Java supports various concurrency models to manage multiple tasks concurrently and efficiently. Some models include:

1. **Thread-Based Concurrency**: This is the most fundamental model where you create and manage individual threads to perform tasks concurrently. Java provides the class and interface for this purpose.
   Identifiable (nameable)
   Platform thread?
   Virtual thread?
   Sharing memory? – some models may not share memory but rely on fork/join
   Joinable? Yes/no
   Note: 1. And 2. Are very similar, but 2 is simpler to use.

2. **Executor-Based Concurrency**: This model uses the Executor framework, primarily with thread pools, to manage threads more efficiently. `ExecutorService` is a key interface that allows you to submit tasks for execution without directly creating threads.

   Primarily suitable for computational independence for each tasklet.
   Blocking synchronization must be avoided.
   Ditto for model 3 below.

3. **Parallel Stream Concurrency**: Java 8 introduced parallel streams for simplified concurrent programming with collections. This model often leverages the Fork-Join framework and automatically divides tasks for parallel execution. The following example illustrates another method that also achieves concurrency using parallel streams:

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class ParallelStreamConcurrency {

    // Method to simulate a CPU-intensive task (e.g., squaring a number)
    public static long processItem(int number) {
        // Simulate processing time
        try {
            Thread.sleep(1); // Small delay to simulate work
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return (long) number * number;
    }

    public static void main(String[] args) {
        int availableProcessors =
            Runtime.getRuntime().availableProcessors();
        System.out.println("Available processors: " +
            availableProcessors);

        // Parallel stream will likely use this many threads,
```

```
        // leaving one core for other tasks.
        int parallelStreamThreads = availableProcessors;
        System.out.println("Parallel stream will use (approx): " +
            parallelStreamThreads);

        List<Integer> numbers = IntStream.rangeClosed(1,
            1000).boxed().collect(Collectors.toList());

        // Sequential Stream
        long startTimeSequential = System.currentTimeMillis();
        List<Long> squaredNumbersSequential = numbers.stream()
                .map(ParallelStreamConcurrency::processItem)
                .collect(Collectors.toList());
        long endTimeSequential = System.currentTimeMillis();

        System.out.println("Sequential Stream Time: " + (endTimeSequential
        -  startTimeSequential) + " ms");

        // Parallel Stream
        long startTimeParallel = System.currentTimeMillis();
        List<Long> squaredNumbersParallel = numbers.parallelStream()
                .map(ParallelStreamConcurrency::processItem)
                .collect(Collectors.toList());
        long endTimeParallel = System.currentTimeMillis();

        System.out.println("Parallel Stream Time: " + (endTimeParallel -
        startTimeParallel) + " ms");
    }
}
```

Output:

```
Available processors: 8
Parallel stream will use (approx): 8
Sequential Stream Time: 1782 ms
Parallel Stream Time: 283 ms
```

The above example shows that the time used when spreading the threads across multiple cores by using parallel streams (true parallelism), drastically reduces processing time when compared to sequential streams.

Parallel streams in Java, while offering potential performance gains through concurrent execution, introduce several concurrency problems if not used carefully. Some of these problems are as follows:

- Shared Mutable State:

  Multiple threads accessing and modifying a shared variable or object concurrently without proper synchronization can lead to inconsistent or incorrect results (data race).
- Non-deterministic Behavior:
  - Parallel streams don't guarantee the order in which elements are processed, which can cause issues if the logic relies on a specific sequence and the output may vary with each execution.

- For small collections or simple operations, the overhead of parallel processing can outweigh the benefits.
- It's crucial to ensure that the task is sufficiently large to justify the overhead of thread creation and management.
- Incorrectly designed parallel stream operations can lead to deadlocks, where threads are blocked indefinitely, waiting for each other to release resources. This is often related to improper synchronization or resource contention.
- Parallel streams can introduce subtle bugs that are difficult to debug.

- Thread Pool Issues:

  Parallel streams typically utilize the `ForkJoinPool.commonPool()`, which can be a bottleneck if other parts of the application also heavily rely on it. Blocking tasks in one parallel stream can affect other parallel streams or even the entire application.

Avoidance Mechanisms:

- Use immutable objects and collections to avoid accidental modification by multiple threads.
- Instead of modifying shared variables, use techniques like reduction or collecting results into thread-safe data structures.
- For collecting results, prefer using thread-safe collectors like `Collectors.toList()` or `Collectors.toSet()`.
- If a shared state must be modified, use thread-safe classes like `ConcurrentHashMap` or `AtomicLong`.
- Test parallel stream code under various conditions to identify potential concurrency issues and ensure correctness.
- For long-running or resource-intensive tasks, consider creating custom `ForkJoinPools` with specific parallelism levels to avoid impacting the common pool.
- Be aware that `flatMap` can implicitly switch back to a sequential stream, negating the parallelism.
- Always measure the performance of your parallel streams to ensure they are actually providing a benefit.

4. **Non-blocking I/O (NIO) Model**: Java's NIO allows a single thread to manage multiple connections through selectors, which is highly efficient for I/O-bound tasks. NIO offers performance benefits but introduces complexities in programming and debugging. Key problems include: (Pre-Java 7)

- Order of Messages:

  NIO doesn't guarantee the order in which messages are sent or received. This can lead to issues in applications where message sequence is critical.

- Deadlock Potential:

  Using a fixed thread pool with NIO can lead to deadlocks. This occurs when threads are blocked waiting for resources that are being held by other threads in the pool, which are also waiting for resources. This is especially true if the threads are handling I/O operations.

- Out-of-Memory (OOM) Errors:

  High message rates in NIO can lead to OOM errors. If the system cannot process incoming data as quickly as it arrives, the buffers can fill up, leading to memory exhaustion.

- Context Switching and Event Loops:

  In NIO, the application needs to manage context switching between clients, especially when using an event loop. This is because the application is responsible for handling the switching of the context (saving and restoring state) instead of the operating system's thread management. This can increase complexity and the potential for errors.

- Increased I/O Latency:

  Non-blocking I/O can introduce a delay between when data is available in the kernel and when the application actually reads it, potentially increasing I/O latency.

- Need for Careful Data Management:

  NIO requires meticulous attention to data consistency. Multiple reads might be necessary to retrieve a complete message, and developers need to ensure that data is not overwritten before it's fully processed.

- Debugging Complexity:

  Debugging NIO applications can be more challenging than debugging blocking I/O applications. This is because the asynchronous nature of NIO can make it harder to track the flow of execution and identify the root cause of issues.

- Programmatic Complexity:

  Implementing and maintaining non-blocking I/O can be more complex than using blocking I/O. The code needs to handle events, manage buffers, and ensure data consistency, which can increase the complexity of the application.

5. **Asynchronous I/O (NIO2) Model**: Introduced in Java 7, this model fully supports asynchronous operations with callbacks or making it suitable for highly concurrent systems.
6. **Virtual Threads Model**: A newer model, finalized in Java 21, where lightweight, user-mode threads are used to handle a large number of concurrent connections efficiently. They make the traditional Thread-per-Connection model scalable.
7. **Actor Model**: A model where concurrent entities (actors) communicate by sending asynchronous messages, without shared mutable state. This model promotes isolation and helps prevent race conditions.