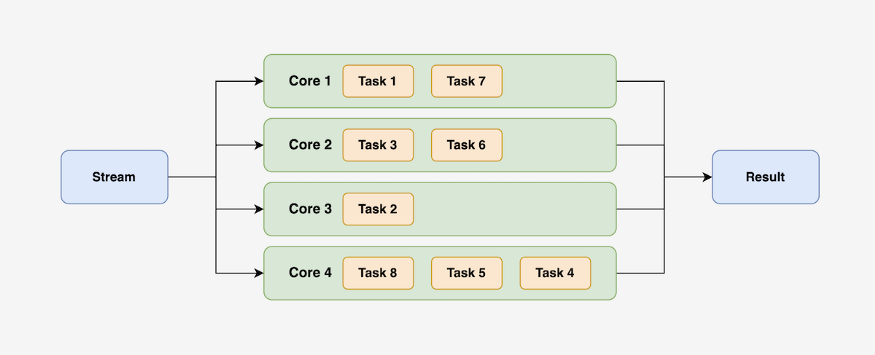
**What is Java Parallel Stream**



Java 8 introduced the concept of parallel streams, which allow for utilizing the power of multi-core machines to improve performance. Parallel streams can be a great tool for performing CPU-intensive tasks more efficiently.

So, then we should use parallel streams everywhere, right?

Not necessarily.

It’s important to consider the various factors and limitations when using parallel streams and to weigh the performance benefits they can offer. In this article, we will explore the different considerations and limitations to keep in mind when using parallel streams, as well as the potential performance gains they can provide.

**Sequential Stream Vs Parallel Stream**

The Stream package was introduced in Java 8 as a powerful tool for processing data collections. It should not be confused with the Java I/O streams, which handle input and output operations.

Features of Stream include:

* Streams act as a wrapper around a data source, such as an array or collection, and provide operations to work with the elements of the data source.
* Stream elements can only be visited once, after which they cannot be accessed again. To visit the same elements again, a new stream must be created.
* Operations performed on a stream do not modify the source data and instead create a new stream with the result of the operation.
* Streams provide a wide range of functions, such as filter, map, find, match, and more, to aid in data processing and manipulation.

**Sequential Stream:**

Sequential stream tasks run in sequence on a single core using a single thread to process. A stream that is not specified as parallel is processed one by one. Thus, Sequential stream does not take advantage of a multi-core processor.

import java.util.Arrays;  
import java.util.stream.IntStream;  
  
public class StreamDemo {  
 public static void main(String[] args) {  
 StreamDemo streamDemo = new StreamDemo();  
  
 streamDemo.sequentialStream();  
 }  
  
 public void sequentialStream() {  
 System.out.println("Sequential Stream");  
 System.out.println("-------------------------------------");  
 this.createAnArrayStream(10)  
 .forEach(this::processElement);  
 }  
  
 public void processElement(int i) {  
 System.out.println("Element " + i + " Processing on " + Thread.currentThread().getName());  
 }  
  
 public IntStream createAnArrayStream(int size) {  
 int[] result = new int[size];  
 for (int i=0;i<size;i++){  
 result[i]=i;  
 }  
 return Arrays.stream(result);  
 }  
}

Output for the above program:

Sequential Stream  
-------------------------------------  
Element 0 Processing on main  
Element 1 Processing on main  
Element 2 Processing on main  
Element 3 Processing on main  
Element 4 Processing on main  
Element 5 Processing on main  
Element 6 Processing on main  
Element 7 Processing on main  
Element 8 Processing on main  
Element 9 Processing on main

**Parallel Stream:**

Parallel streams divide a task into multiple subtasks and utilize multiple cores to process them simultaneously. These streams are specified by calling the parallelStream() method on the Collections interface or the parallel() method on the BaseStream interface.”

*The order of operation is not garunteed in paralleStream but the end result will be same as sequentialStream.*

import java.util.Arrays;  
import java.util.stream.IntStream;  
  
public class StreamDemo {  
 public static void main(String[] args) {  
 StreamDemo streamDemo = new StreamDemo();  
  
 streamDemo.parallelStream();  
 }  
  
 public void parallelStream() {  
 System.out.println("Parallel Stream");  
 System.out.println("-------------------------------------");  
 this.createAnArrayStream(10)  
 .parallel()  
 .forEach(this::processElement);  
 }  
  
 public void processElement(int i) {  
 System.out.println("Element " + i + " Processing on " + Thread.currentThread().getName());  
 }  
  
 public IntStream createAnArrayStream(int size) {  
 int[] result = new int[size];  
 for (int i=0;i<size;i++){  
 result[i]=i;  
 }  
 return Arrays.stream(result);  
 }  
}

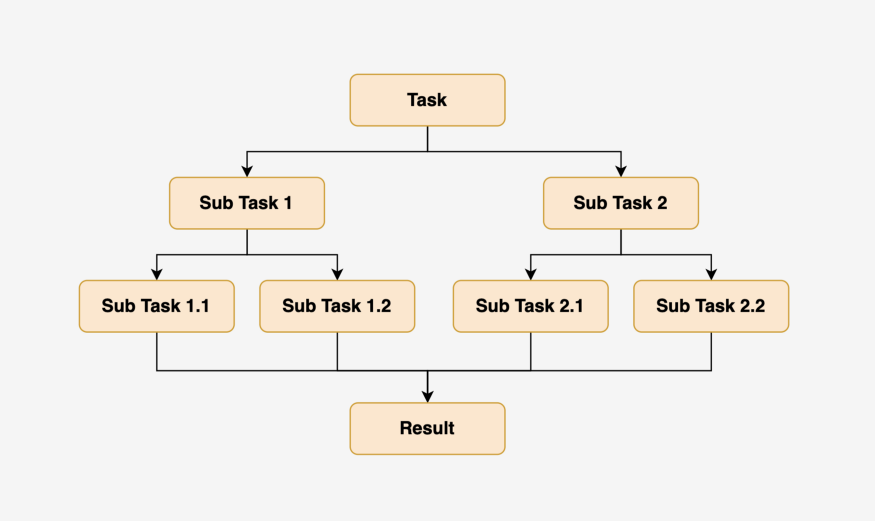
Output for the above program:

Parallel Stream  
-------------------------------------  
Element 6 Processing on main  
Element 5 Processing on main  
Element 8 Processing on main  
Element 9 Processing on main  
Element 7 Processing on ForkJoinPool.commonPool-worker-2  
Element 2 Processing on ForkJoinPool.commonPool-worker-1  
Element 4 Processing on ForkJoinPool.commonPool-worker-2  
Element 1 Processing on main  
Element 0 Processing on ForkJoinPool.commonPool-worker-4  
Element 3 Processing on ForkJoinPool.commonPool-worker-1

**ForkJoinPool in Parallel Stream**

Parallel streams use the ForkJoinPool framework to divide a task into subtasks and process them simultaneously. The commonPool() method is used to create a static instance of ForkJoinPool, which is used internally by the parallel stream to manage the subtasks.

The ForkJoinPool framework breaks a stream into multiple subtasks and processes them simultaneously on different cores. It then combines the results of the subtasks to produce the final output.



**Performance Comparision**

Less see the performance comparison

**Sequential Stream**

This is the code example we will use to evaluate the time it takes for a sequential stream to process.

import java.util.Arrays;  
import java.util.stream.IntStream;  
  
public class StreamDemo {  
 public static void main(String[] args) {  
 StreamDemo streamDemo = new StreamDemo();  
  
 streamDemo.sequentialStream();  
 }  
  
 public void sequentialStream() {  
 System.out.println("Sequential Stream");  
 System.out.println("-------------------------------------");  
 long startTime = System.currentTimeMillis();  
  
 this.createAnArrayStream(1000)  
 .forEach(this::processElement);  
  
 long endTime = System.currentTimeMillis();  
 System.out.println("Time took to process parallel stream: " +  
 convertMilliSecToMinute(endTime - startTime) + " minutes");  
 }  
  
 public long convertMilliSecToMinute(long millis) {  
 return millis/(1000 \* 60);  
 }  
  
 public void processElement(int i) {  
 try {  
 Thread.sleep(1000);  
 } catch (Exception exception) {  
 System.out.println("Exception while processing element " + i + " exception " + exception);  
 }  
 }  
  
 public IntStream createAnArrayStream(int size) {  
 int[] result = new int[size];  
 for (int i=0;i<size;i++){  
 result[i]=i;  
 }  
 return Arrays.stream(result);  
 }  
}

This is the output we get

Sequential Stream  
-------------------------------------  
Time took to process parallel stream: 16 minutes

**Parallel Stream**

This is the code example we will use to evaluate the time it takes for a parallel stream to process.

import java.util.Arrays;  
import java.util.stream.IntStream;  
  
public class StreamDemo {  
 public static void main(String[] args) {  
 StreamDemo streamDemo = new StreamDemo();  
  
 streamDemo.parallelStream();  
 }  
  
 public void parallelStream() {  
 System.out.println("Parallel Stream");  
 System.out.println("-------------------------------------");  
 long startTime = System.currentTimeMillis();  
  
 this.createAnArrayStream(1000)  
 .parallel()  
 .forEach(this::processElement);  
  
 long endTime = System.currentTimeMillis();  
 System.out.println("Time took to process parallel stream: " +  
 convertMilliSecToMinute(endTime - startTime) + " minutes");  
 }  
  
 public long convertMilliSecToMinute(long millis) {  
 return millis/(1000 \* 60);  
 }  
  
 public void processElement(int i) {  
 try {  
 Thread.sleep(1000);  
 } catch (Exception exception) {  
 System.out.println("Exception while processing element " + i + " exception " + exception);  
 }  
 }  
  
 public IntStream createAnArrayStream(int size) {  
 int[] result = new int[size];  
 for (int i=0;i<size;i++){  
 result[i]=i;  
 }  
 return Arrays.stream(result);  
 }  
}

This is the output we get

Parallel Stream  
-------------------------------------  
Time took to process parallel stream: 2 minutes

Here you can see parallel stream runs **14 minutes** faster than a sequential stream.

**Inconsistent Exception In Parallel Stream**

ForkJoinPool divides tasks into subtasks recursively and processes them concurrently. If an exception is thrown on a sub-task, it will be wrapped in a new exception and the original exception will be set as the cause. If an exception is thrown on the main thread, it will be thrown as usual.

Refer to the [Official Comment](https://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/8ec1d42a9bad/src/share/classes/java/util/concurrent/ForkJoinTask.java#l549).

**When not to use Parallel Stream?**

While parallel streams can potentially increase performance, it is not always the best choice for every task. Factors to consider when deciding whether to use a parallel stream include:

* When ordering is important: Parallel streams may not preserve the order of elements in the source data, so if the order of the elements is important for the task, a sequential stream should be used instead.
* When the time taken to process a single element is very small: In this case, the overhead of managing the parallel execution may outweigh any performance gains.
* When the cost of splitting the stream is high: If the data source is not efficient to split, such as a linked list, the cost of splitting it into subtasks may outweigh any performance gains.
* When the NQ product is small: where N stands for the number of source data elements, while Q represents the amount of computation performed per data element. The larger the product of N\*Q, the more likely we are to get a performance boost from parallelization. A rule of thumb is that N should be greater than 10,000 for problems with a trivially small Q, such as summing up numbers
* When the system does not have enough memory or resources to run parallel processing.

It is important to measure and test the performance of the code using parallel streams and compare it with the performance of the sequential streams before deciding which one to use.

The use of a common pool in the ForkJoinPool framework by parallel streams can lead to long-running tasks monopolizing the available threads, resulting in other parallel stream operations being blocked and waiting for thread availability. This is because, The common pool is a shared resource and all parallel tasks use this pool to execute their operations, if one long-running task is using all the threads in the pool, no other task can make use of those threads until the long-running task is done.

**For Example:**Let’s say you have a parallel stream operation that processes a large amount of data, such as filtering and mapping a large list of items. This operation takes a long time to complete and uses up most of the available threads in the common pool.

While this operation is running, another parallel stream operation is started, such as sorting a smaller list of items. However, since the common pool is already being heavily used by the first operation, the second operation has to wait for threads to become available before it can start processing. As a result, the second operation may take longer to complete than it would have if it had been executed by a separate thread pool.

It’s worth noting that we can also create our own thread pool by calling the ForkJoinPool(int parallelism) constructor instead of using the common pool. This would allow us to limit the number of threads used by our parallel stream operations, to avoid blocking other operations that use the common pool.