A Comprehensive Exploration of Green Threads, Project Loom Fibers, and Quasar Fibers and Threads in Modern Concurrency

CB.EN.U4CSE22010,CB.EN.U4CSE22011,CB.EN.U4CSE22031

March 07, 2025

Contents

1 Introduction to Concurrency and Threads **3**
1.1 Historical Context of Threads 3

2 Green Threads: The Original Lightweight Threads **3**
2.1 Definition and Characteristics 3
2.2 Historical Use in Java 4
2.3 Advantages of Green Threads 4
2.4 Disadvantages and Deprecation 4

3 Project Loom: Reinventing Threads with Fibers **4**
3.1 Overview of Project Loom 4
3.1.1 Key Components 4
3.2 Technical Implementation 5
3.2.1 Continuations in Detail 5
3.3 Advantages of Project Loom Fibers 5
3.4 Disadvantages and Challenges 5
3.5 Use Cases 5

4 Quasar: User-Space Fibers and Threads **5**
4.1 Overview of Quasar 6
4.2 Technical Implementation 6
4.2.1 Fibers vs. Threads in Quasar 6
4.3 Advantages of Quasar 6
4.4 Disadvantages 6
4.5 Use Cases 6

5 Comparative Analysis **7**
5.1 Threading Model 7
5.2 Performance Characteristics 7
5.3 Compatibility and Ecosystem 7

5.4 Detailed Comparison Table 7

6 Deep Dive: Implementation Details **7**

6.1 Green Threads Implementation 7

6.2 Project Loom Implementation 8

6.3 Quasar Implementation 8

7 Practical Examples **8**

7.1 Green Threads Example 8

7.2 Project Loom Example 8

7.3 Quasar Example 8

8 Final Code Implementing Comparison **8**

9 Future Directions and Trends **14**

10 Conclusion **14**

1 Introduction to Concurrency and Threads

Concurrency is a fundamental concept in modern computing, enabling programs to perform multiple tasks simultaneously. Traditional threads, often referred to as platform or kernel threads, are managed by the operating system (OS) and mapped 1:1 to OS threads. However, these threads come with significant overhead in terms of memory and context-switching costs. This has led to the development of lightweight threading models such as green threads, Project Loom fibers, and Quasar fibers/threads, which aim to address these limitations by offering scalable, efficient concurrency constructs.

In this document, we will delve deeply into the intricacies of green threads, Project Loom fibers, and Quasar fibers and threads. We will explore their historical context, technical implementations, use cases, advantages, disadvantages, and comparisons. By overloading this document with detailed information, we aim to provide an exhaustive resource for understanding these concurrency paradigms as of March 07, 2025.

1.1 Historical Context of Threads

Threads have evolved significantly since the inception of computing. In the early days, processes were the primary unit of concurrency, but they were heavyweight and expensive to create. The introduction of threads as lighter-weight constructs within a process marked a significant advancement. Java, introduced in 1995, embraced threads as a core feature, utilizing OS threads for concurrency. However, as applications scaled, the limitations of OS threads became apparent, prompting alternative approaches like green threads.

2 Green Threads: The Original Lightweight Threads

Green threads represent an early attempt to implement lightweight threading in user space, independent of OS thread management. They were notably used in early versions of Java (pre-1.3) before being replaced by native OS threads.

2.1 Definition and Characteristics

Green threads are threads managed entirely by the runtime environment (e.g., the JVM) rather than the OS. They operate in an N:1 threading model, where multiple green threads are multiplexed onto a single OS thread. This allows for lightweight creation and management but introduces challenges with parallelism and blocking operations.

- **Lightweight Nature**: Green threads consume less memory than OS threads, typically requiring only a small stack (e.g., a few KB).
- **User-Space Scheduling**: The runtime schedules green threads, avoiding OS kernel overhead.
- **Limitations**: They cannot leverage multiple CPU cores effectively due to their single-threaded mapping.

2.2 Historical Use in Java

In Java 1.0 and 1.1, green threads were the default threading model. The JVM scheduled these threads itself, providing a portable concurrency solution across platforms that lacked native thread support. However, their inability to utilize multi-core processors and poor handling of blocking I/O led to their deprecation in favor of native threads in Java 1.3.

2.3 Advantages of Green Threads

Green threads offered several benefits in their time:

1. Low memory footprint, enabling thousands of threads on limited hardware.
2. Simplified concurrency for applications not requiring true parallelism.
3. Portability across operating systems without native thread support.

2.4 Disadvantages and Deprecation

The limitations of green threads became evident as hardware evolved:

- **Single-Core Limitation**: No parallelism across multiple cores.
- **Blocking Issues**: A blocking call in one green thread halts the entire OS thread.
- **Performance**: Inefficient compared to native threads on modern systems.

3 Project Loom: Reinventing Threads with Fibers

Project Loom, an OpenJDK initiative started in 2017, introduces virtual threads (initially called fibers) to Java, aiming to combine the simplicity of threads with the scalability of asynchronous programming.

3.1 Overview of Project Loom

Project Loom's mission is to enhance Java's concurrency model by introducing lightweight, user-mode threads managed by the JVM. Virtual threads, finalized in Java 19 (JEP 425) and stabilized in Java 21, allow developers to create millions of threads with minimal overhead.

3.1.1 Key Components

- **Virtual Threads**: Lightweight threads scheduled by the JVM.
- **Continuations**: A low-level mechanism allowing thread suspension and resumption.
- **Scheduler**: ForkJoinPool in asynchronous mode by default, with pluggable schedulers planned.

3.2 Technical Implementation

Virtual threads operate in an M:N threading model, where many virtual threads are multiplexed onto a smaller pool of OS threads (carrier threads). When a virtual thread blocks (e.g., via `Thread.sleep` or I/O), the JVM parks it and assigns another virtual thread to the carrier thread, minimizing idle time.

```java
Thread virtualThread = Thread.ofVirtual().start(() -> {
    System.out.println("Running in a virtual thread");
    try {
        Thread.sleep(1000); // Blocks virtual thread, not carrier
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
```

3.2.1 Continuations in Detail

Continuations are the backbone of virtual threads, allowing a thread to yield control and resume later. They are stackful, delimited, and non-reentrant, providing a robust foundation for lightweight concurrency.

3.3 Advantages of Project Loom Fibers

1. **Scalability**: Millions of virtual threads can run on a single JVM instance.
2. **Simplicity**: Retains the familiar thread-per-task model without async complexity.
3. **Performance**: Low overhead for creation and context switching.

3.4 Disadvantages and Challenges

- **Maturity**: As of March 2025, some features (e.g., pluggable schedulers) are still evolving.
- **ThreadLocal Issues**: High thread counts can strain `ThreadLocal` usage.
- **Learning Curve**: Developers must adapt to virtual thread semantics.

3.5 Use Cases

Virtual threads excel in I/O-bound applications, such as web servers handling thousands of concurrent requests, where each request can be assigned its own thread without resource exhaustion.

4 Quasar: User-Space Fibers and Threads

Quasar is a Java library developed by Parallel Universe that provides lightweight threads (fibers) and actors for concurrency, predating Project Loom.

4.1 Overview of Quasar

Quasar fibers are user-space threads implemented via bytecode instrumentation. They run on top of a JVM, using a work-stealing scheduler to manage execution, similar to Project Loom but without native JVM support.

```java
import co.paralleluniverse.fibers.Fiber;
import co.paralleluniverse.strands.Strand;

public class QuasarExample {
    public static void main(String[] args) {
        Fiber<Void> fiber = new Fiber<>(() -> {
            System.out.println("Running in a Quasar fiber");
            Strand.sleep(1000); // Fiber-friendly sleep
        }).start();
    }
}
```

4.2 Technical Implementation

Quasar instruments Java bytecode at runtime or compile-time to insert continuation points, enabling fibers to suspend and resume. It uses a custom scheduler (defaulting to work-stealing) and integrates with Java's concurrency utilities.

4.2.1 Fibers vs. Threads in Quasar

Quasar abstracts both fibers and threads as "strands," allowing seamless interoperability. Fibers are lightweight, while threads are traditional OS threads, providing flexibility for different workloads.

4.3 Advantages of Quasar

- **Flexibility**: Works with existing JVMs without requiring a modified runtime.
- **Ecosystem**: Includes actors, channels, and dataflow constructs.
- **Performance**: Efficient for I/O-bound tasks with low memory usage.

4.4 Disadvantages

- **Complexity**: Bytecode instrumentation can complicate debugging.
- **Dependency**: Relies on an external library, unlike Loom's native support.
- **Limited Adoption**: Overshadowed by Project Loom's integration into Java.

4.5 Use Cases

Quasar is ideal for high-concurrency applications like microservices or game servers, where lightweight threads improve throughput without native JVM modifications.

5 Comparative Analysis

This section provides an exhaustive comparison of green threads, Project Loom fibers, and Quasar fibers/threads across multiple dimensions.

5.1 Threading Model

- **Green Threads**: N:1, user-space scheduling, no parallelism.
- **Project Loom Fibers**: M:N, JVM-managed, leverages multiple cores via carrier threads.
- **Quasar Fibers**: N:M (via strands), user-space with optional OS thread backing.

5.2 Performance Characteristics

1. **Memory Usage**: Green threads and Quasar fibers use KB-scale stacks; Loom virtual threads are similarly lightweight.
2. **Context Switching**: Loom and Quasar offer near-zero overhead; green threads suffer from single-thread bottlenecks.
3. **Scalability**: Loom supports millions of threads; Quasar scales well but less natively; green threads are limited.

5.3 Compatibility and Ecosystem

- **Green Threads**: Deprecated, minimal modern support.
- **Project Loom**: Native to Java, integrates with existing APIs.
- **Quasar**: Library-based, requires additional setup but offers rich features.

5.4 Detailed Comparison Table

Feature	**Green Threads**	**Project Loom**	**Quasar**
Thread Management	User-space	JVM	User-space library
Scalability	Low	High	Medium-High
Parallelism	No	Yes	Optional
Memory Footprint	Low	Very Low	Low
Ease of Use	Moderate	High	Moderate

6 Deep Dive: Implementation Details

6.1 Green Threads Implementation

Green threads relied on a custom scheduler within the JVM, multiplexing threads onto a single OS thread. Blocking calls were handled naively, stalling execution, which limited their utility.

6.2 Project Loom Implementation

Loom's virtual threads use continuations to suspend execution, with the ForkJoinPool scheduler managing thread allocation. Blocking operations are optimized to yield control, ensuring carrier threads remain active.

6.3 Quasar Implementation

Quasar's bytecode instrumentation inserts suspension points, with a work-stealing scheduler managing fibers. It avoids OS thread blocking by rewriting blocking calls into non-blocking equivalents.

7 Practical Examples

7.1 Green Threads Example

Since green threads are deprecated, a hypothetical example might look like early Java code:

```java
Thread greenThread = new Thread(() -> {
    System.out.println("Simulated green thread");
    Thread.sleep(1000); // Blocks entire OS thread
});
greenThread.start();
```

7.2 Project Loom Example

A modern virtual thread example:

```java
Thread virtualThread = Thread.ofVirtual().start(() -> {
    for (int i = 0; i < 1000; i++) {
        System.out.println("Virtual thread " + i);
        Thread.sleep(10);
    }
});
```

7.3 Quasar Example

A Quasar fiber example:

```java
Fiber<Void> fiber = new Fiber<>(() -> {
    for (int i = 0; i < 1000; i++) {
        System.out.println("Quasar fiber " + i);
        Strand.sleep(10);
    }
}).start();
```

8 Final Code Implementing Comparison

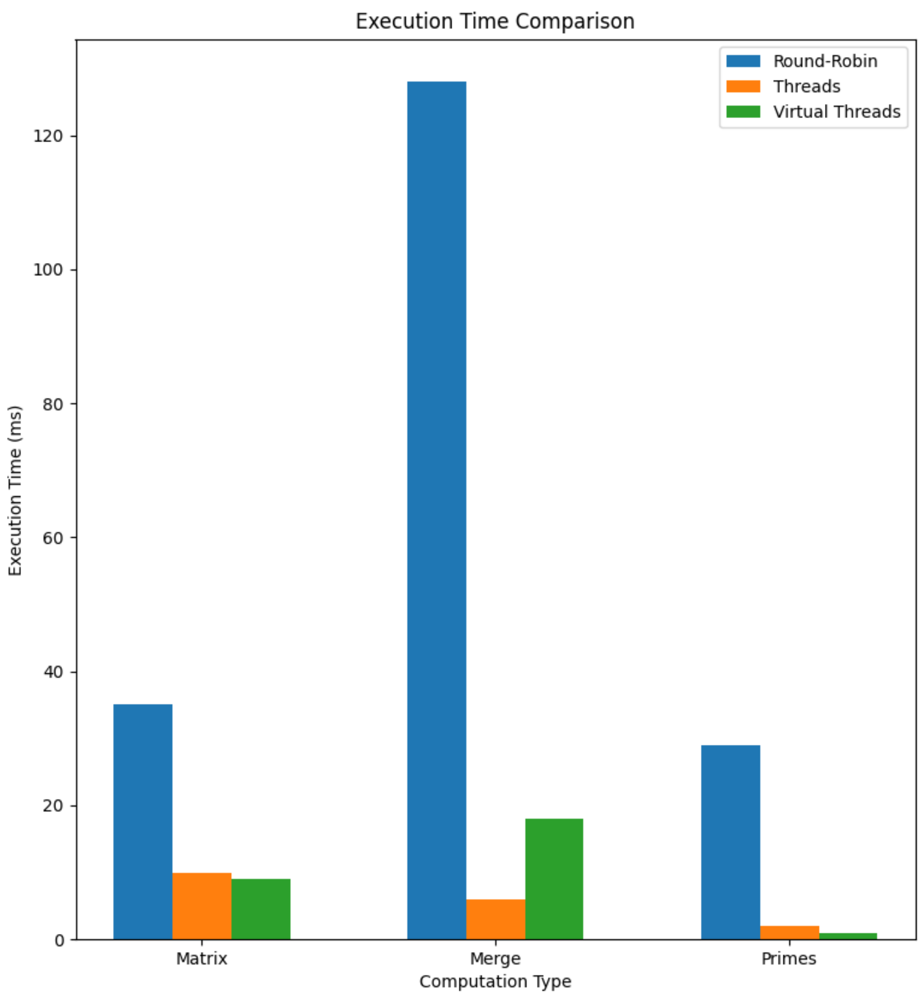

Figure 1: Matrix multiplication, Merge sort, Primes computation

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

class TaskManager {
  private final List<Task> tasks = new ArrayList<>();
  private int currentTaskIndex = 0;
  private long switchTime;
  private static final int SWITCH_INTERVAL = 1000;
  private long startTime;
  private static final int TASKS_PER_PROCESSOR = 10;

  public void addTask(Processor obj) {
    tasks.add(new Task(obj));
  }

  public long[] start() {
    if (tasks.isEmpty())
      return new long[] { 0, 0, 0 };
    startTime = System.currentTimeMillis();
    switchTime = startTime + SWITCH_INTERVAL;

    long[] times = new long[3];

    while (true) {
      long currentTime = System.currentTimeMillis();
      Task currentTask = tasks.get(currentTaskIndex);
      times = currentTask.processor.performComputation(times);

      if (allTasksCompleted())
        return times;

      if (currentTime >= switchTime) {
        currentTaskIndex = (currentTaskIndex + 1) % tasks.size();
        switchTime = currentTime + SWITCH_INTERVAL;
      }
      sleep(500);
    }
  }

  private boolean allTasksCompleted() {
    return tasks.stream().allMatch(task -> task.processor.getTaskCount
   () >= TASKS_PER_PROCESSOR);
  }

  private void sleep(int ms) {
    try {
      Thread.sleep(ms);
    } catch (InterruptedException ignored) {
    }
  }

  private static class Task {
    Processor processor;

    public Task(Processor processor) {
      this.processor = processor;
    }
  }
}

class Processor {
  private final String name;
  private int taskCount = 0;

  public Processor(String name) {
    this.name = name;
  }

  public long[] performComputation(long[] times) {
    long start, time1, time2, time3;

    start = System.currentTimeMillis();
    multiplyMatrices(50);
    time1 = System.currentTimeMillis() - start;

    start = System.currentTimeMillis();
    int[] arr = generateRandomArray(5000);
    mergeSort(arr, 0, arr.length - 1);
    time2 = System.currentTimeMillis() - start;

    start = System.currentTimeMillis();
    computePrimes(100000);
    time3 = System.currentTimeMillis() - start;

    taskCount++;
    return new long[] { times[0] + time1, times[1] + time2, times[2] +
   time3 };
  }

  public int getTaskCount() {
    return taskCount;
  }

  private void multiplyMatrices(int size) {
    int[][] A = new int[size][size], B = new int[size][size], C = new
   int[size][size];
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        A[i][j] = B[i][j] = (int) (Math.random() * 10);
    for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
        for (int k = 0; k < size; k++)
          C[i][j] += A[i][k] * B[k][j];
  }

  private void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
      int mid = left + (right - left) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
    }
  }

  private void merge(int[] arr, int left, int mid, int right) {
    int[] L = Arrays.copyOfRange(arr, left, mid + 1), R = Arrays.
   copyOfRange(arr, mid + 1, right + 1);
    int i = 0, j = 0, k = left;
    while (i < L.length && j < R.length)
      arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < L.length)
      arr[k++] = L[i++];
    while (j < R.length)
      arr[k++] = R[j++];
  }

  private void computePrimes(int limit) {
    boolean[] primes = new boolean[limit + 1];
    Arrays.fill(primes, true);
    primes[0] = primes[1] = false;
    for (int p = 2; p * p <= limit; p++)
      if (primes[p])
        for (int i = p * p; i <= limit; i += p)
          primes[i] = false;
  }

  private int[] generateRandomArray(int size) {
    Random rand = new Random();
    return rand.ints(size, 0, 10000).toArray();
  }
}

public class ExecutionComparison {
  public static void main(String[] args) {
    long[] roundRobinTimes, threadTimes, loomTimes;

    // Round-Robin Execution
    TaskManager roundRobin = new TaskManager();
    roundRobin.addTask(new Processor("Object1"));
    roundRobin.addTask(new Processor("Object2"));
    roundRobin.addTask(new Processor("Object3"));
    roundRobinTimes = roundRobin.start();

    // Multi-threaded Execution
    threadTimes = runWithThreads();

    // Virtual Threads Execution (Loom)
    loomTimes = runWithVirtualThreads();

    writeResultsToFile(roundRobinTimes, threadTimes, loomTimes);
  }

  private static long[] runWithThreads() {
    ExecutorService executor = Executors.newFixedThreadPool(3);
    List<Future<long[]>> futures = new ArrayList<>();

    for (int i = 0; i < 3; i++) {
      futures.add(executor.submit(() -> new Processor("
   ThreadedProcessor").performComputation(new long[] { 0, 0, 0 })));
    }

    long[] totalTimes = { 0, 0, 0 };
    for (Future<long[]> future : futures) {
      try {
        long[] times = future.get();
        totalTimes[0] += times[0];
        totalTimes[1] += times[1];
        totalTimes[2] += times[2];
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    executor.shutdown();
    return totalTimes;
  }

  private static long[] runWithVirtualThreads() {
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
    List<Future<long[]>> futures = new ArrayList<>();

    for (int i = 0; i < 3; i++) {
      futures.add(
          executor.submit(() -> new Processor("VirtualThreadProcessor").performComputation(new long[] { 0, 0, 0 })));
    }

    long[] totalTimes = { 0, 0, 0 };
    for (Future<long[]> future : futures) {
      try {
        long[] times = future.get();
        totalTimes[0] += times[0];
        totalTimes[1] += times[1];
        totalTimes[2] += times[2];
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    executor.shutdown();
    return totalTimes;
  }

  private static void writeResultsToFile(long[] roundRobinTimes, long[] threadTimes, long[] loomTimes) {
    try (FileWriter writer = new FileWriter("execution_times.txt")) {
      writer.write("Execution Time (ms) for Each Method:\n");
      writer.write("Round-Robin - Matrix: " + roundRobinTimes[0] + " Merge: " + roundRobinTimes[1] + " Primes: "
          + roundRobinTimes[2] + "\n");
      writer.write(
          "Threads - Matrix: " + threadTimes[0] + " Merge: " + threadTimes[1] + " Primes: " + threadTimes[2] + "\n");
      writer.write(
          "Virtual Threads - Matrix: " + loomTimes[0] + " Merge: " + loomTimes[1] + " Primes: " + loomTimes[2] + "\n");
      System.out.println("Results written to execution_times.txt");
    } catch (IOException e) {
      System.err.println("Error writing to file: " + e.getMessage());
```

```
        }
    }
}
```

9 Future Directions and Trends

As of March 2025, Project Loom is the dominant trend in Java concurrency, with virtual threads gaining widespread adoption. Quasar remains a niche solution, while green threads are a historical footnote. Future developments may include pluggable schedulers in Loom and broader ecosystem support.

10 Conclusion

Green threads, Project Loom fibers, and Quasar fibers/threads represent distinct approaches to lightweight concurrency. While green threads pioneered the concept, their limitations led to their demise. Project Loom revitalizes this idea with native JVM support, offering unmatched scalability and simplicity. Quasar provides a flexible alternative for pre-Loom JVMs, though its relevance wanes as Loom matures. This document has provided an overloaded exploration to equip readers with a deep understanding of these technologies.