Parallel Stream in Java

*one of the prominent features of Java 8 (or higher) is Java Parallel Stream. It is meant for utilizing the various cores of the processor. Usually, any Java code that has only one processing stream, where it is sequentially executed. However, by using parallel streams, one can separate the Java code into more than one stream, which is executed in parallel on their separate cores, and the end result is the combination of the individual results. The order in which they are executed is not in our control. Hence, it is suggested to use a parallel stream when the order of execution of individual items does not affect the final result.*

Analysis of Parallel Stream

*For increasing the performance of a program, parallel streams are introduced. However, it is not a guarantee that applying a parallel stream will enhance the result. For example, there can be a scenario where code must be executed in a certain order. There are certain instances in which we need the code to be executed in a certain order, and in such a case, it is required to use sequential streams instead of parallel streams.*

Different Ways to Create Stream

*There are two ways we can create, which are listed below and described later as follows:*

1. *Using the parallel() method on a stream*
2. *Using parallelStream() on a Collection*

factorialsList = list.parallelStream().map(x -> *factorial*(x)).toList();

Program Example

package Streams.StreamWithLambda;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.concurrent.atomic.AtomicInteger;  
import java.util.stream.Stream;  
  
public class ParallelStream {  
 public static void main(String[] args) {  
 // A type of stream that enables parallel processing of elements  
 // Allowing multiple threads to process parts of the stream simultaneously  
 // This can significantly improve performance for large data sets  
 // workload is distributed across multiple threads  
  
  
 long startTime = System.*currentTimeMillis*();  
 List<Integer> list = Stream.*iterate*(1, x -> x + 1).limit(20000).toList();  
 List<Long> factorialsList = list.stream().map(x -> *factorial*(x)).toList();  
// List<Long> factorialsList = list.stream().map(ParallelStream::factorial).toList();  
 long endTime = System.*currentTimeMillis*();  
 System.*out*.println("Time taken with sequential stream: " + (endTime - startTime) + " ms");  
 System.*out*.println("factorialsList with streams" + factorialsList);  
  
  
 startTime = System.*currentTimeMillis*();  
 factorialsList = list.parallelStream().map(x -> *factorial*(x)).toList();  
// factorialsList = list.parallelStream().map(ParallelStream::factorial).toList();  
// factorialsList = list.parallelStream().map(ParallelStream::factorial).sequential().toList();  
 endTime = System.*currentTimeMillis*();  
  
 // for sequential parallel stream  
 factorialsList = list.parallelStream().sequential().map(ParallelStream::*factorial*).toList();  
 System.*out*.println("Time taken with parallel stream: " + (endTime - startTime) + " ms");  
 System.*out*.println("factorialsList with streams" + factorialsList);  
  
 // Parallel streams are most effective for CPU-intensive or large datasets where tasks are independent  
 // They may add overhead for simple tasks or small datasets  
  
 // Cumulative Sum  
 // [1, 2, 3, 4, 5] --> [1, 3, 6, 10, 15]  
// List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
// int sum = 0;  
// List<Integer> cumulativeSum = numbers.stream().map(x -> {  
// int i = sum + x;  
// sum = i;  
// return sum;  
// }  
//  
// ).toList();  
// System.out.println("Expected cumulative sum: [1, 3, 6, 10, 15]");  
// System.out.println("Actual result with parallel stream: " + cumulativeSum);  
  
  
 List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5);  
 AtomicInteger sum = new AtomicInteger(0); // as variable used in map should be declared as final  
 List<Integer> cumulativeSum = numbers.stream().map(sum::addAndGet).toList();  
 System.*out*.println("Expected cumulative sum: [1, 3, 6, 10, 15]");  
 System.*out*.println("Actual result with parallel stream: " + cumulativeSum);  
  
 }  
  
 private static long factorial(int n) {  
 long result = 1;  
 for (int i = 2; i <= n; i++) {  
 result \*= i;  
 }  
 return result;  
 }  
}