A Java Framework for High Level Parallel Programming using Powerlists

Virginia Niculescu*†, Frédéric Loulergue¶, 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
¶School of Informatics Computing and Cyber Systems, Northern Arizona University, USA, frederic.loulergue@nau.edu

Abstract—**Parallel programs based on the Divide&Conquer paradigm could be successfully defined in a simple way using powerlists. These parallel recursive data structures and their algebraic theories offer both a methodology to design parallel algorithms and parallel programming abstractions to ease the development of parallel applications.**

The paper presents how programs based on powerlists can be implemented in Java using the JPLF framework we developed. The design of this framework is based on powerlists theory, but in the same time follows the object-oriented design principles that provide flexibility and maintainability. Examples are given and performance experiments are conducted. The results emphasise the utility and the efficiency of the framework.

Keywords-**Parallel recursive structures; Parallel programming; Java; Performance; Models; Framework.**

I. Introduction

Parallel architectures are now mainstream, and span from smartphones to supercomputers, yet parallel programming remains difficult and error-prone. It is thus important not only to provide software abstractions to ease the development of parallel programs, but also conceptual frameworks to ease the design of parallel algorithms.

The theory of lists [4] is such a conceptual framework. Algorithmic skeletons [5] provide the software abstractions needed to develop parallel applications using methodologies based on the theory of lists. There are many skeleton libraries, for e.g. [6], [7], and MapReduce [9] can also be considered as a skeletal parallelism approach.

Other conceptual frameworks include parallel recursive structures such as powerlists [15]. Powerlists are naturally dealt with in a Divide & Conquer (DC) manner. DC is an important programming paradigm and is also a parallel programming pattern. The advantage of powerlists over lists is that they provide two different views over the underlying data, simplifying the design of the algorithms on powerlists.

However, unlike the theory of lists, there is currently no framework for a mainstream programming language that supports parallel programming with powerlists. The latest development of Java, qualifies it to be used for applications where the performance is a critical issue. From these, we think that having a good Java framework that allows the implementation of powerlists parallel programs is worth of consideration.

The goal of the work presented in this paper was twofold: first providing a Java framework to ease the development of parallel programs by mainstream programmers, as well as providing a flexible and maintainable architecture for parallel programmers. In order to do so, our framework follows object-oriented design principles. Powerlists could be easily extended to more general structures that are not constraint to a power of two length (as ParLists and PList [13]), but which still preserve the powerlists advantages. The next step of framework development would be to extend it to use also these kinds of structures.

The paper is organised as follows. Related work is presented in section II. Next, we give a general description of powerlists in section III, and then in section IV the JPLF framework that supports Java implementations of powerlists parallel programs is presented. Sections V-A and V-B present some applications and the practical experiments related to them. We conclude the paper with section VI, also revealing the goals of our further work.

II. Related work

Powerlists are recursive data structures and they can be successfully used for solving problems which are divide and conquer in nature in a very simple manner. These qualify them to be used in skeleton-based parallel programming frameworks. In [1] Achatz and Shulte present transformation rules to parallelise divide-and-conquer (DC) algorithms over powerlists. Their goal was to derive programs for the *massively data parallel model*. Powerlists have been used also in the design of a high level framework for scaling linear algebra computations across a cluster of GPUs [3]. The method is illustrated by using powerlists for a matrix multiplication example. Powerlists facilitate the partitioning of data. An automated scheduling of the partitioned matrices over a GPU cluster was obtained.

In [17] we showed that powerlists could be successfully considered as a base for a high level model for parallel computation that respects the general requirements of abstractness, software development methodology, architecture independence, cost measures, no preferred scale of granularity, efficiently implementable, as these are specified in [18].

Algorithmic skeletons [5] have been used for the development of various systems providing the application programmer with suitable abstractions. They initially came from the functional programming world, but in time, they have been taken by the other programming paradigms too. Efficient skeleton based parallel programming frameworks have been developed targeting multi-core hardware, possibly equipped with GPUs, as well as distributed clusters [10], [11].

Java has been used before as support for defining structured parallel programming environments based on skeletons. Lithium [2] is implemented as a Java package and represents both the first skeleton based programming environment in Java and the first complete skeleton based Java environment exploiting macro-data flow implementation techniques. Calcium [6] and Skandium [14] are two other Java skeleton frameworks. Java 8 Streams are playing an important role in bringing functional programming to Java, and they are also based on algorithmic skeletons.

III. Powerlist Theory

Powerlists have been introduced by J. Misra [15]. They allow working at a high level of abstraction, especially because the index notations of arrays are not used. Instead functions on powerlists are defined recursively by splitting their arguments. A powerlist is a linear data structure whose elements are all of the same type. The length of a powerlist is a power of two.

To help the design of parallel programs using powerlists, an algebra (that allows program transformation) and several induction principles are defined on these data structures. Besides powerlists, similar theories such as ParLists and PLists were defined [13], in order to cover also lists with non power-of-two length, and divide and conquer functions that split the problem in different numbers of subproblems. In this paper we focus on powerlists.

In this section, we write $PL\langle X,n\rangle$ for the type of a powerlist that has 2^n elements each being of type X. A powerlist with a single element a is called a *singleton*, and is denoted by $[a]$. If two powerlists have the same length and elements of the same type, they are called *similar*.

Two *similar* powerlists can be combined into a powerlist data structure with double length, in two different ways:

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

Therefore, the constructor operators for powerlists are:

$$\begin{array}{lll} [.] & : & X \rightarrow PL\langle X,0\rangle \\ .|. & : & PL\langle X,n\rangle \times PL\langle X,n\rangle \rightarrow PL\langle X,n+1\rangle \\ .\natural. & : & PL\langle X,n\rangle \times PL\langle X,n\rangle \rightarrow PL\langle X,n+1\rangle \end{array} \tag{1}$$

Powerlist algebra is defined by these operators and by axioms that assure the existence of a unique decomposition of a powerlist, using one of *tie* or *zip* operators; and the fact that *tie* and *zip* operators commute. The proofs of properties on powerlists are based on a structural induction principle defined on powerlists, which consider a base case, and two possible variants for the inductive step: one based on operator *tie*, and the other based on *zip*.

Functions are defined based on the same principle. As a powerlist is either a singleton, or the combination of two powerlists using either *zip* or *tie*, a function on a powerlist can be defined recursively by case: In the base case, the powerlist is a singleton, otherwise it is the combination of two powerlists on which some possibly recursive calls are done. For example, the high order function map, which applies a scalar function to each element of a powerlist is defined as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{map : (X \rightarrow Z) \times PL\langle X,n\rangle \rightarrow PL\langle Z,n\rangle} \\ map(f,\ [a]) & = & [f(a)] \\ map(f,\ p\,|\,q) & = & map(f,\ p)\,|\,map(f,\ q) \end{array} \tag{2}$$

For reduction with $\oplus$ associative operator, we also have a very simple powerlist definition:

$$\begin{array}{lll} \multicolumn{3}{l}{red : (X \times X \rightarrow X) \times PL\langle X,n\rangle \rightarrow X} \\ red(\oplus,\ [a]) & = & [a] \\ red(\oplus,\ p\,|\,q) & = & red(\oplus,\ p) \oplus (red(\oplus,\ q) \end{array} \tag{3}$$

For both these functions, alternative definitions based on the *zip* operator could be given.

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

$$\begin{array}{lll} \multicolumn{3}{l}{inv :\ PL\langle X,n\rangle \rightarrow PL\langle X,n\rangle} \\ inv([a]) & = & [f(a)] \\ inv(p\,|\,q) & = & inv(p) \natural inv(q) \end{array} \tag{4}$$

The parallelism of the functions is implicit: each application of a deconstruction operator (*zip* or *tie*) means that we may achieve two processes (programs) that could run in parallel. So, we obtain a tree decomposition, which is specific to divide&conquer programs. Having two decomposition operators eases the definition of different programs (as can be noticed from inv definition), but in the same time induces some problems when these high-level programs have to be implemented on concrete parallel machines.

Next, we consider the class of all powerlist functions that could be defined with several powerlists as input arguments, and the result could be either a scalar or, in turn, a powerlist; and such a function respects the following proposition:

Proposition 1. *A powerlist function f could have several* similar *powerlist arguments $p_1, p_2, \ldots, p_k$ and the result could be also a powerlist (in this case $\oplus$ below is a powerlist operator, otherwise it is a scalar operator).*

If the decomposition operators of the arguments are $\odot_1, \ldots, \odot_k$ (where $\odot_i$ could be $|$ or $\natural$) then the function

f could be written as:

$$\begin{aligned} & f([a_1],\ [a_2],\ \ldots,\ [a_k]) = \Upsilon(a_1,\ a_2,\ \ldots,\ a_k) \\ & f(p^l_1 \odot_1 p^r_1,\ p^l_2 \odot_2 p^r_2,\ \ldots,\ p^l_k \odot_k p^r_k) = \\ & \qquad \Phi(p^l_1, p^r_1, p^l_2, p^r_2, \ldots, p^l_k, p^r_k) \\ & \quad \oplus \quad \Psi(p^l_1, p^r_1, p^l_2, p^r_2, \ldots, p^l_k, p^r_k) \end{aligned} \tag{5}$$

where Υ is a function on scalars, Φ and Ψ are functions on powerlists, which contain – in their definitions – calls of f on powerlists of size equal to half of the size of the input lists p_i.

IV. Powerlists Framework in Java

In order to allow easy definition of powerlists functions in Java we have built a framework that provides general support implementations. The design of the JPLF framework is guided by the types defined in the theory and by their specific properties and operations.

`IBasicList` is a type used to allow working with simple basic lists, and which it is also used as a unitary supertype of the specific types defined inside the powerlist theory. Also, this prepares the extension of the framework with types that correspond to PList and ParList data structures. Its implementation `BasicList` is based on an `ArrayList` as a storage container, but other types for the storage could be provided in the future.

The list types defined in our framework respect a property defined by the following design choice:

Design choice 1. *We assume that a list has all its elements stored into the same container object* – the storage – *but two neighbour list elements are not necessarily stored into neighbour locations of the storage: some distance could be between them.*

A. The Powerlist Data-Structure in Java

Since the *powerlist* is the main type that sustains the powerlist theory, we need a good and efficient representation of it in Java. The interface `IPowerList` defines the type and `PowerList` class provide an implementation of it.

The two operations *tie* and *zip* are used to split a powerlist, but they also can be used as constructors. The theory considers that the powerlist functions are defined by considering 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.

To force this definition, the subtypes `ITiePowerList` and `IZipPowerList` are defined. `ITiePowerList` is the type of all powerlists for which we implicitly apply *tie* operator – as construction/deconstruction operator. Similarly, `IZipPowerList` is the type of all the *powerlists* for which we implicitly apply *zip* operator – as construction/deconstruction operator. This helps us defining polymorphic functions for splitting and combining operations.

When a powerlist is divided, the result is formed by two similar sublists. In order to avoid element copy, the storage is preserved for both and only the storage information is updated. For each list`l`, the storage information `SI(l)` is formed of:

- reference to the storage container `base`,
- the start index `start`,
- the end index `end`,
- the incrementation step `incr`.

From a given list with storage information `SI(list)` being `{base, start, end, incr}`, *tie* and *zip* deconstruction operators create two lists `left_list` and `right_list` as follows:

Op.	Side	SI
tie	left	`{base, start, (start+end)/2, incr}`
	right	`{base, (start+end)/2, end, incr}`
zip	left	`{base, start, end-incr, incr*2}`
	right	`{base, start+incr, end, incr*2}`

Note: In our implementation, the storage information contains also a variable `offset` that it is usually equal to 0. A non zero `offset` means that the first element of the list is stored at the position `start+offset*incr`. This is only used for shifting and rotating operations.

The associated class diagram that corresponds to the data structure types is shown in Figure 1.

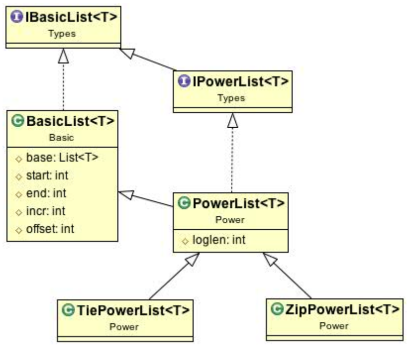

Figure 1: The class diagram of classes corresponding to lists implementation.

B. Framework Design

The framework has several important components with different, but yet interconnected, responsibilities. Their responsibilities are for:

- simple lists (`BasicList`) implementations,
- powerlist structures implementations,
- powerlist functions implementations,
- powerlist functions executions.

This separation of concerns allows us to modify them independently, offering the possibility of extension by providing new or improved ways for execution, or for storage, *etc.*

1) Parallel Programs Definition: A powerlist function expresses the specific computation by either using *tie* or *zip* deconstruction operators for splitting the powerlist arguments. The result could be a simple data (`PowerFunction` case), or a powerlist (`PowerResultFunction` case).

All powerlist functions specify how the powerlist arguments are split, and also, if it is the case, how the result powerlist is constructed for two similar powerlists (`combine` function). This specification is based on a list of construction/deconstruction operators that is an ordered list op_args with values from $\{tie,\ zip\}$.

Proposition 1 implies that a certain powerlist argument is always split by using the same operator. Also, if the result is a powerlist, this is constructed at each step by using the same operator. Based on this proposition, in the framework, the construction and deconstruction operators are not explicitly specified for each function; instead they are implied by the powerlists types – if they are `TiePowerList`s, *tie* operator is used, and if the type is `ZipPowerList`s then *zip* operator is used. So, it is very important when a specific function is called, to prepare it such that the types of the arguments be the types implied by the set op_args. The `PowerList` class provides two methods `toTiePowerList` and `toZipPowerList` that transforms a general `PowerList` into a specific one that has specific implementation for splitting and construction.

In order to allow the implementation of the divide and conquer functions over powerlists we use the *Template Method* design pattern [12]. The `PowerFunction` class defines the template method `compute` that implements the divide&conquer solving strategy. The following code snippet shows the code of the template method `compute` defined for `PowerFunction`:

```
public Object compute() {
  if (test_basic_case())
    result = basic_case();
  else {
    split_arg();
    PowerFunction<T> left = create_left_function();
    PowerFunction<T> right = create_right_function();
    Object res_left = left.compute();
    Object res_right = right.compute();
    result = combine(res_left, res_right);
  }
  return result;
}
```

For a new function, the user should provide implementations for the following methods:

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

Still, it is not mandatory to provide implementations for all of them, their implicit definitions could be used. For example, for *map* we have to give definition only for `basic_case()`, while for *reduce* we have to provide an implementation only for `combine()`. For the functions `create_left_function()` and `create_right_function()` specialised definitions should be given to assure that the new created functions correspond to the particular definition of the function that it is to be defined. Implicitly, the function `test_basic_case()` verifies if the powerlist argument is a singleton, as the powerlist theory specifies. But the method could be overridden and could force ending the recursion before singleton list are encountered. For the functions that do not follow a classical divide&conquer definition the `compute` method could be overridden. Using the presented framework infrastructure, new powerlist functions could be defined by extending the `PowerFunction` or `PowerResultFunction` classes; the first is chosen when the result is a simple, scalar type object (as for `Reduce`), and the second is used for the functions that return powerlists (as for `FFT`)[1]. Figure 2 emphasises these variants. Also, in order to compare the performance of the obtained programs to sequential implementation of the required problem (possible based on other methods) function on `BasicList`s could also be defined by extending either the class `BasicListFunction` or `BasicListResultFunction`.

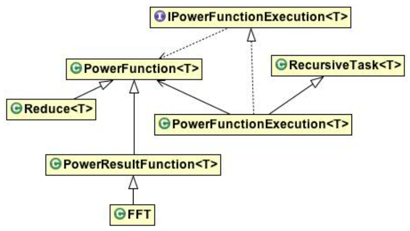

Figure 2: The class diagram of classes corresponding to functions on lists and their execution.

2) Parallel Programs Executions: The execution of a powerlist function is defined separately, in order to allow its modification or specialisation. `IPowerFunctionExecution` is the type that covers the responsibility of executing a powerlist function. It provides methods for setting the function that is going to be execute, and a `compute` function. This execution could be used for any function that follows the divide& conquer pattern.

The class `PowerFunctionExecution` provides an implementation based on the `ForkJoinPool` Java executor. Other implementations could be easily defined, such that other executors to be used. Figure 2 shows the relations between the power function classes and `PowerFunctionExecution`.

The Java `ForkJoinPool` is an implementation of the `ExecutorService` interface, and it is designed for work that can be broken into smaller pieces recursively. As with any Java `ExecutorService` implementation, this distributes tasks to worker threads in a thread pool. The `ForkJoinPool` executor is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy. This executor allows a simple definition of the tasks that we choose to execute in parallel: – each time a split operation is done, a new task is forked for computing the right part, the left part computation being taken by the current task. In this way no more tasks than the number of list elements are created [20].

The implementation of the `compute` method of the class `PowerFunctionExecution` relies on the fact that the pow-

[1]The source code is on: https://github.com/vniculescu/pares_src

erlists functions are defined based on the `Template Method` pattern. Its implementation follows the same skeletons as that used by the `compute` method defined for any powerlist function. Here, both `PowerFunction`s created inside the `compute` method of the `PowerFunction` class (`left` and `right`) are wrapped into separate execution tasks. For the task `right_function_exec` a forked execution is called, and the task `left_function_exec` is computed by the calling task. This implies that no more tasks than the length of the powerlist are created for the function execution.

C. Distributed Lists

Ideally, the implementation of parallel programs described with powerlists consider that any application of the operators *tie* or *zip* as deconstructors, leads to two new processes running in parallel, or, at least, to assume that for each element of the list there is a corresponding process.

`PowerFunctionExecution` forks a new task for the execution of `right-part-function`. This means that the number of tasks grows linearly with the size of the data. In this ideal situation, the time-complexity is usually logarithmic (if the combination step complexity is a constant), depending on $loglen$ of the input list.

In many cases, a more practical approach is preferable: to consider a bounded number p of parallel tasks. In this case we have to transform de input list, such that no more than p tasks are created. This transformation of the input list corresponds to a data distribution. A list of length n is transformed into a list of p sublists, each having n/p elements. In the framework, this responsibility is assigned to the class `Transformer` that defines several functions:

- `toTieDepthList` and `toZipDepthList`,
- `toTieFlatList` and `toZipFlatList`.

For `Transformer` class implementation *Singleton* pattern has been used [12]. The transformations does not imply any elements copying, but just the creation of a new list that uses the same storage, has p elements, each being a `BasicList` object, again with the same storage. The storage information SI is set for each sublist depending on which operator *tie* or *zip* is used for this decomposition. So, the time-complexity associated to this operation is $O(p)$.

The model execution of the functions defined on lists of sublists is different just for the basic case. If the element of a singleton list that corresponds to the basic case has the type `IPowerList` – is a sublist, then a simple sequential execution of the function on that sublist is called. The analysis presented in [16] assures the fact that the final result is correct. For the functions on sublists, sequential execution is based implicitly on recursion, which implies many function calls. In a language such as Java, a recursive implementation is not very efficient, and so, if an equivalent function defined over `IBasicList` (based on iteration) could be defined, then this will be used instead.

V. Applications and Experiments

In order to evaluate the usability of the implemented framework, in this section we will consider two classical problems giving solutions to these problems using our framework. We will also discuss some performance implications of the framework's parameters by conducting some experiments for the two problems, considering different input sizes and different computing solutions.

A. Applications

1) Reduce: The powerlist representation of the reduce computation is given in Section III. The definition of function red could be done either using *tie* or *zip* operator.

Corresponding to this definition, we have defined a class `Reduce<T>` that extends the class `PowerFunction<T>`. The associative operator is given using an instance of the function interface `BinaryOperator` of the package `java.util.function` from Java Platform SE 8, so that a lambda expression could be used to specify it.

The `Reduce` class overrides the method `combine` that applies the associative operator on the results of the recursive calls on left and right lists. The method `basic_case()` is overridden just to include also the case when the argument is a list of sublists, in which case the singleton case uses a sequential reduce function on `BasicList`s. For example, the class `Reduce` could be used to add a list of numbers stored into an `ArrayList` base.

```
int limit = 1<<5;
ArrayList<Double> base = new ArrayList<Double>(limit);
[...] // base initialisation
BinaryOperator<Double> op = (a,b)->(a+b);
PowerList<Double> list =
    new PowerList<Double>(base, 0, limit-1);
ForkJoinPool executor = ForkJoinPool.commonPool();
```

The following code snippets could be used in order execute the reduction:

- sequentially but also recursively using powerlists

```
Reduce<Double> rf =
    new Reduce<Double>(op, list.toTiePowerList());
Double result1 = (Double) rf.compute();
```

- in parallel using powerlists

```
PowerFunctionExecution<Double> exec =
    new PowerFunctionExecution<Double>(rf);
Double result2 = (Double) executor.invoke(exec);
```

- in parallel using a powerlist of sublists

```
Transformer t = Transformer.getInstance();
IPowerList<BasicList<Double>> dlist =
    t.toTieDepthList(list, 1<<3).toPowerList();
Reduce<Double> drf = new Reduce(op, dlist);
PowerFunctionExecution dexec =
    new PowerFunctionExecution<Double>(drf);
Double result3 = (Double) executor.invoke(dexec);
```

2) Fast Fourier Transform: For a polynomial p with complex coefficients, Fourier Transform could be obtained by evaluating p on a specific sequence of points: $(W\ p)$. The points form a powerlist $(W\ p) = (\omega^0, \omega^1, .., \omega^{n-1})$, where

n is the length of p and w is the nth principal root of 1. It can be noticed that $(W\ p)$ depends only on the length of p but not on its elements; hence, for two similar powerlists p, q we have $(W\ p) = (W\ q)$.

A basic sequential implementation of Fourier Transform of polynomial P simply computes the values of the polynomial in all the points of the list $(W\ p)$. The corresponding time complexity is $O(n^2)$. This is what has been implemented using `BasicList` data structure. Since $(W\ p)$ contains powers of the nth principal root of 1, and since they have special relations with the roots of 1 of lower order, the Fourier transform can be recursively computed in $O(n \log n)$ steps, using the Fast Fourier Transform algorithm [8]. The powerlist representation of this algorithm, proved in [15], is:

$$\begin{cases} \mathit{fft}([a]) & = & [a] \\ \mathit{fft}(p \natural q) & = & (P + u \times Q) \,|(P - u \times Q) \end{cases} \tag{6}$$

where $P = \mathit{fft}(p)$, $Q = \mathit{fft}(q)$ and $u = \mathit{powers}(p)$.

The result of the function $\mathit{powers}(p)$ 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 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 theoretical parallel time-complexity of *fft* computation using this powerlist definition is $O(\log n)$ parallel steps using $O(n)$ processors.

B. Experiments

All the experiments have been executed on a Nextscale IBM machine with two Intel Xeon CPU E5-2697 v2 @ 2.70GHz, each processor having 12 physical computing cores (24 with hyper-threading), 128 GB RAM memory and approximately 2.7 TB storage. Thus, a total number of 24 physical computer cores where used in our experiments (48 also considering the virtual ones given by the hyperthreading mechanism). The machine was running Linux Red Hat Enterprise 6.5 on 64bit and JDK 8.

For the two examples with *Reduce* function, the results are compared with the results obtained using Java Parallel Streams, and emphasise that we obtained at least similar performance. Both frameworks are based on *ForkJoinPool* executor that has certain particularities: the performance is very unpredictable since is dependent on a pool of threads, where the tasks are added into each thread queue, and if necessary *work-stealing* principle is used. There is a particularity of the *ForkJoinPool* executor so called "warm-up" behaviour: a second similar execution inside the same program is very much improved since information about tasks creation is preserved; because of this we carefully split all the executions in separate processes which run in separate Java virtual machines.

The *Fast-Fourier Transform* algorithm cannot be directly implemented using *Parallel Streams*. This problem imposes splitting the input list based on the *zip* operator. With this example the advantages of using powerlists are very clear emphasised – they are not only related to performance but also to extending the space of the problems that could be solved using it.

For all experiments, each displayed value is the average value over a series of 100 measures.

1) Reduce: In order to test the performance of the *Reduce* operation using our implementation of powerlists, we conducted a series of experiments, comparing the computational times of different associative operators, operand types and number of operands. We have considered the following cases:

- *Reduce* for a list of complex numbers, and for the associative operator: $(a, b) \rightarrow a + b + a * b$,
- *Reduce* for multiplication of a series of matrices,

In the first *Reduce* experiment, the argument list contains simple `Complex` numbers (the complex numbers implementation is taken from The Apache Commons Mathematics Library [19]), and the associative operator is defined by the formula $(a, b) \rightarrow a + b + a * b$. The reason for choosing this operator is based on the fact that we tried to increase the computational work executed at each step, and also to avoid the dynamic optimizations of JVM. Figure 3 shows the `PowerList.Reduce` performance for powerlists of size $2^n, n = \overline{15, 24}$ split into 2^p sublists, $p = \overline{3, 9}$. It can be seen that the best time performance while increasing the data size (2^n) was obtained for those values of p for which 2^p is closest to the number of physical computer cores involved in the computation (i.e. $p = 4$ or $p = 5$ for our 24 physical cores test system). Figure 4 shows the comparison between `PowerList.Reduce`, ParallelStreams `reduce`, and sequential `BasicList` computations. We can see in this figure that the `PowerList.Reduce` implementation achieves approximately the same performance as the Java ParallelStreams implementation, and even better for $n = 24$.

For the second *Reduce* experiment we generated powerlists of random 10×10 matrices of 64 bits floating point numbers, and we have used the matrix and its corresponding multiplication operation as also implemented by The Apache Commons Mathematics Library (which is a simple implementation of $\mathcal{O}(n^3)$ time-complexity). Figures 5 and 6 show the `PowerList Reduce` performance for a powerlist of size $2^n, n = \overline{8, 20}$ split into 2^p sublists, $p = \overline{3, 9}$, and the comparison with the similar computation with PowerStreams and BasicLists. We obtained in these two figures similar results as those obtained in figures 3 and 4.

On the sublists, the sequential algorithm defined for `BasicList` is used. `BasicList` implementation for reducing was based in this case on a simple iteration of the elements. This sequential variant on `BasicList` is faster, but equivalent to the sequential powerlist variant, which is based on

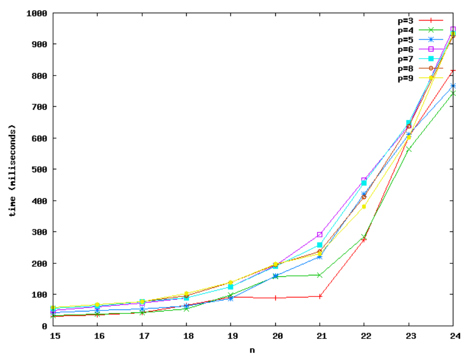

Figure 3: `Reduce` with complex associative operator using `PowerLists` that is split into 2^p sublists, each having 2^{n-p} elements.

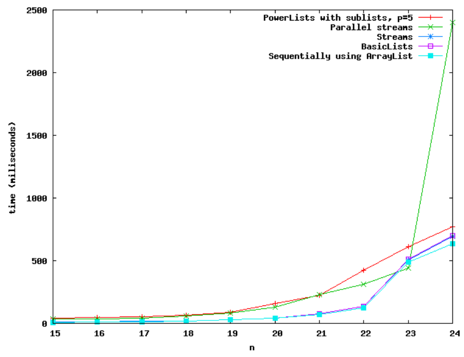

Figure 4: Complex associative operator: `PowerList.Reduce` vs. ParallelStreams `reduce` vs. Sequential – `BasicList`.

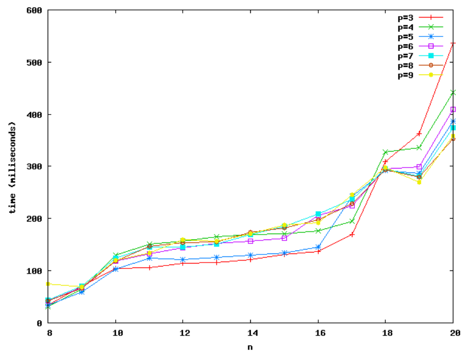

Figure 5: `PowerList.Reduce` with matrix multiplication using `PowerLists` that is split into 2^p sublists, each having 2^{n-p} elements.

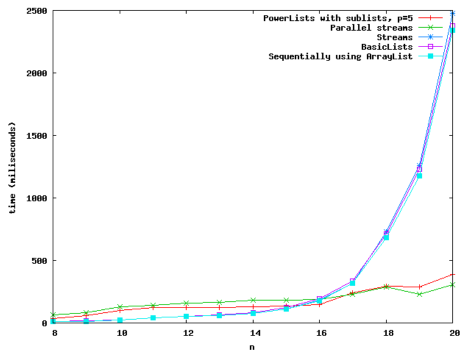

Figure 6: Matrix multiplication: `PowerList.Reduce` vs. ParallelStreams `reduce` vs. Sequential – `BasicList`.

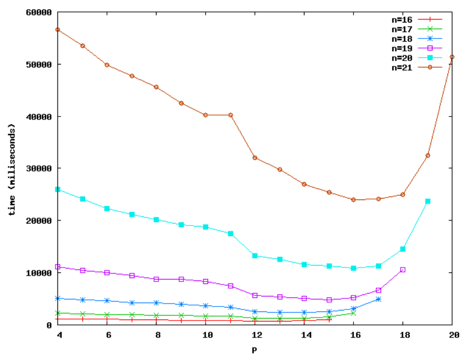

Figure 7: Execution time for FFT using `PowerLists` with 2^p sublists for polynomials of different orders (2^n)

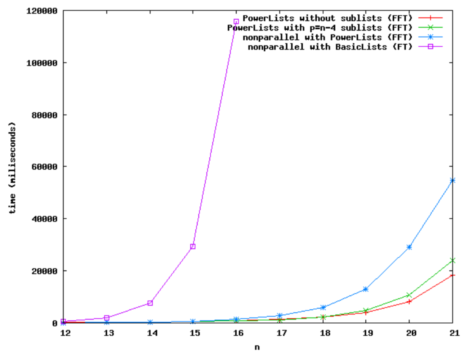

Figure 8: Execution time for Fourier Transform for polynomials of different orders (2^n)

recursion.

The best choices for p depends of the value of n. For the first experiment with lists of complex numbers, for small values of p, some small anomalies can be noticed for $n = 20$ and $n = 21$ – they are due to specific automatic JVM improvements of the code. In average, for both experiments, the case of $p = 5$ could be considered the best; $2^5 = 32$ which is closest to the number of physical machine cores.

From these experiments we may conclude that the performance of `PowerList Reduce` is similar to that obtained with `ParallelStreams`.

2) Fast-Fourier Transform: As we have emphasised in section V-A, Fast-Fourier Transform can be expressed in a very easy and elegant way using powerlists. So, sequential implementation is also based on `PowerLists` but in this case a simple computation without a `ParallelExecutionFunction` call is used.

A corresponding algorithm on `BasicLists` implements a simple, non-fast Fourier Transform, and so less efficient. This is why, in this case, on the sublists we have applied the powerlists function too, but without parallel execution.

Figure 7 shows the `PowerList FFT` performance for a powerlist of size $2^n, n = \overline{16, 21}$ split into 2^p sublists, $p = \overline{4, 20}$. More specifically, it shows how the total size of the data set (i.e. 2^n) and the length of a sublist (i.e. 2^{n-p}; remember that the powerlist with 2^n elements is divided into 2^p sublists) affect the performance of the computation. The best running times were obtained when the length of the sublist, 2^{n-p}, is closest to the number of physical cores in the system (i.e. 24 physical cores). For example, for the $n = 21$ line, the lowest execution time is obtained for $p = 16$, the length of a sublist being $2^{n-p} = 32$. Similarly, for the $n = 20$ line, the lowest execution time is obtained for $p = 16$, the length of a sublist being $2^{n-p} = 16$. For the $n = 19$ line, the lowest execution time is obtained for $p = 15$, the length of a sublist being $2^{n-p} = 16$. 16 and 32 are the powers of 2 that are closest to 24, the number of physical cores in the system. The same happens for the other values of n and p depicted in this figure. In Figure 8 we emphasise a comparison between different variants of FFT programs: a parallel, powerlist based algorithm on the initial list; a parallel, powerlist based algorithm on a partitioned list with 2^p sublists, $p = n-4$ (considering the results from Figure 7 that show that best performance is obtained when $n-p$ is 4 or 5), and two sequential programs: a fast recursive one based on powerlists, and a non-recursive implementation on `BasicList`s.

VI. Conclusion and Future Work

We have presented how programs defined on powerlists could be transformed into real code in the Java programming language. Examples for Reduce and Fast-Fourier Transform have been presented and the experiments done for them show that the framework is practical, performant, and allow simple development of efficient parallel programs.

The experiments shows that for the reduce operation the best time is obtained when we use a list of sublists, and on the sublists a simple sequential algorithm is used. For FFT a very good variant is that when we let the parallel execution to go until creating tasks with simple elements. A possible explanation could be due to the particular implementation of `ForkJoinPool` executor, which has been used.

The comparison, for reduce operation, with Java parallel streams emphasises that the performance of our framework is at least equal to the performance of parallel streams, while it comes with the benefits brought by the possibility of using beside the classical concatenation operator, the *zip* operator. Using in combination these two operators – *tie* and *zip* the user can define very simple definitions of the parallel programs based on recursive functions. The framework design is based on design patterns that provide easy definition of the new concrete programs, but also the possibility to extend the framework to accept other similar data structures and also other execution models. By applying separation of concerns principle, we achieve a framework that separates a data-structure behaviour of its storage, and also separates the execution of a function by its definition. We plan to extend our framework such that to include also *PLists* programs, and so introducing the possibility to define multiways divide & conquer programs. Also, the execution could be extended to distributed memory systems.

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] R. Bird, “An introduction to the theory of lists,” in *Logic of Programming and Calculi of Discrete Design*, M. Broy, Ed. Springer, 1987, pp. 5–42.

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

[6] 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.

[7] P. Ciechanowicz and H. Kuchen, “Enhancing Muesli’s Data Parallel Skeletons for Multi-core Computer Architectures,” in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 108–113.

[8] 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.

[9] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *OSDI*. USENIX Association, 2004, pp. 137–150.

[10] U. Dastgeer and C. W. Kessler, “Smart containers and skeleton programming for gpu-based systems,” *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 506–530, 2016.

[11] S. Ernsting and H. Kuchen, “Algorithmic skeletons for multi-core, multi-GPU systems and clusters,” *Int. J. High Perform. Comput. Netw.*, vol. 7, no. 2, pp. 129–138, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1504/IJHPCN.2012.046370

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

[13] J. Kornerup, “Data structures for parallel recursion,” Ph.D. dissertation, University of Texas, 1997.

[14] M. Leyton and J. M. Piquer, “Skandium: Multi-core Programming with Algorithmic Skeletons,” in *PDP*. IEEE, 2010, pp. 289–296.

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

[16] V. Niculescu, “Data-Distributions in PowerList Theory,” in *Theoretical Aspects of Computing (ICTAC)*, ser. LNCS, C. B. Jones, Z. Liu, and J. Woodcock, Eds., vol. 4711. Springer, 2007, pp. 396–409.

[17] V. Niculescu, “PARES – A Model for Parallel Recursive Programs,” *Romanian Journal of Information Science and Technology (ROMJIST)*, vol. 14, no. 2, pp. 159–182, 2011.

[18] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *Computing Surveys*, vol. 30, no. 2, pp. 123–169, June 1998.

[19] "Commons Math – The Apache Commons Mathematics Library," accessed: 2017-05-10. [Online]. Available: http://commons.apache.org/proper/commons-math/

[20] "The Java^{TM} Tutorials: Fork/Join," accessed: 2017-05-10. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html