# Enhancing Java Streams API with PowerList Computation

Virginia Niculescu[*][†], Darius Bufnea [*][‡], Adrian Sterca [*][§]
[*]*Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania*
[†]*vniculescu@cs.ubbcluj.ro,* [‡]*bufny@cs.ubbcluj.ro,* [§]*forest@cs.ubbcluj.ro*

***Abstract*—Since they were introduced, Java streams were very fast embraced by the industry, being currently used at a large scale. The parallelism enabled by them is very easy to achieve, but it is constrained either by the used parallelism model (in some cases), or by the set of operations that could be specified using streams. We investigate in this paper the possibility to enhance the computation types that could be defined using the Java streams API by introducing into this infrastructure the PowerList theory based computation. Powerlists are recursive data structures that together with their associated algebraic theory offer both abstractions in order to ease the development of parallel applications, and also a methodology to design parallel algorithms. The Java streaming infrastructure could be adapted to support them in a great measure. We present here such an adaptation, and we analyse and discuss the advantages and constraints. This analysis is exemplified by application examples.**

***Keywords*-parallel programming; streams; recursive structures; Java; performance; models.**

## I. Introduction

An increasing interest in Java for High Performance Computing (HPC) has been registered based on the appealing features of this language for programming multi-core cluster architectures, particularly multithreading support, the built-in networking, and also the continuous increase of the performance of the Java Virtual Machine (JVM). It didn't attain yet a similar level as C, C++, and Fortran on the mainstream HPC community, but still the latest developments are promising.

An important approach in defining high level parallel models is based on algorithmic skeletons [10]; they have been used for the development of various systems providing the application programmer with suitable abstractions.

Java 8 Streams is playing an important role in bringing functional programming to Java, and the streams are also based on algorithmic skeletons. One of the most important features of Java 8 is the integration of lambda expressions in the language. Lambda expressions can be combined with an almost functional interface API (Streams). Although it is not fully functional programming, because the way of introducing arrays and objects in Java Streams is via side effects, it is close to functional programming style.

In the same time, the Java stream library provides the ability to do parallelisation easily, and in a reliable manner. It is actually quite simple to use in the sense that the user only has to invoke a few methods and the rest is managed by the Streams API.

*Powerlist*s and their associated theories offer a skeleton based approach in construction of parallel programming models. *Powerlist* theory [18] allows formal definitions of a broad class of parallel programs based on the divide-and-conquer paradigm.

We developed a Java framework – JPLF – that enables the definition, and different types of parallel execution of *PowerList* functions. During this development process several comparisons have been done with Java Stream computations, and from these emerged the idea to try to include the *PowerList* specific computations directly into the Java Streams API.

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

The paper is organized as follows. In section II we give a general description of *PowerLists* and of their associated theories, and then some general aspects about the already developed framework - JPLF - are given in section III. Section IV presents the proposed adaptation of the Java streams to accept *PowerList* functions. An analysis of the advantages and the constraints of our approach is given in Section V. Related work is presented in section VI, and then section VII presents some conclusions.

## II. PowerList Theory

The theory of *powerlists* data structures has been introduced by J. Misra [18], and especially because the index notations are not used, it offers an elegant way for defining divide-and-conquer programs at a high level of abstraction. The functions on *PowerList*s are defined recursively by splitting their arguments based on two deconstruction operators. A *PowerList* is a linear data structure with elements of the same type, with the specific characteristic that the length of a *PowerList* is always a power of two.

In order to allow reasoning about the correctness of the parallel programs using *PowerList*s, an algebra (that allows also program transformation) and several induction principles are defined on these data structures. Other similar theories such as *ParLists* and *PLists* were defined [16], for working also with lists with non power-of-two lengths, and divide-and-conquer functions that split the problem in a number of subproblems. They extend the set of computation skeletons that could be defined using these data structures.

The main advantage and specificity of the *PowerList* is the fact that there are two constructors (and correspondingly two desconstructors) that could be used: two *similar* powerlists (with the same length and type) can be combined into a new, double length, power list data structure, in two different ways:

- using the operator *tie*, written $p \,|\, q$, the result containing elements from $p$ followed by elements from $q$,
- using the operator *zip*, written $p \natural q$, the result containing elements from $p$ and $q$, alternatively taken.

The proofs of properties on *PowerLists* are based on a structural induction principle defined on *PowerLists*, which consider a base case (for singletons), and two possible variants for the inductive step: one based on the *tie* operator, and the other based on *zip*.

Functions are defined based on the same principle. As a *PowerList* is either a singleton (a list with one element), or a combination of two *PowerLists*, a *PowerList* function can be defined recursively by cases. For example, the high order function $map$, which applies a scalar function to each element of a *PowerList* is defined as follows:

$$\begin{array}{lcl} map(f,\ [a]) & = & [f(a)] \\ map(f,\ p\,|\,q) & = & map(f,\ p)\,|\,map(f,\ q) \end{array} \tag{1}$$

The classical $reduce$ function could be defined in a similar manner.

For both $map$ and $reduce$, alternative definitions based on the *zip* operator could also be given. These could be useful if - depending on the memory allocation, and access – one could be more efficient than the other.

Still, there are functions where the existence of both operators is essential. Function $inv$ permutes the input list $p$ such that the element with the index $b$ in p will be on the position given by the reversal of the corresponding *bit* string representation of $b$:

$$\begin{array}{lcl} inv([a]) & = & [a] \\ inv(p\,|\,q) & = & inv(p) \natural inv(q) \end{array} \tag{2}$$

Many other functions, for example the Fast Fourier Transform, benefit from the existence of the two operators *tie* and *zip*.

The Fast Fourier Transform algorithm defined by Cooley and Tukey [12] has a very simple *PowerList* representation, which has been proved in [18]:

$$\left\{ \begin{array}{lcl} fft([a]) & = & [a] \\ fft(p \natural q) & = & (P + u \times Q)\,|(P - u \times Q) \end{array} \right. \tag{3}$$

where $P = fft(p)$, $Q = fft(q)$ and $u = powers(p)$. The result of the function $powers(p)$ is the *PowerList* $(w^0, w^1, .., w^{n-1})$ where $n$ is the length of $p$ and $w$ is the $(2 \times n)$th principal root of 1.

The operators $+$ and $\times$ used in the *fft* definition are extension of the binary addition and multiplication operators on *PowerLists*. They have simple definitions that consider as an input two similar *PowerLists*, and specify that the elements on the similar positions are combined using the corresponding scalar operator.

The parallelism of the functions is implicit: each application of a deconstruction operator (*zip* or *tie*) implies two independent computations that may be performed in two processes (programs) that could run in parallel. So, we obtain a tree decomposition, which is specific to divide-and-conquer programs, and having two decomposition operators eases the definition of different programs, but at the same time may induce some problems when these high-level programs have to be implemented on concrete parallel machines.

The *PList* data structure was introduced in order to develop programs for the recursive problems which can be divided into any number of subproblems, numbers that could be different from one level to another [16]. It is a generalisation of the *PowerList* data structure and it has three constructors for creating *Plists*: one that creates singletons from simple elements, one based on concatenation of several lists, and the other based on alternative combining of the lists. The corresponding operators are $[\,.\,]$, ($n$-way $|$), and ($n$-way $\natural$); for a positive $n$, the ($n$-way $|$) takes $n$ similar *PList* and returns their concatenation, and the ($n$-way $\natural$) returns their interleaving.

In *PList* algebra, ordered quantifications are needed to express the lists' construction. The expression

$$[\ |\ i : i \in \overline{n} : p.i]$$

is a closed form for the application of the $n$-way operator $|$, on the *PLists* $p.i, i \in \overline{n}$ in order. The range $i \in \overline{n}$ means that the terms of the expression are written from 0 trough $n-1$ in the numeric order.

For example, if we have $p.i = [i * 3, i * 3 + 1, i * 3 + 2]$ then we have:

$$\begin{array}{lcl} [\ |\ i : i \in \overline{3} : p.i] & = & [0, 1, 2, 3, 4, 5, 6, 7, 8] \\ [\ \natural\ i : i \in \overline{3} : p.i] & = & [0, 3, 6, 1, 4, 7, 2, 5, 8] \end{array}$$

The existence of the two decomposition operators differentiates these theories from other list theories, and also represents an important advantage in defining many parallel algorithms.

## III. JPLF Framework

A Java framework – JPLF – for parallel computation defined based on *PowerList* theory has been developed and details about it can be found in [19], [20] and [21]. The framework provides general support implementations for computing *Powerlist* functions; it facilitates multithreading parallel programming, and since Java can be accepted as an alternative for HPC programming, the framework was extended for supporting the execution also on cluster architectures, based on a Java MPI binding (such as [24]) in combination with the multithreading facilities.

We shortly present it here just to emphasize to which extent we may define similar *PowerList* theory based computations using only Java Streams.

The multithreading computation facilitated by JPLF is based on using thread pools and the tested implementation uses the `ForkJoinPool` executor [25], as is the parallelisation of Java Streams. The MPI implementation needs more complex execution, but it could be mixed with multithreading.

The design of this framework was guided by the types defined in the *PowerList* theory, and by their specific properties and operations.

The two characteristic operations: *tie* and *zip* are used to split a *PowerList*, but they could also be used as constructors. The theory considers that the *PowerList* functions are defined by applying one deconstruction operator (for divide operation) for each input *PowerList*, and a combining function. If the result is also a *PowerList*, then the combining function is based on a *PowerList* construction operator, too.

The definition of the divide-and-conquer functions over *PowerList*s is done based on the *template method* design pattern [15]. The `PowerFunction` class defines the template method `compute` that implements the solving strategy. For any new function, the user should provide implementations for the following methods:

- `basic_case`
- `combine`
- `create_left_function`, `create_right_function`

An important advantage of the framework is the fact that the execution is managed separately from the *PowerList* function definition. The executors definition is based on the primitive operations: `basic_case`, `combine`, `create_left_function`, and `create_right_function`, and so, it is possible to define different execution variants for a *PowerList* program: sequential, multithreading, MPI or others.

The class of *PowerList* functions allowed by the JPLF framework is characterized by the fact that these functions could be computed recursively based on their values on the split argument lists (obtained using a decomposition operator). This includes a broad class of functions as it has been proved in [18] and [16]: Fast Fourier Transform, Batcher sort, Bitonic sort, Prefix sum, Gray codes, etc.

We may emphasise the following three phases of a *PowerList* function execution:

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.

For functions such as $map$ or $reduce$ the descending phase has only the role of distributing the input data to the processing elements. The input data are not transformed during this process. Even for Fast Fourier Transform ($fft$) we have the same case – only in the ascending phase special operations need to be done.

There are in fact very few cases when the input is transformed at the descending phase, or additional operations are necessary at this phase. Such cases could involve some additional computations on the sublists obtained at each step. A very simple example is computing the value of a polynomial in a given point:

$$vp([a],x) = [a]$$
$$vp((p\#q),x) = vp(p,x^2) + (x \cdot vp(q,x^2)) \tag{4}$$

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

Also, at each decomposition step the square of the point value – $x$ – should be computed.

These additional operations at the descending phase could be encountered for functions which are based on the *tie* operator, too.

$$\begin{array}{rcl} f([a]) & = & [a] \\ f(p|q) & = & f.(p \oplus q)|f.(p \otimes q) \end{array} \tag{5}$$

where $\oplus$ and $\otimes$ are some extended binary operators.

Also, there are many cases when function transformations could be applied – such as tupling – in order to eliminate these additional computations [22].

In [19] a comparison between the performance of some algorithms' implementations using Java parallel streams and using the JPLF framework with multithreading execution is done, and it emphasizes that for applications based on simple concatenation, the performance results are similar, but this framework has the advantage of the additional support for applications that need more complicated data decomposition as those that involves additional operations at the decomposition phase or those that need the $zip$ operator (as $fft$). The MPI executors facilitates a much larger scalability and so better performance. The JPLF also includes *PList* functions, that express multi-way divide-and-conquer computations [21]. The JPLF framework could be promoted as a Java package, too.

## IV. Java Stream Adaptation to PowerList

The adaptation is inspired by the implementation of the JPLF framework and by the common Java stream applications.

Since Java streams are now a well-known feature of the language, we didn't introduce a special section for explaining them. The official Oracle documentation ([26]) and some other excellent technical articles ([27], [28]) contain many explanations and examples.

Briefly, we can say that a Java stream is a sequence of objects represented as a conduit of data. Usually it has a source where the data are stored 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. A type with a monad structure defines what it means to chain operations, or nest functions of that type together.

There are many already implemented operations that could be sent to a stream, the most common being `map`, `filter`, or `reduce`.

### *The* `collect` *template method*

A more general operation is defined by the function `collect` with the following definition:

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

This performs a mutable reduction operation on the elements of the calling stream.

A mutable reduction is one in which the reduced value is a mutable result container, such as an `ArrayList`, and elements are incorporated by updating the state of the result rather than by replacing the result.

The arguments have the following responsibilities:

- **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**: an associative, non-interfering, stateless function that must fold an element into a result container.
- **combiner**: an associative, non-interfering, stateless function that accepts two partial result containers and merges them, which must be compatible with the accumulator function. The combiner function must fold the elements from the second result container into the first result container.

The following example shows how the words in a given list could be concatenated, including a comma between each pair of two words.

```
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 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.

With this definition, the function `collect` qualifies for the role of a template method to be used for implementing a structural divide-and-conquer skeleton based operations. Our adaptation of Java Streams for accepting *PowerList* type computation will use it with this purpose.

For the splitting phase we will analyze the iterators used by the Stream API.

The supplier will help creating places for the result of the *base case* - the leaves into the divide-and-conquer associated computation tree, the accumulator will be used to set the values on leaves, and the combiner to compute the values of the interior nodes.

### A. *Spliterator specialisation*

The parallel computation of the streams is directed by the existence of a special type of iterator – `Spliterator` – and by the usage of the `ForkJoinPool` executor. It is known that the `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, and we mention here only the `trySplit` operation that partitions off some of its elements as another `Spliterator`, to be used in possibly-parallel operations.

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 should be provided - `TieSpliterator` and `ZipSpliterator`. The UML class diagram for these classes is presented in Figure 1

A source split using a `ZipSpliterator` could not be recreated by using simple concatenation, so we need to provide operations for the *tie* and *zip* constructor operators, too. This could be achieved by defining a class `PowerList` that extends a list (more specifically an `ArrayList` – but any `RandomAccess` collection could also fit); the class provides `tieAll` and `zipAll` methods, which append the elements of a collection argument, accordingly (Figure 2).

The `SpliteratorPower2` defines a specific characteristic `POWER2` that expresses the fact that the number of the elements of the stream is a power of two. This is necessary in order to verify that we work with a stream on which we may apply *PowerList* functions.

### B. *PowerFunction definition*

Based on the previous definitions, we may now define a simple computation of *PowerList* functions using Java Stream. The driving force is represented by the specialized

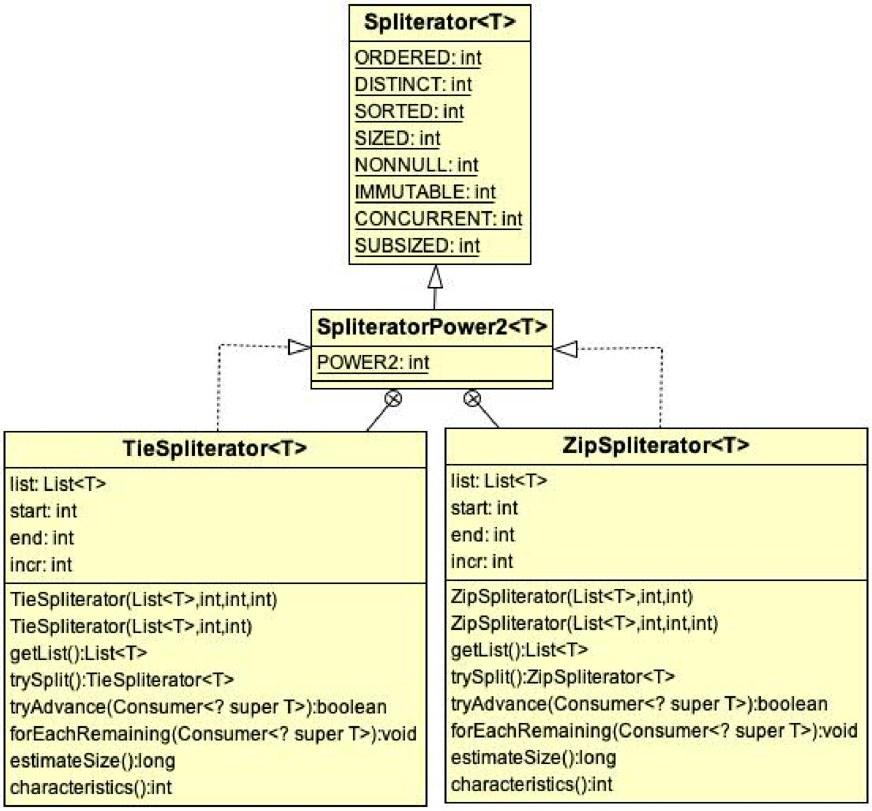

Figure 1: The class diagram of classes TieSpliterator and ZipSpliterator.

spliterators, which are going to be used together with the `collect` function.

The first example is the definition of an identity function, meant to verify the correct decomposition and combining. The following code snippet shows spliterator instantiating, the creation of the stream based on this spliterator, and then the call of the `collect` method with the appropriate arguments.

```
List<Double> list_int = //... some data

//create the ZipSpliterator
ZipSpliterator<Double> sp_it =
 new ZipSpliterator<Double>
    (list_int,0,list_int.size()-1);

//create the stream based on ZipSpliterator
     instance
Stream<Double> myStream =
  StreamSupport.stream(sp_it, true);

//define a specialisation of the collect function
List<Double> li = myStream.collect(
  PowerList<Double>::new,
  PowerList<Double>::add,
  PowerList<Double>::zipAll );
```

It can be noticed that the stream was created using the specialized spliterator – `ZipSpliterator` – this is the way we force the decomposition based on this specific spliterator. The class `StreamSupport` facilitates the creation of a parallel stream starting from an iterator (if the second argument is equal to **`true`**, a parallel stream is created).

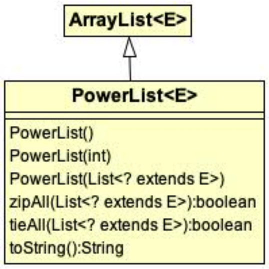
ArrayList<E>
PowerList<E>
PowerList()
PowerList(int)
PowerList(List<? extends E>)
zipAll(List<? extends E>):boolean
tieAll(List<? extends E>):boolean
toString():String

Figure 2: The diagram of the PowerArray class.

When executing the `collect` function, the stream is decomposed using the same `ZipSpliterator` instance, and then recomposed based on the function `zipAll` of the class `PowerList`.

If instead of providing as the accumulator a simple `add` function, we give a function that first applies an operation (`Function<Double,Double> f`) and then adds the value, a map definition is obtained.

```
(list, d) -> { d= f(d); list.add(d);}
```

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 for each specific function – *APowerFunction*, a class `APowerFunction` that implements the `Collector` interface could be defined, and then for its execution we just need to invoke the `collect` function with an argument, which is an instance of that class.

For a function that doesn't impose any additional computation at the splitting phase, the definition is straightforward: just the implementation for the three specific functions (supplier, accumulator, and combiner) should be provided.

We will consider the example of computing the value of a polynomial in a point, that has a very simple parallel *PowerList* definition, but which also involves some operations at the splitting phase.

The *PowerList* definition of it was given in Equation 4.

The definition imposes the propagation at the splitting phase of the square of $x$ value, while still preserving the previous value of $x$ to be used at the combining phase.

The class `PolynomialValue` implements the `Collector< Double, PolynomialValue, PolynomialValue>` interface, the result being stored inside an instance of type `PolynomialValue`.

The value of a polynomial in the point $x$ is computed from the values of other two polynomials (resulted by partitioning the coefficient list) but in a point equal to $x^2$.

The class defines three attributes: the value $x$, the value of the polynomial $val$, and the degree of the value $x$.

```
class PolynomialValue implements Collector< Double
    , PolynomialValue, PolynomialValue>{
  private double x;
  private double val = 0;
  private int x_degree = 1;
 public PolynomialValue(double x){
   this.x = x;
  }
  public PolynomialValue(PolynomialValue pv){
   this.x_degree = pv.x_degree;
   this.x = pv.x ;
   this.val = pv.val;
  }
@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;
  };
}
```

For the polynomials resulted from the first split (first level) the *$x$_degree* is equal to 2, for those from the second level *$x$_degree* is equal to $2^2$, and so on. This means that when a splitting operation is done, the value of *$x$_degree* should be modified.

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`**.

```
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;
  }
```

Splitting operations are executing into tasks which are executed in parallel and if all of them access the same resource, this should be protected through a synchronized block.

Each time the `trySplit` function is called, the exponent of $x$ is doubled (for the next level polynomials), but the global exponent is updated only if its value is less than the local iterator value. The reason for this verification is due to the non-determinism of parallel task execution.

The supplier provides a new instance of `PolynomialValue`, but one that it is 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 a connection between the different phases of the computation – splitting, leaf(basic case), combining.

The following code snippet presents the execution of the function `PolynomialValue`. First an instance of the `PolynomialValue` type is created – refered to `pv`. Through this, an instance of type `PolynomialValue.PZipSpliterator` is created over the list of the given coefficients; for this spliterator we verify that it has the `Power2` characteristics. Then the associated parallel stream is created using the class `StreamSupport`. The execution of the function is done by invoking the `collect` function on the stream with an argument equal to the `PolynomialValue` object – `pv`.

```
List<Double> list_int = //...the coeffients 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);
   }
```

## V. Analysis and Discussion

In the JPLF framework the definition of the powerlist classes and functions were oriented on avoiding the need to copy the elements from one container to another. This was possible for multithreading implementation and it was based on updating only the *data structure information*, which contains: the reference to the storage, and the access pattern to the elements $(start, end, increment)$.

For the Java streams adaptation this was not possible since the `collect` function has been used as a template method and assumes the creation of new containers that are combined in several steps.

The greatest difficulty in adapting Java Streams API to accept *PowerList* computation was the lack of communication between computation phases: splitting and composing. Splitting phase is directed by the `Spliterator` instance, which is not directly connected to the computation defined by the `collect` template method. The solution was to define a specialised spliterator as an inner class inside the `Collector` class that defines the *PowerList* function. The specialised spliterators allow the definition of some specific operations to be done at the splitting phase, but in addition, since the inner class instances are intrinsically connected to the outer class object, they are allowed to modify the state of that outer object and so have access to a global shared state.

This was essential for the definition of the function that computes the value of a polynomial in a point. The solution for the stream implementation of this case was not so straightforward, and relies on the fact that all the decompositions are done until the same layer.

There are similar cases when the solutions could be simpler. For example, if we consider the *PowerList* function defined in the equation 5, in the overridden implementation of the `trySplit` method, the elements should be updated correspondingly, before the new `Spliterator` instance is created. But in this case there is no need for updating a global state, and some optimisations could be tried (e.g. parallelize the application of the $\oplus$ and $\ominus$ operators).

Still, it is possible that for some other, very special functions, this kind of solutions could not be found.

A general mechanism of communication between these two computation phases could be defined as:

- define a specialised spliterator as an inner class of the `Collector` class that defines the *PowerList* function.
- allow the spliterator instance to modify/update the state of the outer class instance (we denote it by `functionObject`).
- create the new container defined by the supplier by copying the `functionObject`.
- create the initial spliterator (which is also used to create the input stream) by using the same `functionObject`.

In general, the `Spliterator` method

```
void forEachRemaining(Consumer<? super T> action)
```

is also important for parallel execution since the splitting is automatically stopped when a limit that depends on the system is attained. This means that the basic case is, in many situations, applied to sublists (*PowerList*s) that are not singletons, but with a length greater than 1. The implicit implementation applies the accumulator for each element. The computation on these sublists is done sequentially and the computation could be specialised by overriding this function. For example, for the Fast Fourier Transform, the computation on these sublists could be defined as a sequential computation of a polynomial in a given point.

This overriding should be provided in a specialisation of spliterator (either `ZipSpliterator` or `TieSpliterator`) which has to be defined inside the class that defines the *PowerList* function.

Definitions of the existing stream function - as `map` or `reduce` based on a `ZipSpliterator` could make sense in some performance tests where different memory access patterns for the elements could give some differences; depending on the system (caches, etc.) properties or data representation, linear or cyclic data distributions could lead to better performance.

Since the definition of the `Spliterator` interface offers only the possibility to split the data in two parts (each time), the possibility to include also the *PList* extension, and so multi-way divide-and-conquer is not possible (yet). If the definition of the `Spliterator` would be extended with a `trySplit` method that returns a set of Spliterators that all together cover all the elements of the source, than the adaptation to *PList* would become possible.

*Performance Analysis*

We conducted some experiments to assess the performance of the proposed implementation. The considered example was the computation of the value of a polynomial in a given point. The polynomial was represented through the list of its coefficients, and the tests were done for different polynomial degrees, from $2^{20}$ to $2^{26}$. The sequential implementation was based on a simple stream based computation, and for the parallel execution we have used the implementation that we have analysed in the previous section.

The experiments were run on an 8 CPU core machine and are depicted in Figure 3 and Figure 4. For each list length value we performed 5 runs of tests and we averaged the obtained results over these 5 runs of experiments.

The first figure shows the speed-up of parallel execution (sequential_execution_time over parallel_execution_time). We can notice here that the speed-up is very good in most of the considered cases, attaining for some of them almost the maximum value 8, which is the number of cores. We also notice there is a dropout in speed-up for the list length of $2^{24}$. This is very probable due to some automatic optimisation of the sequential execution that Java Virtual Machine managed to do for this special case. This explanation seems plausible because the sequential execution time for the value $2^{24}$ is almost 3 times less than the sequential execution time for $2^{23}$, as it can be noticed in Figure 4.

Figure 4 shows the average execution times expressed in milliseconds for the considered polynomial degrees.

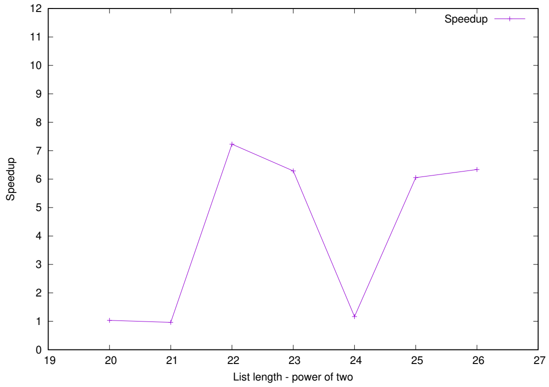

Figure 3: The speedup of the parallel execution, for different polynomial degrees.

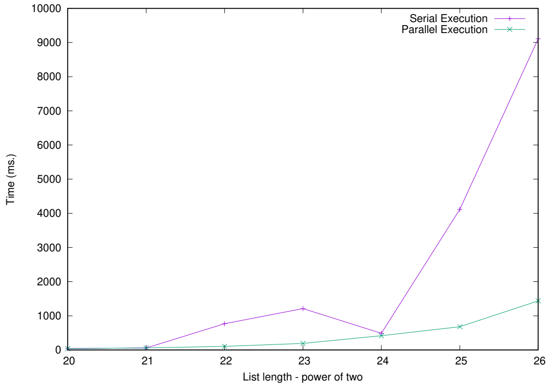

Figure 4: The execution time for the sequential and parallel executions.

## VI. Related work

Java Streams were influenced in their development by many formalisms, especially from functional programming settings. Bird-Meertens formalism (BMF) is such an example that facilitates a calculus for deriving programs from specifications [5]. In the functional programming setting, the skeleton based approach enjoyed a big success, since functional programming concepts allow simple representation of the skeletons. Skeletons have been incorporated into parallel functional languages either as syntactic extensions (e.g. Eden [6]), or as high-order functions within existing languages such as Haskell [7] or ML [8]. Homomorphisms (in particular on join lists [9]) represent a special kind of functions that are very efficient for simple representation of parallel programs that follow the divide-and-conquer structure. They allow representations as compositions between $map$ and $reduce$ functionals.

The *PowerList* theory has been used in other works that try to facilitate the definition of formal and efficient parallel programs. For example, in [1] transformation rules to parallelise divide-and-conquer (DC) algorithms over *PowerList*s are presented. The goal of this work was to derive programs for the *massively data parallel model*. In [3] *PowerLists* are used to capture both parallelism and recursion succinctly, and automatically schedule partitioned matrices over a GPU cluster. They are also used as proving mechanisms – in [4] adder circuits specified using *PowerLists* are proved correct with respect to addition on the natural numbers.

Other works consider the possibility to boost Java performance using different hardware types. In [13] an experimental framework – Jacc – which allows developers to program GPGPUs directly from Java is described. The goal of Jacc, is to allow developers to benefit from using heterogeneous hardware whilst minimising the amount of code refactoring required. Another similar paper [14] presents a framework that enables Java applications to be deployed across a variety of heterogeneous systems while exploiting any available multi- or many-core processor. Java applications are compiled and optimized for the hardware at run-time.

Java has been used lately in High Performance Computing area [23] and also as support language for defining structured parallel programming environments based on skeletons. Such examples are *Lithium* [2] which is implemented as a Java package, *Calcium* [11] and *Skandium* [17].

## VII. Conclusions

We have presented an adaptation of the Java Stream API to allow the definition and execution of the functions defined based on the *PowerList* theory.

*Powerlists* are naturally dealt within a divide-and-conquer manner. Divide-and-conquer is an important programming paradigm and it is also a parallel programming pattern/skeleton. 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*.

The adaptation uses the `collect` function as a template method for defining the Divide-and-Conquer *PowerList* skeletons. The basic case – the trivial sub_problems – are solved using the `supplier` and the `accumulator` arguments of the `collect` method, and for combining two sub_results the `combiner` argument is used. The *PowerList* functions are defined as classes implementing the `Collector` interface, and so it wraps the three arguments used in the `collect` function. This also facilitates the specialisation of the splitting phase. The splitting phase is defined by using `Spliterator` specialisations.

The analysed examples emphasise the fact that for a large majority of *PowerList* functions, the definition inside Java Stream API could be done without difficulty based on the proposed adaptation. The basic cases should be treated carefully since we don't have control over the level at which parallel decomposition stops. The functions for which additional operations are needed at the splitting phase could become more difficult but there are mechanisms that can lead to solutions.

The performance obtained for the parallel execution of these functions proved to be very good, as our experiments for computing the value of a polynomial in a given point, shows.

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.

## References

[1] K. Achatz and W. Schulte, "Architecture independent massive parallelization of divide-and-conquer algorithms," Fakultaet fuer Informatik, Universitaet Ulm, 1995.

[2] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in Java," *Future Generation Computer Systems*, vol. 19, pp. 611–626, 2003.

[3] A. S. Anand and R. K. Shyamasundarn, "Scaling computation on GPUs using powerlists," in *Proceedings of the 22nd International Conference on High Performance Computing Workshops (HiPCW)*. Oakland: IEEE, 2015, pp. 34–43.

[4] D. Kapur and M. Subramaniam. *Mechanical verification of adder circuits using powerlists*. Journal of Formal Methods in System Design. 1998, vol.13 no.2 pp.127-158.

[5] R. Bird, and and O. de Moor. *Algebra of Programming*. Prentice Hall. 1996.

[6] R. Loogen and Y. Ortega-Mallen and R. Pena-Mari. *Parallel Functional Programming in Eden*. Journal of Functional Programming, 2005, no 15, pp. 431-475.

[7] K. Hammond, J. Berthold and Rita Loogen. *Automatic Skeletons in Template Haskell*.Parallel Processing Letters, vol.13, no 3, 2003, pp. 413-424.

[8] R. Di Cosmo, Z. Li and S. Pelagatti and P. Weis. *Skeletal Parallel Programming with OcamlP3l 2.0*. Parallel Processing Letters, 2008, vol. 18, no. 1 pp.149-164.

[9] M. Cole. *Parallel Programming with List Homomorphisms*. Parallel Processing Letters, 1995, vol.5, no.2, pp.191-203.

[10] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[11] D. Caromel and M. Leyton, "A transparent non-invasive file data model for algorithmic skeletons," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–10.

[12] J. W.Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput.*, vol. 19, pp. 297–301, 1965.

[13] J. Clarkson, C. Kotselidis, G.Brown, M. Lujan. "Boosting Java Performance Using GPGPUs". in *International Conference on Architecture of Computing Systems*, LNCS, volume 10172, 2017, pp. 59-70.

[14] J. Clarkson, J. Fumero, M. Papadimitriou, F.S. Zakkak, M. Xekalaki, C. Kotselidis, M. Lujan. "Exploiting high-performance heterogeneous hardware for Java programs using graal," in ManLang '18 *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ACM. Linz, Austria. Sept. 12 - 13, 2018, pp. 4:1–4:13 .

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley, 1995.

[16] J. Kornerup."Data structures for parallel recursion," Ph.D. dissertation, University of Texas, 1997.

[17] M. Leyton and J. M. Piquer. "Skandium: Multi-core Programming with Algorithmic Skeletons," in *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 289–296.

[18] J. Misra. "Powerlist: A structure for parallel recursion," *ACM Trans. Program. Lang. Syst.* vol. 16, no. 6, pp. 1737–1767, 1994.

[19] V. Niculescu, F. Loulergue, D. Bufnea, and A. Sterca. "A

Java Framework for High Level Parallel Programming using Powerlists," in *Parallel and Distributed Computing, Applications and Technologies* (PDCAT). IEEE, Taipei, Taiwan. 2017, pp. 255-262.

[20] V. Niculescu, D. Bufnea, A. Sterca. "MPI Scaling Up for Powerlist Based Parallel Programs," in *Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2019)*, pp. 199-204, DOI: 10.1109/EMPDP.2019.8671597, February 13-15, 2019, Pavia, Italy.

[21] V. Niculescu, D. Bufnea, A. Sterca and R. Silimon."Multiway Divide and Conquer Parallel Programming based on PLists," in *Proceedings of the 27th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1-6, DOI: 10.23919/SOFTCOM.2019.8903794, September 19-21, 2019, Split, Croatia.

[22] V. Niculescu, F. Loulergue. "Transforming powerlist based divide&conquer programs for an improved execution model," in *High Level Parallel Programming and Applications* (HLPP), Orleans, France, 2018.

[23] G.L. Taboada, S. Ramos, R.R. Expsito, J.Tourio, R. Doallo. "Java in the High Performance Computing arena: Research, practice and experience," *Science of Computer Programming* (2013), 78(5), pp. 425-444.

[24] "Intel MPI Library Developer Reference for Linux OS: Java Bindings for MPI-2 Routines," [Online] https://software.intel.com/en-us/mpi-developer-reference-linux-java-bindings-for-mpi-2-routines, accessed: 2020-02-10.

[25] "The Java Tutorials: Fork/Join," [Online] https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, accessed: 2020-02-10.

[26] "Stream (Java Platform SE 8)," [Online] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, accessed: 2020-02-10.

[27] R.-G. Urma. "Processing Data with Java SE 8 Streams," [Online] https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html, Java Magazine, March/April 2014, accessed: 2020-02-10.

[28] B. Winterberg. "Java 8 Stream Tutorial," [Online] https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/, 2017 July, accessed: 2020-02-10.