. Learning Goals

By the end of this project you should be able to:

- Understand and implement the K-means clustering algorithm.
- Apply Fork/Join to a data-parallel workload.
- Structure a machine learning algorithm using object-oriented principles.
- Compute and interpret clustering quality metrics such as SSE (sum of squared errors).

3. Technical & OOP Requirements (All Required)

Core Functional Requirements

1. Sequential K-Means: Represent data points (at least 2D). Initialize K centroids (for example, random points from the dataset). Iteratively assign each point to the nearest centroid, recompute centroids as the mean of assigned points, and stop after a maximum number of iterations or when centroid movement is below a threshold.
2. Parallel K-Means (Fork/Join): Use Fork/Join to parallelize at least the assignment step. Split the list of points into chunks, and have each task assign its subset of points to the nearest centroids. Optionally parallelize centroid recomputation if time allows.
3. Data Handling & Evaluation: Use either a synthetic 2D dataset (generated in code) or a simple CSV file. Compute SSE (sum of squared errors within clusters) as a clustering quality metric, and compare sequential versus parallel runtime and SSE.

Object-Oriented Design Requirements

Your design should model the domain and algorithm clearly. A suggested starting structure is:

- class Point (or Vector) – represents a data point, for example with double[] coordinates or (x, y) fields.
- class Cluster – holds a centroid and the list (or indices) of assigned points.
- class KMeansConfig – holds parameters such as int k, int maxIterations, double tolerance.
- class KMeansSequential – provides methods to run K-means clustering sequentially.
- class KMeansParallel – provides a parallel implementation using ForkJoinPool and an inner KMeansAssignTask.
- class DataSetLoader – optional helper class if you load data from CSV.
- class KMeansExperiment – runs sequential and parallel versions, measures runtime and SSE, and prints/records results.

4. Suggested Implementation Roadmap

1. Implement Point, Cluster, and KMeansSequential for a small synthetic dataset and verify that clusters appear reasonable.

1. Implement KMeansParallel with a RecursiveAction to parallelize the assignment step across chunks of points.
2. Add SSE computation and implement KMeansExperiment to run sequential and parallel K-means for several values of K and dataset sizes.
3. Collect runtime and SSE results into tables or simple plots and discuss where parallelization helps and how clustering quality behaves.
4. Write a report describing the algorithm, your object-oriented design, and your experimental findings.

5. Deliverables

- Source Code: domain classes (Point, Cluster, etc.), KMeansSequential, KMeansParallel, dataset loading helper (if used), and KMeansExperiment.
- Report (approximately 3–5 pages): explanation of the K-means algorithm, description of your parallel design with Fork/Join, experimental results (runtime and SSE) for different K values and dataset sizes, and discussion of behavior and trade-offs.
- Short Team Presentation (5–8 minutes): algorithm overview, design, key results, and what you learned.

Bonus (up to +5 points):

- Up to +3 points – GUI or visualization (for example, 2D scatter plot of clusters with colors, or animation of centroid movement across iterations).
- Up to +2 points – Extra features such as better initialization (for example, a k-means++-like approach) or running multiple random restarts and choosing the clustering with lowest SSE.

Project 5 – Parallel Sudoku Solver (Backtracking + Task Splitting)

1. Project Overview

In this project, your team will build a Sudoku solver in Java using backtracking, then create a parallel version using the Fork/Join framework. You will compare correctness and performance of the sequential and parallel solvers.

2. Learning Goals

By completing this project, you should be able to:

- Implement a backtracking search to solve a constraint problem (Sudoku).
- Use ForkJoinPool and RecursiveTask/RecursiveAction to parallelize search.
- Design a clear object-oriented model for representing Sudoku boards and solvers.
- Analyze when parallelism helps (and when it does not) in search problems.

3. Technical & OOP Requirements (All Required)

Core Functional Requirements

1. Sudoku Representation: Represent a standard 9×9 Sudoku board. Support reading an initial puzzle (with blanks) and writing the final solved board.
2. Sequential Solver: Implement a recursive backtracking solver that finds an empty cell, tries valid digits (1–9) according to Sudoku rules, and backtracks when a conflict is found. It must correctly solve valid puzzles and may optionally detect unsolvable ones.
3. Parallel Solver (Fork/Join): Use Fork/Join to explore parts of the search tree in parallel. At selected early decision points, create separate tasks for different candidate digits. Limit branching depth to avoid creating too many tiny tasks by using a threshold. Ensure exactly one valid solution is returned for standard puzzles and stop searching once a solution is found.
4. Comparison & Experiments: Run both solvers on a set of puzzles (easy, medium, hard; at least 3–5 puzzles). Measure runtime of sequential versus parallel solvers and summarize results in a table.

Object-Oriented Design Requirements

Your design must be object-oriented. A suggested starting point is:

- class SudokuBoard – 9×9 grid of integers (0 for empty) with methods such as get, set, isValidMove(row, col, value), isComplete(), and clone().
- interface SudokuSolver – defines a method such as boolean solve(SudokuBoard board) or Optional<SudokuBoard> solve(SudokuBoard board).
- class SequentialSudokuSolver implements SudokuSolver – implements the pure backtracking solver.
- class ParallelSudokuSolver implements SudokuSolver – uses ForkJoinPool and an inner class (for example, SolveTask extends RecursiveTask<SudokuBoard>) to parallelize search.
- class SudokuIO – loads puzzles from text files or arrays and prints solutions.
- class SudokuExperiment – runs multiple puzzles with both solvers and records runtimes.

You may adjust names and details, but you must separate the data model (board) from the algorithms (solvers) and avoid putting all logic into a single class.

4. Suggested Implementation Roadmap

1. Board & I/O: Implement SudokuBoard and SudokuIO. Load a puzzle and print it to verify input/output.
2. Sequential Solver: Implement SequentialSudokuSolver using backtracking and test it with known puzzles to confirm correctness.

3. Parallel Solver: Design your splitting strategy (where to fork tasks and when to revert to sequential solving). Implement ParallelSudokuSolver using Fork/Join and tune the threshold.
4. Experiments: Select several puzzles of varying difficulty. Measure and record the time taken by sequential and parallel solvers, and organize the results into a table.
5. Report & Slides: Summarize your algorithm, design, and experimental results. Include at least one figure or table comparing runtimes.

5. Deliverables

- Source Code: SudokuBoard, SudokuSolver interface, SequentialSudokuSolver, ParallelSudokuSolver, SudokuIO, and SudokuExperiment.
- Report (3–5 pages): short explanation of Sudoku rules and backtracking, design and class structure (with a diagram is a plus), parallelization strategy (splitting and thresholds), and experimental results with discussion.
- Team Presentation (5–8 minutes): demo of solving one or two puzzles and explanation of how parallelization works and what you learned.

Bonus (up to +5 points):

- Up to +3 points – GUI Sudoku board that loads puzzles, shows them on screen, and solves them on button click.
- Up to +2 points – Extra features such as a puzzle generator or batch-solving multiple puzzles with aggregate statistics (for example, average solving time).

Project 6 – Parallel Pathfinding on Grid (Multiple Shortest Paths)

1. Project Overview

In this project, your team will implement shortest path search on a 2D grid with obstacles, then extend it to handle many path requests in parallel using Java's concurrency tools.

2. Learning Goals

By completing this project, you should be able to:

- Implement BFS or Dijkstra's algorithm on a grid.
- Use ExecutorService or ForkJoinPool to process multiple independent pathfinding tasks in parallel.
- Model grids, nodes, and paths using a clean object-oriented design.
- Measure and compare throughput in sequential and parallel execution.

3. Technical & OOP Requirements (All Required)

Core Functional Requirements

1. Grid Representation: Represent a 2D grid with walkable cells and blocked cells (walls or obstacles), and allow setting start and goal positions.
2. Sequential Pathfinding: Implement a shortest-path search using BFS (for unweighted grids) or Dijkstra's algorithm (for weighted grids). Return the path as a list of cells, or a clear indication if no path exists.
3. Multiple Path Requests: Define multiple independent PathRequest objects, each with a start and goal on the same grid. Implement a driver that can handle all requests sequentially (one after another) and in parallel using ExecutorService or Fork/Join.
4. Parallel Execution & Comparison: Use ExecutorService with Callable<Path> (recommended) or Fork/Join tasks for each PathRequest. Measure total time to handle a batch of requests sequentially and in parallel (possibly with different thread counts) and compare results.

Object-Oriented Design Requirements

A suggested class structure is:

- class Grid – holds the 2D map (for example, a matrix of cells) and provides methods to query walkability and neighbors.
- class Cell or Node – represents a position (row, column) and may store additional data such as cost if you use weighted paths.
- class Path – represents a list of cells from start to goal.
- class PathRequest – encapsulates a Grid, a start cell, and a goal cell.
- interface PathFinder – defines Path findPath(PathRequest request).
- class BFSPATHFinder or class DijkstraPathFinder – sequential implementation of the chosen algorithm.
- class ParallelPathfindingEngine – uses ExecutorService or ForkJoinPool to run many PathFinder tasks in parallel.
- class PathfindingExperiment – creates multiple PathRequest objects, runs sequential and parallel modes, records times, and prints a summary.

You can adapt this structure, but your design should clearly separate the grid/data model, pathfinding logic, and the parallel orchestration of multiple requests.

4. Suggested Implementation Roadmap

1. Grid & Basic Pathfinding: Implement Grid, Cell/Node, and Path classes. Implement a sequential PathFinder using BFS and test it on small grids with known expected paths.
2. Multiple Requests: Implement PathRequest and PathfindingExperiment. Generate a set of path requests (for example, random start/goal pairs on the same grid).
3. Parallel Engine: Implement ParallelPathfindingEngine using ExecutorService with a fixed thread pool. Each request is handled by a Callable<Path> submitted to the pool.
4. Experiments: For different numbers of requests (for example, 10, 50, 100), measure total time for sequential and parallel processing under different thread counts. Organize results in a table.
5. Report & Slides: Explain the grid representation, algorithm choice (BFS or Dijkstra), concurrency design, and present experimental results with discussion.

5. Deliverables

- Source Code: Grid, Cell/Node, Path, PathRequest, PathFinder implementations, ParallelPathfindingEngine, and PathfindingExperiment.
- Report (3–4 pages): problem description and algorithm, object-oriented design overview (class responsibilities and interactions), parallelization approach (tasks and thread pools), and experimental results with interpretation.
- Team Presentation (5–8 minutes): demo on one or more sample grids and explanation of the algorithm and concurrency approach.

Bonus (up to +5 points):

- Up to +3 points – GUI or visualization that shows the grid, obstacles, start/goal cells, and the computed path (for example, using colors).
- Up to +2 points – Advanced features such as weighted cells with Dijkstra, heuristics (such as A*), or support for different movement rules (for example, diagonal moves).