Java 8

Playtime

Parallel data processing and performance

Chapter 6

Agenda

- Processing data in parallel with parallel Streams
- Performance analysis of parallel Streams
- The Fork/Join framework
- Splitting a Stream of data using a Spliterator

Processing data in parallel

- With the new Stream interface collections of data can be manipulate in a declarative way
- the shift from external to internal iteration enables the native Java library to gain control over how to iterate the elements of a Stream
- This opens the possibility to execute a pipeline of operations on these collections that automatically makes use of the multiple cores on the computer

Parallel processing before Java 7

1. split the data structure containing your data into subparts
2. assign each of these subparts to a different thread
3. synchronise them opportunely to avoid unwanted race conditions
4. wait for the completion of all threads
5. finally reaggregate the partial results
6. quite cumbersome!

Fork/Join framework

- Java 7 introduced the Fork/Join framework
  - to perform these operations more
  - consistently and
  - in a less error prone way
- Still difficult to use
- Later on we will look at this possibility

parallel Streams

- important to know how parallel Streams work internally
- if this aspect is ignored, unexpected results could appear
- the way a parallel Stream gets divided into chunks before processing the different chunks in parallel can be the origin of odd results
- take control of this splitting process by implementing and using custom Spliterator

An example: Calculate sum

```
public class TestParallelOperations {

   public static long sequentialSum(long n) {
       return Stream.iterate(1L, i -> i + 1)
       .limit(n)
       .reduce(Long::sum)
       .get();
   }

   @Test
   public void calculatesumOfFirst100NaturalNumbers() {
       System.out.println(sequentialSum(100));

}
```

Compare code with traditional way

```java
public static long sequentialSum(long n) {
      return Stream.iterate(1L, i -> i + 1)
      .limit(n)
      .reduce(Long::sum)
      .get();
  }
```

```java
public static long iterativeSum(long n) {
       long result = 0;
       for (long i = 1; i <= n; i++) {
              result += i;
       }
       return result;
       }
}
```

What must be done

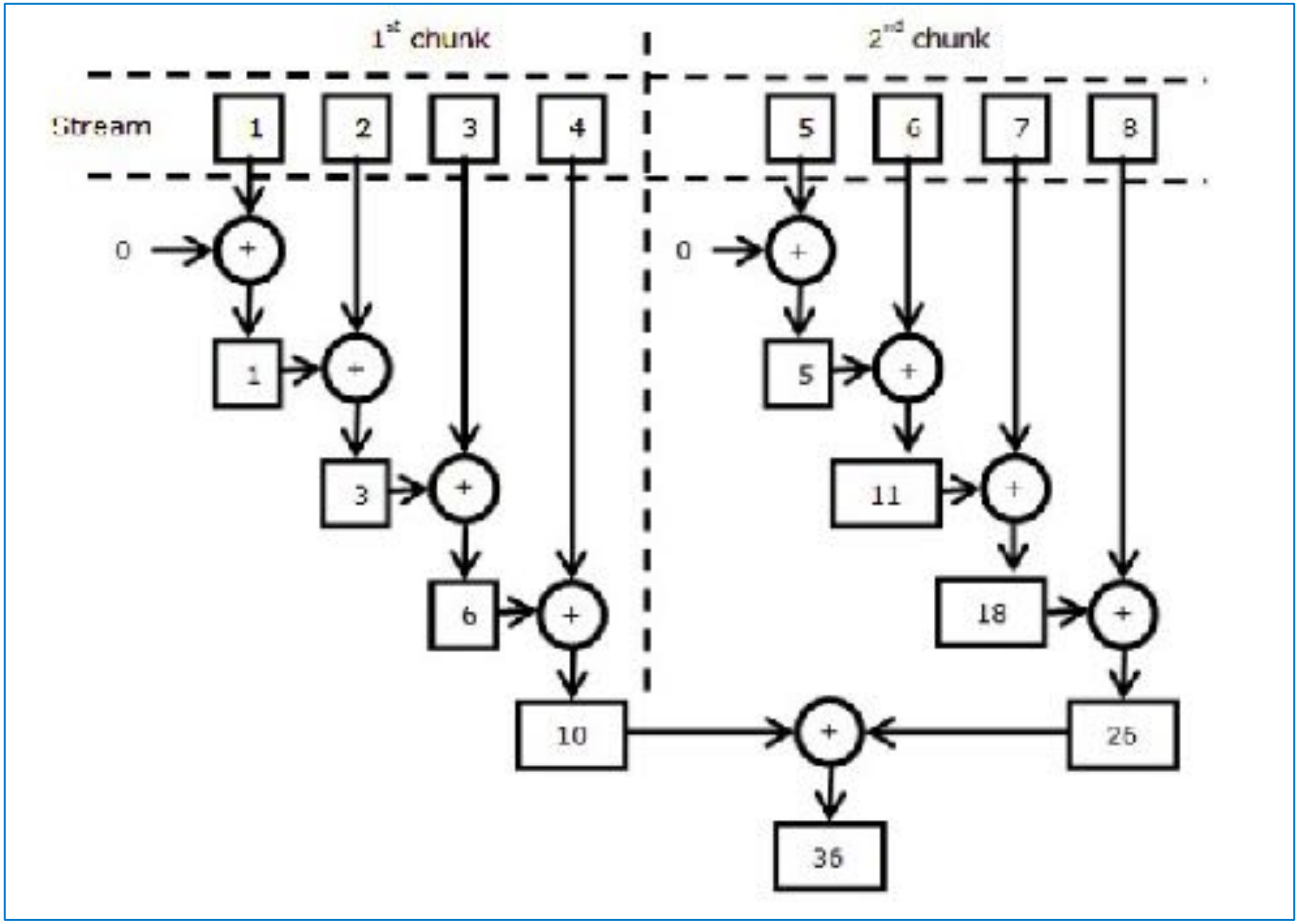
- This operation seems to be a good candidate to leverage parallelization especially for large values of **n**
- However where to start?
- Is synchronising on the result variable a good idea?
- How many threads to use?
- Who does the generation of numbers?
- Who adds them up?
- Stop worrying adopt parallel streams!

Turn stream into a parallel one

```
public static long parallelSum(long n) {

      return Stream.iterate(1L, i -> i + 1)
                   .limit(n)
                   .parallel()
                   .reduce(Long::sum)
                   .get();
}
```

Parallel stream uses chunks



- The stream is internally divided into multiple chunks

What happens when calling parallel?

- Note: calling parallel() on a sequential Stream doesn't imply any concrete transformation on the Stream itself
- Internally, a boolean flag is set to signal that all the operations that follow the parallel() invocation must be run in parallel
- Switching from and to parallel/sequential processing can be achieved by calling parallel() and sequential()

Parallel, sequential just say so

```
public static long m (long n) {
return Stream.iterate(……)
 .limit(n)
 .parallel()
 .filter(….)
 .sequential()
 .map(…)
 .parallel()
 .reduce(….);
}
```

- The filter() and reduce() operations will be performed in parallel,
- The map() operation sequentially

Put the parallel option to the test

- Compare:

```
public static long parallelSum(long n) {
      return Stream.iterate(1L, i -> i + 1)
                   .limit(n)
                   .parallel()
                   .reduce(Long::sum)
                   .get();
}
```

- and:

```
public static long sequentialSum(long n) {
      return Stream.iterate(1L, i -> i + 1)
      .limit(n)
      .reduce(Long::sum)
      .get();
  }
```

The testharness

```
public long measureSumPerf(Function<Long, Long> adder,
                            long n) {
  long fastest = Long.MAX_VALUE;
  for (int i = 0; i < 10; i++) {
     long start = System.nanoTime();
     long sum = adder.apply(n);
     long duration = (System.nanoTime() - start) / 1_000_000;
     System.out.println("Result: " + sum);
     if (duration < fastest)
        fastest = duration;
   }
 return fastest;
}
```

- Each method parallelSum and sequentialSum will be called 10-times, the fastest execution time is printed

The raw data

```java
@Test
public void testDifferentAdders() {
  long time;
  time = measureSumPerf(TestParallelOperations::sequentialSum,10_000_000);
  System.out.println("fastest time of sequentialSum is " + time);
  time=measureSumPerf(TestParallelOperations::parallelSum,10_000_000);
  System.out.println("fastest of parallelSum is " + time);
  time=measureSumPerf(TestParallelOperations::iterativeSum,10_000_000);
  System.out.println("fastest of iterativeSum is " +time);
}
```

```
Fastest of TestParallelOperations::sequentialSum is 131
fastest of TestParallelOperations::parallelSum is 640
fastest of TestParallelOperations::iterativeSum is 6
```

- Very disappointing! The parallelSum is by far the slowest! Why?
- Note: iterative, the old fashioned java way is fast because summing and generation of numbers is combined

Why this unexpected result?

- There are two issues

1. iterate() generates boxed objects, which have to be unboxed to numbers before they can be added
2. iterate() is difficult to divide into independent chunks to execute in parallel

- a mental model that some stream operations are more “parallelizable” than others is handy

Behind the scenes

- The whole list of numbers is not available at the beginning of the reduction process
- The Stream can't partition itself efficiently in chunks to be processed in Parallel
- By flagging the Stream as parallel only overhead of allocating each sum operation on a different thread is added to the sequential processing
- parallel programming can be tricky!

Using more specialized methods

- how can the parallel Stream be used to leverage the multicore processors in an effective way
- Use LongStream.rangeClosed()
  - 2 advantages compared to iterate()

1. It works on primitive long numbers directly so there's no boxing and unboxing overhead
2. It produces ranges of numbers, which can be easily splitted into independent chunks.

What is the (un)box overhead?

```java
public static long rangedSum(long n) {
return LongStream.rangeClosed(1, n)
.reduce(Long::sum)
.getAsLong();
}
```

```
fastest time of rangedSum is 20
fastest time of sequentialSum is 127
```

- Note ranged sum is not yet parallelized
- Apparently there is a huge overhead in boxing and unboxing

Can parallelize help further?

```
public static long parallelRangedSum(long n) {
return LongStream.rangeClosed(1, n)
       .parallel()
       .reduce(Long::sum)
       .getAsLong();
}
```

```
fastest of parallelRangedSum is 3
Finally an improvement over the old “pre” java 8 iteration
fastest of iterativeSum is 6
```

```
public static long iterativeSum(long n) {
  long result = 0;
  for (long i = 1; i <= n; i++) {
       result += i;
  }
  return result;
}
```

Prerequisite for parallel succes

1. using the right data structure
2. and then making it work in parallel
3. Important:

The parallelization process isn't free

- Requires to recursively partition the Stream
- Assign the reduction operation of each sub stream to a different thread
- Reaggregate the results of these operations in a single value
- Moving data between multiple cores can be expensive
  - so it's important that work done in parallel on another core takes longer than the time taken to transfer the data from one core to another

Using parallel Streams correctly

Plausible alternative

```
public class Accumulator {
    public long total = 0;
    public void add(long value) { total += value; }
}

public static long sideEffectSum(long n) {
   Accumulator accumulator = new Accumulator();
   LongStream.rangeClosed(1, n)
             .parallel()
             .forEach(accumulator::add);
   return accumulator.total;
}
```

```
fastest time of parallelSum is 2
```

Times aren't everything

- Note: the swaggering result!

```
Result: 10308601425092
Result: 14207428618068
Result: 2099609531250
Result: 4938067001829
Result: 6676516192027
Result: 4270900971005
Result: 8000289514804
Result: 6459467974634
Result: 7897230220057
Result: 5516525834428

Instead of:
Result: 50000005000000
```

- Shared state isn't save with muliple threads!

Recommendations

- Some qualitative advice that could be useful when considering to use a parallel Stream
- If in doubt, measure
- Watch out for boxing
  - Java 8 includes primitive Streams (IntStream, DoubleStream etc.) for this reason
    - they often outweigh the benefits of parallel Streams

Recommendations

- Some operations naturally perform worse on a parallel Stream
  - i.e. limit and findFirst that rely on the order of the elements
  - i.e. findAny will perform better than findFirst because it is not constrained to operate in the encounter order
- You can always turn an ordered Stream into an unordered one

Recommendations

- For small amount of data, choosing a parallel Stream is almost never a winning decision
- Take into account how well the data structure underlying the Stream decomposes.
  – i.e. ArrayList can be split much more efficiently than a LinkedList, because the first can be evenly divided without traversing it

Recommendations

- The characteristics of a Stream, and how the intermediate operations can modify them
  - a SIZED Stream can be divided into equal parts, and be processed in parallel more effectively
  - a filter operation can throw away an unpredictable number of elements, making the size of the Stream itself unknown
- Consider whether a terminal operation has a cheap or expensive merge step
  - If expensive then re-aggregation of the partial results can outweigh the parallelization

The ForkJoin framework

- Designed to:
  - recursively split a parallelizable task
  - then combine the results of each subtask
- Implements the ExecutorService interface
  - This implementation distributes those subtasks to worker threads
  - The threads are part of a thread pool, called the ForkJoinPool

Working with RecursiveTask

- The ForkJoinPool expects tasks that
  - subclasses RecursiveTask<R> R is the resulttype of the parallelized task
  - or implementations of RecursiveAction if the task returns no result

```
class ParallelTask extends RecursiveTask<T>{

    @Override
    protected T compute(){
        return null;
    }
}
NOTE:
public abstract class RecursiveTask<V> extends ForkJoinTask<V>
```

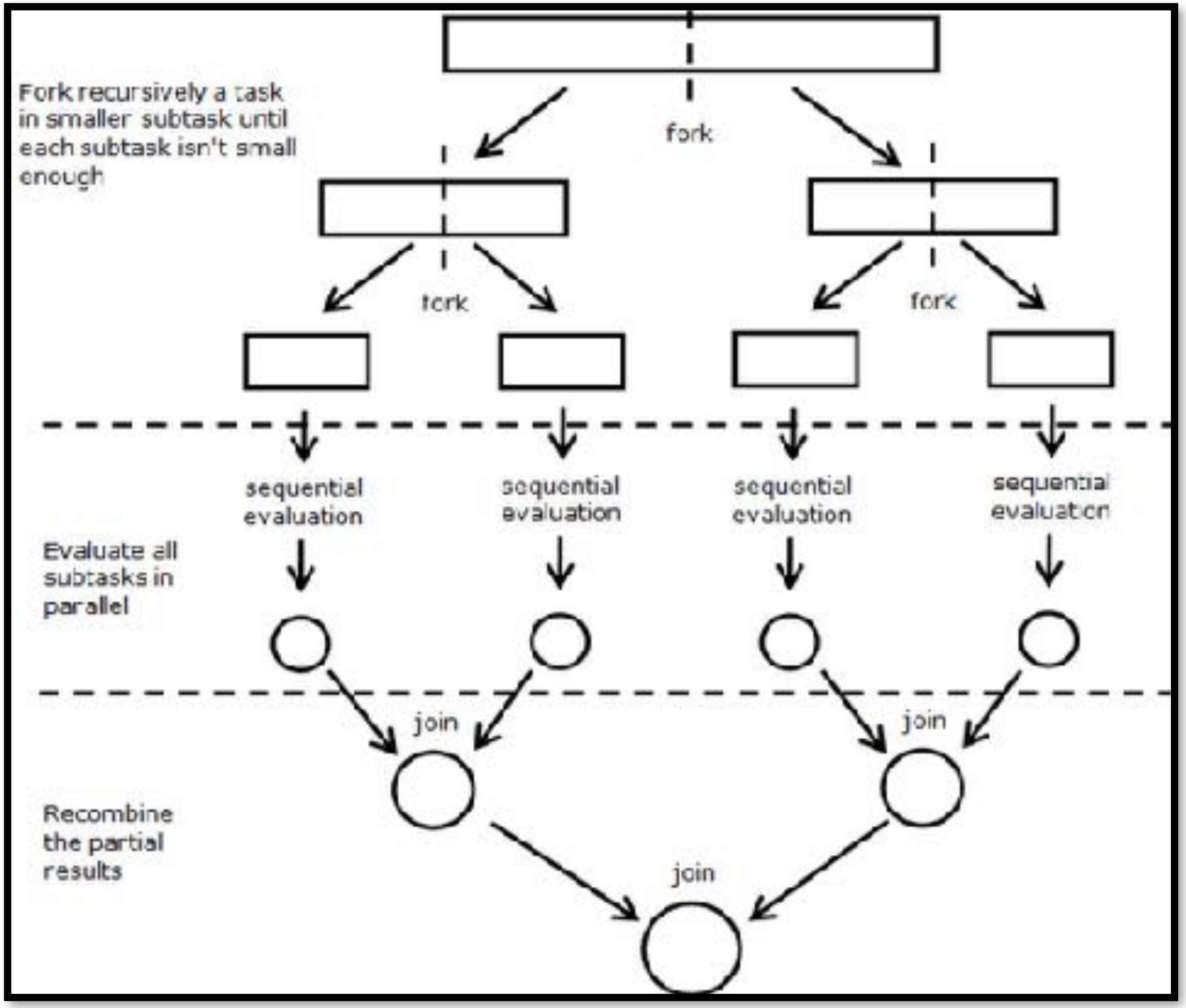
protected abstract T compute()

- Should contain logic for:
  - splitting the task at hand into subtasks
  - and the algorithm to produce the result of a single subtask when it’s no longer convenient to further divide it

General pattern

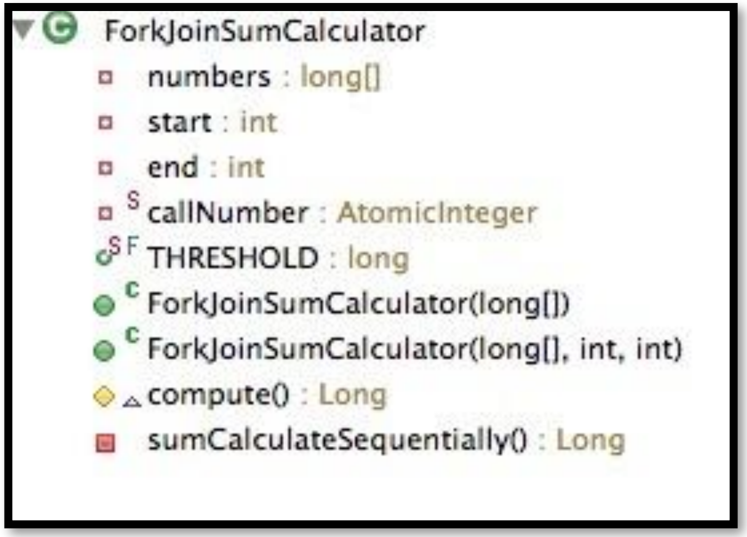
```
if (task is small enough or no longer divisible) {
        compute task sequentially
} else {
        split task in two subtasks
        call this method recursively possibly further splitting each subtask
        wait for the completion of all subtasks
        combine the results of each subtask
}
```

In a picture



Fork recursively a task in smaller subtask until each subtask isn't small enough
fork
fork
fork
sequential evaluation
sequential evaluation
sequential evaluation
sequential evaluation
Evaluate all subtasks in parallel
join
join
Recombine the partial results
join

An example: Summing long array




```java
@Test
public void test() {
  long[] numbers = LongStream.rangeClosed(0L, 1_000_000L)
                             .toArray();
  ForkJoinSumCalculator fjSum =
           new ForkJoinSumCalculator(numbers);
  Long sum = new ForkJoinPool().invoke(fjSum);
  System.out.println(sum);
}
```

Structure of ForkJoinSumCalculator

```
public class ForkJoinSumCalculator extends RecursiveTask<Long>{
     private long [] numbers;
     private int start;
     private int end;
     private static  AtomicInteger callNumber = new AtomicInteger(0);

    public static final long THRESHOLD=10_000 ;

    public ForkJoinSumCalculator(long[] numbers) {
         super();
         this.numbers = numbers;
         start=0;
         end=numbers.length;
    }
    public ForkJoinSumCalculator(long[] numbers, int start, int eind) {
         System.out.printf("callNumber  %4d start =  %d  einde = %d\n",
                                  callNumber.getAndIncrement(),start,eind);
     this.numbers=numbers;
     this.start=start;
     this.end=eind;
}
```

….continued on next page

ForkJoinSumCalculator continued

```
@Override
protected Long compute() {
    int length=end-start;
    if(length<THRESHOLD) {
        return sumCalculateSequentially();
    }
    ForkJoinSumCalculator leftTask =
       new ForkJoinSumCalculator(numbers,start,start+(length/2));
    leftTask.fork();
    ForkJoinSumCalculator rightTask =
       new ForkJoinSumCalculator(numbers, start + length/2, end);
    Long rightResult = rightTask.compute();
    Long leftResult = leftTask.join();
    return leftResult + rightResult;
}
private Long sumCalculateSequentially() {
    long sum = 0;
    for (int i = start; i < end; i++) {
    sum += numbers[i];
    }
    return sum;
}
```

Three interesting calls

```
protected Long compute() {
   …
   ForkJoinSumCalculator leftTask =
   new ForkJoinSumCalculator(arguments..);
   //1 Why a fork?
   leftTask.fork();
   ForkJoinSumCalculator rightTask =
   new ForkJoinSumCalculator(arguments);
   //2 Why a compute?
   Long rightResult = rightTask.compute();
   //3 Why a join?
   Long leftResult = leftTask.join();
   return leftResult + rightResult;
}
```

Reminder of the code

```
protected Long compute() {
  int length=end-start;
  if(length<THRESHOLD) {
     return sumCalculateSequentially();
  }
  ForkJoinSumCalculator leftTask =
       new ForkJoinSumCalculator(numbers,start,start+(length/2));
  leftTask.fork();
  ForkJoinSumCalculator rightTask =
       new ForkJoinSumCalculator(numbers, start + length/2, end);
  Long rightResult = rightTask.compute();
  Long leftResult = leftTask.join();
  return leftResult + rightResult;
}
private Long sumCalculateSequentially() {
  long sum = 0;
  for (int i = start; i < end; i++) {
         sum += numbers[i];
  }
  return sum;
}
```

Done some experiments

- Played with different scenario's

1. leftTask.fork, rightTask.compute,leftTask.join
2. leftTask.fork, rightTask.fork, leftTask.join, rightTask.join
3. No big difference in results were observed
4. Note however that we have a 100 task evenly divided over 8 cores
5. Also using a tenfold bigger THRESHOLD did not result in a noticeable difference
6. To be continued