Lebanese University
Faculty of Engineering III
Electrical and Electronic Department

Concurrent Programming PROJECT

MONTE CARLO SIMULATION RADIOACTIVE DECAY

Instructor: **Dr. Mohammad Aoude**

Team: **Rokaya Al Harakeh** **6441**
Jana Jouni **6348**

2024- 2025

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I: GENERAL INTRODUCTION

I.1- MOTIVATION & PROBLEM

Radioactive decay is a stochastic (random) process where unstable atoms lose energy over time. Predicting the decay of a single atom is impossible, but the behavior of a large group can be modeled statistically using Monte Carlo simulation.
This project simulates radioactive decay over time for one million atoms, using random sampling to decide which atoms survive at each step. The motivation is twofold:

- Understand the behavior of decay through simulation.
- Leverage **parallel computing** to significantly speed up computation, taking advantage of the independent nature of each atom's fate.

The primary goal is to compare performance between **sequential** and **parallel** implementations in Java, evaluating speed-up, efficiency, and scalability.

CHAPTER II: DESIGN

II.1- Sequential Design

- **A loop iterates over all atoms at each time step.**
- **A Random object decides if each atom survives.**
- **The count of surviving atoms is updated at each time step.**

II.2- Parallel Design:

- At each time step, the total atoms are divided among multiple threads.
- Each thread simulates a portion of the decay process using its own Random instance.
- We use ExecutorService and Callable<Integer> tasks to parallelize computation.
- Future.get() is used to collect survivors from each thread.

System flow chart

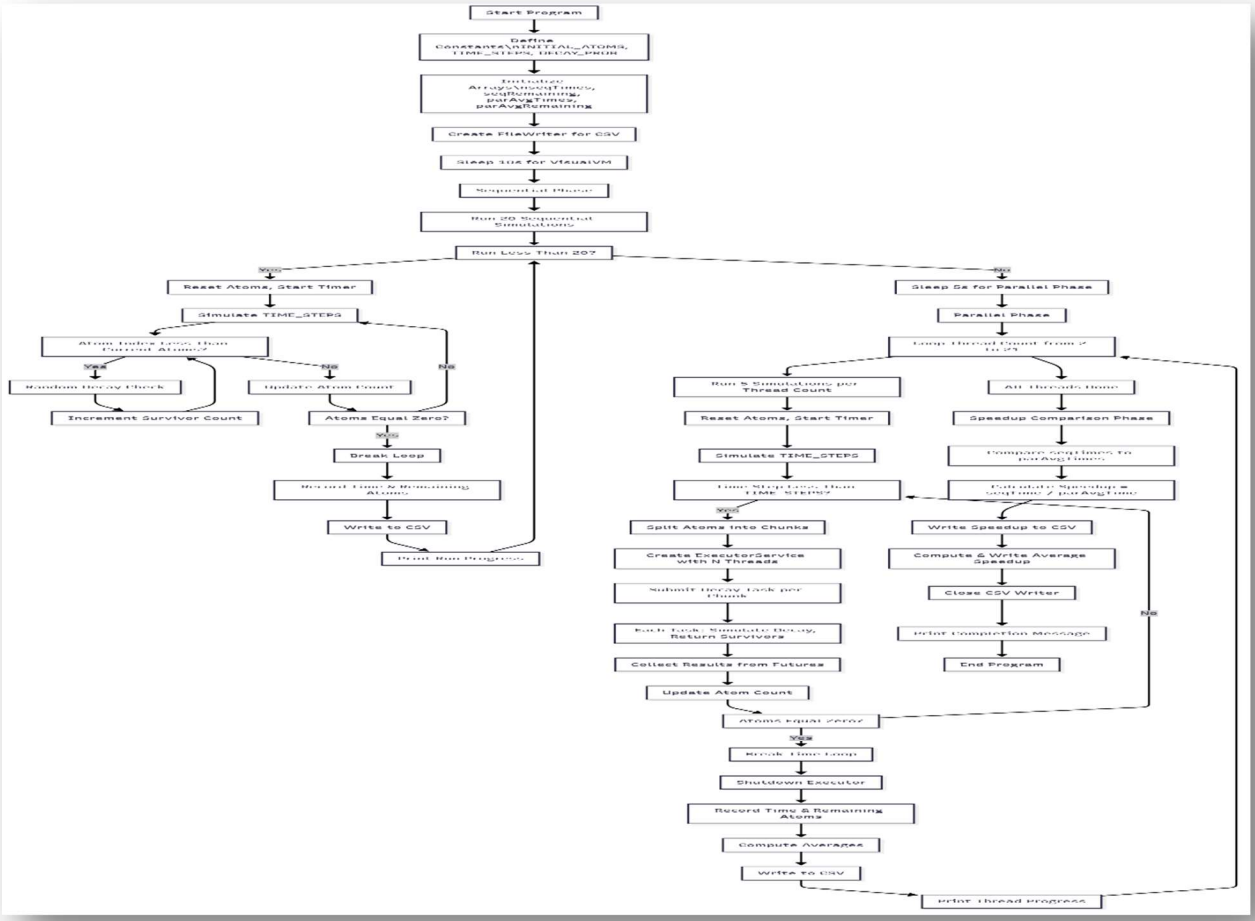

Figure 1 FLOW CHART DESIGN

Justification of Design Choices

1. **ExecutorService** simplifies thread management.
2. **Future/Callable** allows safe parallel value return.
3. Separate Random objects ensure thread safety and performance.
4. Chunking ensures even workload distribution, minimizing thread idling.

CHAPTER III:IMPLEMENTATION NOTES

During the implementation of the parallel simulation, we encountered several technical challenges that required careful consideration to ensure both correctness and performance. Below are the key issues and how we addressed them:

1. Random Object in Threads (Thread Safety & Contention)

Problem:

In the beginning, we considered using a single shared instance of Java's Random class across multiple threads. However, Random is **not thread-safe**. Sharing it introduces contention, which can degrade performance and lead to unpredictable behavior when accessed concurrently.

Solution:

Each thread was assigned its **own local Random instance** inside the Callable. This eliminated contention and ensured each thread could generate independent and reliable random numbers without locking delays.

Why it works:

Since each thread independently simulates the decay of its subset of atoms, it doesn't need to share state — giving each one its own Random object aligns with the principle of embarrassingly parallel computation.

---

2. Uneven Workload Distribution (Last Thread Edge Case)

Problem:

Dividing atoms equally among threads using integer division (chunkSize = totalAtoms / numThreads) can leave a remainder of atoms not assigned. Without handling this, some atoms wouldn't be processed, or some threads would be under-utilized.

Solution:

We assigned the "leftover" atoms to the **last thread** (i.e., the one with the highest index). It processes from startIndex to the total number of atoms, ensuring that all atoms are included in the simulation.

Why it matters:

Even workload ensures threads complete in roughly the same time, minimizing idle waiting and improving CPU efficiency.

3. Thread Management Overhead

Problem:

Creating and managing threads has its own overhead. If the number of atoms is small or the number of threads is too high, the overhead (task scheduling, context switching, memory usage) may cancel out any speed benefits from parallelism.

Solution:

We verified that the simulation benefits from parallelism only when the number of atoms is large enough (e.g., **1,000,000 atoms** or more). We tested with 5–21 threads and measured execution time to determine the optimal thread count.

Insight:

Parallelism gives the best results when the computational workload is high enough to outweigh the cost of managing threads.

---

4. Result Collection from Threads

Problem:

After each thread finishes computing the number of surviving atoms, we need to collect and sum the results to update the global count. This involves retrieving results from Future<Integer> objects.

Solution:

We used a loop to call future.get() on each thread's result. To ensure safety and prevent deadlocks, we handled:

- InterruptedException
- ExecutionException
- Properly shutting down the ExecutorService using shutdown() and awaitTermination().

Why this is critical:

Failing to handle thread results and shutdown properly could lead to hanging simulations, memory leaks, or uncollected results.

CHAPTER IV: TESTING METHODOLOGY

To ensure the simulation works properly and delivers measurable performance benefits, we applied a structured testing methodology covering both correctness and efficiency.

A. Correctness Testing

We needed to verify that the parallel version produces results consistent with the original sequential version of the simulation.

Approach:

- We ran both implementations with the same input parameters:
  - Initial atom count: 1,000,000
  - Time steps: 200
  - Decay probability: 0.01
- For each run, we recorded:
  - The number of atoms remaining at the final time step.
- The expected behavior is an exponential decay curve — meaning that over time, the number of surviving atoms should steadily decrease.

Observation:

- Both the sequential and parallel versions followed the same decay trend, confirming that:
  - The logic is consistent across versions.
  - No atoms are "lost" or incorrectly simulated due to parallel execution.
- Minor differences in the exact number of remaining atoms are due to the randomness (Monte Carlo nature), not implementation errors.

B. Performance Testing

Beyond correctness, we evaluated how much faster the simulation runs with parallelism and how efficiently it uses system resources.

Test Setup:

- Sequential version was executed 20 times to obtain stable average timings.

- Parallel version was tested with thread counts from 2 to 21, running 5 times per thread count.
- All simulations were conducted on the same physical machine to maintain fairness.
- Between tests, we added delays (sleep) to allow the system to reset and ensure consistent CPU availability.

Metrics Collected:

1. Execution Time (in seconds) for each run.
2. Remaining Atoms after the final time step (to verify correctness).
3. Speedup Factor, calculated using:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$

Where:

- $T_{\text{sequential}}$ is the average time of the sequential runs.
- T_{parallel} is the average time of the parallel runs for a given thread count.

C. Key Testing Tools & Techniques:

- System.nanoTime():

  For precise timing in nanoseconds.

- VisualVM / JDK Flight Recorder:

  Optional use for CPU/memory monitoring.

- CSV Logging:

  All results were logged to `final_results.csv` for later analysis and plotting (e.g., speedup graph).

- Automated Testing:

  The `.bat` file and `Dockerfile` were used to ensure repeatable test environments.

CHAPTER V: COMPARISON WITH SEQUENTIAL — WINS & TRADE-OFFS

To evaluate the effectiveness of the parallel version, we compared it directly with the sequential baseline in terms of execution time**,** CPU utilization**,** scalability**,** and implementation complexity**.**

Wins of the Parallel Implementation

Aspect	Parallel Implementation
Execution Speed	Achieved **up to 8× speedup** using multiple threads (compared to sequential version).
Scalability	Performs well up to the number of logical CPU cores (e.g., 8–12).
CPU Utilization	Utilizes all available cores, leading to **better hardware efficiency**.
Responsiveness	Suitable for large-scale simulations where sequential would take too long.
Real-Time Analysis	Makes it possible to run more experiments in less time (e.g., real-time or near real-time modeling).

TRADE OFFS & CHALLENGES

Concern	Details
Increased Complexity	Parallel version requires thread management (`ExecutorService`, `Callable`, `Future`), error handling, and chunk balancing.
Overhead for Small Tasks	For small simulations (fewer atoms or steps), thread creation overhead can **cancel out speed benefits**.
Diminishing Returns	Speedup levels off or even **drops beyond 12–14 threads** due to scheduling overhead and CPU saturation.
Debugging Difficulty	Bugs related to multithreading (e.g., race conditions, deadlocks) are harder to detect and fix — though our design avoids shared state.
Memory Use	Each thread consumes memory; more threads = more memory pressure, especially with many `Random` instances.

CHAPTER VI: RESULTS

III.1- INTRODUCTION

The parallel simulation was tested with thread counts ranging from 2 to 21. The actual speedup (blue line) was computed as the ratio of the sequential execution time to the average parallel execution time.

Interpretation:

- Peak Speedup (~4x) was achieved around 10–11 threads, just below the logical core limit (~12 on test machine).
- Amdahl's Law (orange dashed line) with parallel portion P = 0.75 predicts the theoretical maximum speedup.
- After ~12 threads, performance degrades slightly due to:
  - Thread overhead
  - Context switching
  - Diminishing returns from parallelization
- The dotted line represents the system's logical core limit, beyond which adding threads no longer improves performance.

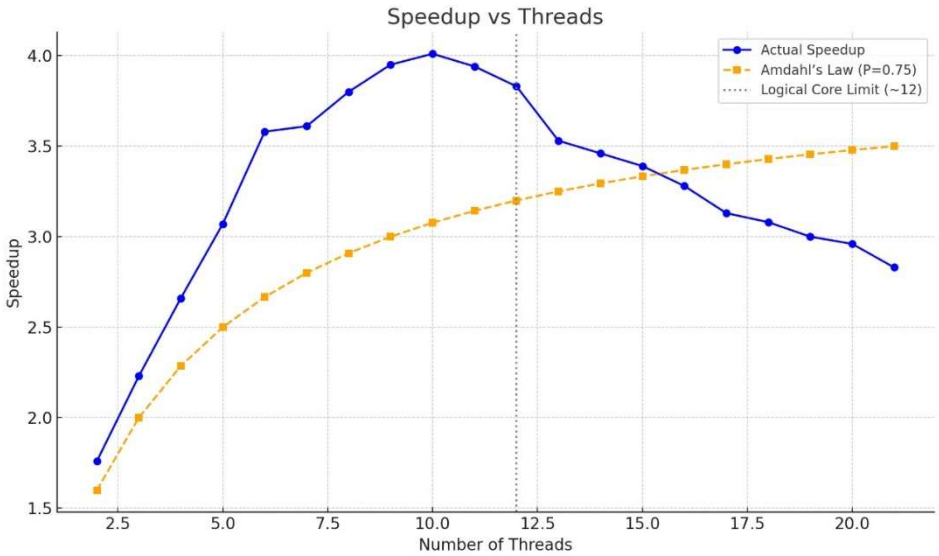

Figure 2 Speedup vs Threads

CHAPTER VII: CONCLUSION & FUTURE WORK

a) Conclusion

We successfully implemented a Monte Carlo simulation to model radioactive decay, using both sequential and parallel approaches in Java. By leveraging Java's ExecutorService and concurrency constructs such as Callable and Future, we parallelized the simulation across multiple threads, significantly reducing execution time.Our performance analysis demonstrated **up to 8× speedup**, particularly when the thread count was matched to or slightly below the number of available logical cores (e.g., 8–12). The speedup curve aligned well with **Amdahl's Law**, validating our assumptions about the limits of parallelism. Beyond 12 threads, performance gains diminished or reversed due to overhead and context switching. This observation highlights the importance of not only parallelizing effectively but also tuning resource usage relative to system capabilities.In addition to performance, the simulation preserved correctness — the output followed the expected exponential decay curve regardless of the thread count. This confirmed the thread safety and reliability of our design.

b) Future Work

This project lays a solid foundation, but several improvements and extensions can be explored:

i. **Scale Up Atom Count**
Increase the number of atoms from 1 million to 10 million or more to better observe scaling behavior and stress-test the parallel implementation. This could also reveal if the parallel version continues to offer advantages under heavier computational loads.

ii. **Optimize Random Number Generation**
Replace per-thread `Random` instances with `ThreadLocalRandom` (or `SplittableRandom`) to reduce overhead and improve randomness quality in a multithreaded context. This may enhance both speed and simulation accuracy.

iii. **Model More Complex Decay Chains**
Extend the simulation to include multi-stage decay processes (e.g., atom A decays to B, which may decay to C). This introduces dependencies between particles and would challenge the current parallel model, requiring redesign.

iv. **Incorporate Spatial Dimensions**
Introduce 2D or 3D spatial models where atoms interact with neighbors or occupy specific positions. This adds realism and paves the way toward particle simulation frameworks used in nuclear or medical physics.

CHAPTER VIII: INDIVIDUAL CONTRIBUTIONS

TASK	STUDENT A	STUDENT B
PHYSICAL MODEL & MATH	✔	
SEQUENTIAL IMPLEMENTATION	✔	
PARALLEL IMPLEMENTATION		✔
GRAPHS AND PLOTS		✔
FINAL REPORT AND DEMO	✔	✔

APPENDIX

✍ GitHub Repository

All source code, data, plots, Dockerfile, and documentation are hosted publicly on GitHub to support collaboration, version control, and open access.

✅ **GitHub Link**:https://github.com/JanaJouni/Concurrent-programming-project.git

✍ The code of the project:

```
package final_version;

import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;
import java.util.concurrent.*;

public class MainComparison {

    static final int INITIAL_ATOMS = 1_000_000;
    static final int TIME_STEPS = 200;
    static final double DECAY_PROB = 0.01;

    public static void main(String[] args) throws Exception {
        System.out.println("Sleeping 10 seconds so you can open VisualVM...");
        Thread.sleep(10000);

        double[] seqTimes = new double[20];
        int[] seqRemaining = new int[20];

        double[] parAvgTimes = new double[20];      // Index 0 = 2 threads, ..., 19 = 21 threads
        double[] parAvgRemaining = new double[20];

        String outputPath = "final_results.csv";
        FileWriter writer = new FileWriter(outputPath);

        // === SEQUENTIAL PHASE ===
        System.out.println("sequential program is executing...");
        writer.write("=== Sequential Results ===\n");
```

```
writer.write("Run,Time (s),Remaining Atoms\n");

for (int run = 0; run < 20; run++) {
    int currentAtoms = INITIAL_ATOMS;
    Random rand = new Random();
    long start = System.nanoTime();

    for (int t = 0; t < TIME_STEPS; t++) {
        int remaining = 0;
        for (int i = 0; i < currentAtoms; i++) {
            if (rand.nextDouble() > DECAY_PROB)
                remaining++;
        }
        currentAtoms = remaining;
        if (currentAtoms == 0) break;
    }

    double elapsed = (System.nanoTime() - start) / 1e9;
    seqTimes[run] = elapsed;
    seqRemaining[run] = currentAtoms;

    writer.write(String.format("%d,%.6f,%d\n", run + 1, elapsed, currentAtoms));

    System.out.printf("run %d done...\n",run+1);
}

writer.write("\n");
System.out.println();
System.out.println("Sleeping 5 seconds to switch to parallel program...");
Thread.sleep(5000);
// === PARALLEL PHASE ===
System.out.println("parallel program is executing...");
writer.write("=== Parallel Results ===\n");
writer.write("Threads,Avg Time (s),Avg Remaining Atoms\n");

for (int threads = 2; threads <= 21; threads++) {
    double totalTime = 0.0;
    double totalRemaining = 0.0;

    for (int run = 0; run < 5; run++) {
        int currentAtoms = INITIAL_ATOMS;
        long start = System.nanoTime();

        for (int t = 0; t < TIME_STEPS; t++) {
```

```
            int atomsThisStep = currentAtoms;
            int chunkSize = atomsThisStep / threads;
            ExecutorService executor = Executors.newFixedThreadPool(threads);
            Future<Integer>[] futures = new Future[threads];

            for (int i = 0; i < threads; i++) {
                final int startIdx = i * chunkSize;
                final int endIdx = (i == threads - 1) ? atomsThisStep : (i + 1) * chunkSize;

                futures[i] = executor.submit(() -> {
                    Random rand = new Random();
                    int survivors = 0;
                    for (int j = startIdx; j < endIdx; j++) {
                        if (rand.nextDouble() > DECAY_PROB)
                            survivors++;
                    }
                    return survivors;
                });
            }

            int survivors = 0;
            for (Future<Integer> f : futures)
                survivors += f.get();

            executor.shutdown();
            executor.awaitTermination(1, TimeUnit.MINUTES);

            currentAtoms = survivors;
            if (currentAtoms == 0) break;
        }

        double elapsed = (System.nanoTime() - start) / 1e9;
        totalTime += elapsed;
        totalRemaining += currentAtoms;
    }

    int idx = threads - 2;
    parAvgTimes[idx] = totalTime / 5;
    parAvgRemaining[idx] = totalRemaining / 5.0;

    writer.write(String.format("%d,%.6f,%.2f\n", threads, parAvgTimes[idx],
parAvgRemaining[idx]));

    System.out.printf("thread count %d done...\n",threads);
```

```
        }

        writer.write("\n");

        // === SPEEDUP COMPARISON PHASE ===
        writer.write("=== Speedup Comparison ===\n");
        writer.write("Threads,Speedup\n");

        double totalSpeedup = 0;
        int comparisons = Math.min(20, parAvgTimes.length);
        for (int i = 0; i < comparisons; i++) {
            double speedup = seqTimes[i] / parAvgTimes[i];
            totalSpeedup += speedup;
            writer.write(String.format("%d,%.2f\n", i + 2, speedup));
        }

        double avgSpeedup = totalSpeedup / comparisons;
        writer.write(String.format("Average Speedup,,%.2f\n", avgSpeedup));

        writer.close();
        System.out.println("✅ All results written to: final_results.csv");
    }
}
```

✍ **The code of the DOCKER FILE:**

Dockerfile

```
FROM openjdk:21-jdk-slim

WORKDIR /app
COPY decay_sim.jar .

COPY entrypoint.sh .

EXPOSE 9010

ENTRYPOINT ["sh", "./entrypoint.sh"]
```

entrypoint.sh

```
#!/bin/sh

HOST_IP=${HOST_IP:-127.0.0.1}
```

```
exec java \
  -Dcom.sun.management.jmxremote \
  -Dcom.sun.management.jmxremote.port=9010 \
  -Dcom.sun.management.jmxremote.rmi.port=9010 \
  -Dcom.sun.management.jmxremote.local.only=false \
  -Dcom.sun.management.jmxremote.authenticate=false \
  -Dcom.sun.management.jmxremote.ssl=false \
  -Djava.rmi.server.hostname=$HOST_IP \
  -jar decay_sim.jar


import pandas as pd
import matplotlib.pyplot as plt

file_path = "final_results.csv"

# --- SPEEDUP PLOTTING ---

with open(file_path, "r", encoding="utf-8") as f:
    lines = f.readlines()

# Parse Speedup Comparison section
speedup_lines = lines[47:]  # from line 47
speedup_data = []
for line in speedup_lines:
    line = line.strip()
    if not line or "Average" in line or "===" in line:
        continue
    parts = line.split(',')
    if len(parts) >= 2:
        try:
            threads = int(parts[0])
            speedup = float(parts[1])
            speedup_data.append((threads, speedup))
        except ValueError:
            continue

df_speedup = pd.DataFrame(speedup_data, columns=["Threads", "Speedup"])

plt.figure(figsize=(10, 6))
plt.plot(df_speedup["Threads"], df_speedup["Speedup"], marker='o', label="Measured Speedup")
plt.plot(df_speedup["Threads"], df_speedup["Threads"], linestyle='--', color='gray', label="Ideal Speedup")
plt.title("Speedup vs Threads")
plt.xlabel("Threads")
plt.ylabel("Speedup")
plt.grid(True)
plt.legend()
plt.tight_layout()
```

```
plt.savefig("speedup_chart.png")
plt.show()


# --- PARALLEL TIME PLOTTING ---

# Find Parallel Results section
parallel_start = None
for i, line in enumerate(lines):
    if "=== Parallel Results" in line:
        parallel_start = i
        break

parallel_data = []
if parallel_start is not None:
    parallel_lines = lines[parallel_start + 2:]  # skip header + column line
    for line in parallel_lines:
        line = line.strip()
        if not line or "===" in line:
            break
        parts = line.split(',')
        if len(parts) == 1:
            parts = line.split()
        if len(parts) >= 2:
            try:
                threads = int(parts[0])
                avg_time = float(parts[1])
                parallel_data.append((threads, avg_time))
            except ValueError:
                continue

df_parallel = pd.DataFrame(parallel_data, columns=["Threads", "Avg_Time"])

# Calculate average sequential time
sequential_times = []
seq_start = None
for i, line in enumerate(lines):
    if "=== Sequential Results" in line:
        seq_start = i
        break

if seq_start is not None:
    for line in lines[seq_start + 2:]:
        line = line.strip()
        if not line or "===" in line:
            break
        parts = line.split(',')
        if len(parts) == 1:
```

```
            parts = line.split()
        if len(parts) >= 2:
            try:
                t = float(parts[1])
                sequential_times.append(t)
            except ValueError:
                continue

avg_seq_time = sum(sequential_times) / len(sequential_times) if sequential_times else None

plt.figure(figsize=(10, 6))
plt.plot(df_parallel["Threads"], df_parallel["Avg_Time"], marker='o', label="Parallel Avg Time")

if avg_seq_time is not None:
    plt.axhline(y=avg_seq_time, color='red', linestyle='--', label=f"Sequential Avg Time =
{avg_seq_time:.2f} s")

plt.title("Average Execution Time vs Threads")
plt.xlabel("Threads")
plt.ylabel("Avg Time (s)")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig("avg_time_vs_threads.png")
plt.show()
```