Enhancing Java Streams API with PowerList Computation

Virginia Niculescu, Darius Bufnea, Adrian Sterca
Faculty of Mathematics and Computer Science, Babeș-Bolyai University, Romania
Summary

- JPLF framework
- Motivation and Goals
- Java Streams
- Powerlist and PList Theories
- Adapting Java Streams for PowerList functions
- Polynomial value computation
- Conclusions
JPLF framework for parallel programming

- JPLF
  - Java framework for parallel programming
    - including High Performance Computing
  - start from a formal base -- *PowerList theory* – in order to assure correctness
  - allow efficient executions on both shared memory and distributed memory systems

- divide-and-conquer skeleton

![Java Parallel List Framework](image)
Motivation and Goals

- **Goals**
  - The purpose of this work is to investigate if the computations defined based on *PowerList and PList* theories could be specified using the *Java Streams* infrastructure, and to which extent.

- *Java Stream API is very popular nowadays, and so if some powerful parallel programming skeletons could be adopted to be executed inside this API, then they will be easily popularized while the expressiveness of the Java Stream API increases.*
Java streams

- A Java stream is a sequence of objects.
- Usually it has a data source and a destination where they are transmitted.
- A stream is not a repository, but it operates on a data source such as an array or a collection.
- More formally, we may consider the streams as being monads, which is a structure that represents computations defined as sequences of steps.
- In Java Stream API there are many already implemented operations that could be sent to a stream, the most common being *map, filter, reduce, collect*.
- **Parallel computation is possible for parallel streams.**
Powerlist Theory

Powerlist Theory was introduced by J. Misra in 1994.

- PowerLists are homogeneously typed sequences of elements.
- The size of a PowerList is a power of two.
- A **singleton** is a PowerList containing a single element - notation \([v]\)
- Two PowerLists of the same size and same type for elements are **similar**.
- **Two constructors** exist to combine two similar PowerLists:
  - \(( | )\) the operator **tie** yields a PowerList containing the elements of \(p\) followed by elements of \(q\);
  - \(( # )\) the operator **zip** returns a PowerList containing alternatively the elements of \(p\) and \(q\).
- The functions on Powerlists are **recursive functions** defined based on a **structural induction**.
  - the **base case** considers the singleton list
  - the **recursive case** may use either \(( | )\) or \(( # )\) for decomposing/composing
Enhancing Java Streams API with PowerList Computation

Functions examples

- Map - applies a scalar function $f_s$ to each element of a PowerList
  
  \[
  \begin{align*}
  \text{map}(f_s, [v]) &= [f_s(v)] \\
  \text{map}(f_s, pl_1|pl_2) &= \text{map}(f_s, pl_1)|\text{map}(f, pl_2)
  \end{align*}
  \]

- Reduce – for an associative operator $\oplus$

  \[
  \begin{align*}
  \text{red}(\oplus, [v]) &= [v] \\
  \text{red}(\oplus, pl_1|pl_2) &= \text{red}(\oplus, pl_1) \oplus \text{red}(\oplus, pl_2)
  \end{align*}
  \]

- Fast Fourier Transform

  \[
  \begin{align*}
  \text{fft}([x]) &= [x] \\
  \text{fft}(p\oplus q) &= (F_p + u \times F_q) |(F_p - u \times F_q)
  \end{align*}
  \]

where

- $F_p = \text{fft}(p)$, $F_q = \text{fft}(q)$, $u = \text{powers}(p)$,
- $+, -$ are the correspondent associative binary operators extended to PowerLists.
- $\text{powers}(p)$ is the PowerList $[w^0, w^1, ..., w^{|p|-1}]$ where $|p|$ denotes the size of $p$ and $w$ denotes the 2nd principal root of 1.
The PList extension of PowerLists (Kornerup, 1997) lifts the restriction of the length being a power of 2.

PLists are also defined based on tie and zip operators.

For PLists, both tie and zip are generalized to take as arguments an arbitrary number of similar PLists.

The map function on PLists can then be defined as:

\[
\begin{align*}
map(f, [], [a]) &= [f(a)] \\
map(f, n::l, p_i) &= \bigg|_{i \in n} \bigg| \bigg| \bigg|_{p_i}
\end{align*}
\]

where \( n::l \) denotes the list with head element \( n \), and with tail \( l \) (a list) and \( |_{i \in n} p_i \) is the generalized tie applied on \( n \) lists \( p_i \)
PowerList functions execution

express *Divide-and Conquer* computation

1) *Descending/splitting* phase that considers the operations needed to split the list arguments, and additional operations, if they exist.

2) *Leaf* phase that considers the operations executed on singletons.

3) *Ascending/combining* phase that considers the operations needed to combine the list arguments, and additional operations, if they exist.
Classes of functions

1. splitting $\equiv$ data decomposition
The class of functions for which the splitting phase needs only data decomposition. Examples: map, reduce, fft
   - much easier to implement!

2. splitting $\not\equiv$ data decomposition
The class of functions for which the splitting phase needs also additional computation besides the data decomposition.
Example: $f (p|q) = f (p+q) | f(p-q)$
   or
   “polynomial value function”
Example: polynomial value

- The functions from the second class involve some additional computations on the sublists obtained at each step.

- Example: computing the value of a polynomial in a given point:

\[
vp([a], x) = [a] \\
vp((p#q), x) = vp(p, x^2) + (x \cdot vp(q, x^2))
\]

where \((x \cdot p)\) means that every element of the list \(p\) is multiplied with \(x\) (it could be considered a map function)
Adaptation strategy

- use *Spliterator* for the splitting phase

- use *collect* function for leaf and combining phases
The *collect* template method

```java
collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
```

- performs a mutable reduction operation on the elements of the calling stream

- **mutable reduction** => the reduced value is a mutable result container, and elements are incorporated by updating the state of the result rather than by replacing the result.

  - **supplier**: a function that creates a new mutable result container;
    - in a parallel execution, this function may be called multiple times and must return a fresh value each time.
  
  - **accumulator**: fold an element into a result container.

  - **combiner**: accepts two partial result containers and merges them,
    - fold the elements from the second result container into the first result container
Enhancing Java Streams API with PowerList Computation

Example

List<String> list = Arrays.asList("Ana", "Lia", "Dan");
String result = list.parallelStream()
    .collect(
        StringBuilder::new, // the supplier
        (response, element) ->
        response.append(" ").append(element),
        // the accumulator
        (response1, response2) ->
        response1.append(""," ").append(response2.toString()))
    // the combiner
    .toString();

- the words in a given list are concatenated, including a comma between each pair of two words

- **combiner** function is specific to the parallel execution of the **collect** method:
  - if the stream hadn’t been parallel, the combiner would not be used and so
    - the comma wouldn’t be added
The function `collect` has also a definition that receives as an argument a `Collector`.

`Collector` is an interface that provides a wrapper for the `supplier`, `accumulator`, and `combiner` objects.

`Collector<T,A,R>`

where the type parameters have the following significance:

- T - the type of input elements;
- A - the mutable accumulation type;
- R - the result type.

This variant is more convenient to be used for PowerList functions, because a specific function – `APowerFunction`, could implements the `Collector` interface, and then for its execution we just need to invoke the `collect` function with an argument, which is an instance of that class.
Spliterator

- The parallel computation of the parallel streams is directed by the existence of a special type of iterator – Spliterator – and by the usage of the ForkJoinPool executor.

- ForkJoinPool executor is specialized in the computation of the recursive tasks, and so it is appropriate for the divide-and-conquer computational model.

- The Spliterator interface defines several methods, such that
  - trySplit operation that partitions off some of its elements as another Spliterator
  - by default, the partitioning is performed linearly, in “segments”, which is somehow similar to the operator tie from PowerLists.
In order to control the partitioning, new implementations for the `Spliterator` interface are provided

- `TieSpliterator`
- `ZipSpliterator`

`trySplit` defined correspondingly
Tie and Zip constructors

=> we need to provide operations for the *tie and zip constructor operators*

- This could be achieved by defining a class *PowerList* that extends any *RandomAccess* collection (e.g. *ArrayList*)
- the class provides the methods:
  - *tieAll*
  - *zipAll*
- *zip* operation: when executing the *collect* function, the stream is decomposed using *ZipSpliterator* instance, and then recomposed based on the function *zipAll* of the class *PowerList*.
- *tie* operation: similar
General approach

- Define a class that implements Collector -> ACollector
- Provide definition for the three methods: supplier, accumulator, combiner
- Create the needed specialized Spliterator -> aSpliterator
- Define the stream based on this aSpliterator
- Call the collect method using an instance of the ACollector

⇒ this works very easy for the functions that belongs to the first class,
  ⇒ZipSpliterator or TieSpliterator can be used
But
  ⇒for the functions belonging to the second class, additional adjustments needed:
    ⇒define new specialization of the Spliterator
    ⇒connect the splitting phase with the other phases
Example: computing the value of a polynomial in a point
very simple parallel PowerList definition, but
involves some operations at the splitting phase

class PolynomialValue implements
Collector<Double, PolynomialValue, PolynomialValue>
{
  private double x; // the given point
  private double val = 0; // the result
  private int x_degree = 1;

  // constructors

  @Override
  public Supplier<PolynomialValue> supplier () {
    return () -> {
      return new PolynomialValue(this);
    };
  }

  @Override
  public BiConsumer<PolynomialValue, Double> accumulator () {
    return (pv1, d) -> {
      pv1.val =
      pv1.val*Math.pow(pv1.x,pv1.x_degree) + d;
    };
  }

  @Override
  public BinaryOperator<PolynomialValue> combiner () {
    return (pv1, pv2) -> {
      pv1.x_degree/=2;
      pv1.val = pv1.val*
      Math.pow(this.x, pv1.x_degree)
      +pv2.val;
      return pv1;
    };
  }
}
Specialization of ZipSpliterator

```java
class PZipSpliterator
    extends SpliteratorPower2.ZipSpliterator<Double>
    
    protected int x_degree=1; //local attribute
    public PZipSpliterator(
        List<Double> list, int start, int end, int
        incr, int x_degree) {
        super(list,start,end, incr);
        this.x_degree = x_degree;
    }
    public PZipSpliterator trySplit() {
        int lo = start;
        int step = incr;
        if (start + step <= end) {
            x_degree*=2; // !!!!! updating the exponent
            synchronized(PolynomialValue.this)
            {
                if (PolynomialValue.this.x_degree < x_degree
                )
                    PolynomialValue.this.x_degree = x_degree;
            }
            incr *= 2;
            start += step;
            return PolynomialValue.this.new
                PZipSpliterator(list, lo, end-step, incr, x_degree);
        } else // too small to split
        return null;
    }
}
```

- This could be solved by defining a specialisation of `ZipSpliterator`, defined as an inner class inside the class `PolynomialValue`.
- In this way, all the instances of the inner class will have access to the instance of the outer class - `PolynomialValue.this`. 
Connecting the computation phases

- The supplier provides a new instance of *PolynomialValue*,
  - but it should be one created as a copy of the *initial PolynomialValue instance*, which also has to be
    - the one through which the *initial spliterator was created*.
- The reason for this is the need for
  - connection between the different phases of the computation
    - splitting,
    - leaf(basic case),
    - combining.
Usage example

```java
List<Double> list_int = //...the coefficients list
PolynomialValue pv = new PolynomialValue(x);
PolynomialValue.PZipSpliterator sp_it =
    pv.new PZipSpliterator
    (list_int, 0, list_int.size() - 1, 1);
if (sp_it.hasCharacteristics(
    SpliteratorPower2.POWER2)) {
    System.out.println("characteristic POWER");
    Stream<Double> myStream =
        StreamSupport.stream(sp_it, true);
    PolynomialValue valp =
        myStream.collect(pv);
}
```
Performance Analysis - Polynomial value

Speedup

Time

HIPS 2020

Enhancing Java Streams API with PowerList Computation
Conclusions (1)

- The adaptation of Java Streams uses:
  - the `collect` function as a *template method* for defining the divide-and-conquer PowerList skeletons.
  - specializations of the `Spliterator`

- The PowerList functions are defined as classes implementing the `Collector` interface, and so it wraps the three arguments used in the `collect` function.

- The analysed examples emphasise the fact that for a large majority of PowerList functions, the definition inside Java Stream API could be done very easy based on the proposed adaptation.
The performance obtained for the parallel execution of these functions proved to be very good.

The advantage of Powerlists over general lists is that they provide two different views over the underlying data, simplifying the design of the algorithms on Powerlists.

Extension to PLists (so multiway divide-and-conquer) needs an extension of the definition of the Spliterator such that \texttt{trySplit} method to be able to return a \textit{set} of \textit{Spliterators} that all together cover all the elements of the source.
Thank you!

Contact:
- Virginia Niculescu
- email: vniculescu@cs.ubbcluj.ro
- Faculty of Mathematics and Computer Science, Babeș-Bolyai University, Romania
- 1, M. Kogalniceanu, Cluj-Napoca, Romania