Reflection on the Design of Parallel Programming Frameworks

Virginia Niculescu, Frédéric Loulergue, Adrian Sterca

▶ **To cite this version:**

Virginia Niculescu, Frédéric Loulergue, Adrian Sterca. Reflection on the Design of Parallel Programming Frameworks. Evaluation of Novel Approaches to Software Engineering, pp.154-181, 2021, 10.1007/978-3-030-70006-5_7 . hal-03160688

HAL Id: hal-03160688

https://hal.science/hal-03160688v1

Submitted on 5 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reflection on the Design of Parallel Programming Frameworks

Virginia Niculescu[1], Frédéric Loulergue[2], and Adrian Sterca[1]

[1] Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania {vniculescu,forest}@cs.ubbcluj.ro
[2] School of Informatics, Computing and Cyber Systems, Northern Arizona University, USA Frederic.Loulergue@nau.edu

Abstract. Parallel programming is much more complex and difficult than sequential programming, and it is therefore more challenging to achieve the same software quality in a parallel context. High-level parallel programming models, if implemented as software frameworks, could increase productivity and reliability. Important requirements such as extensibility and adaptability for different platforms are required for such a framework, and this paper reflects on these requirements and their relation to the software engineering methodologies that could put them in practice. All these are exemplified on a Java framework – JPLF; this is a high-level parallel programming approach being based on the model brought by the *PowerLists* associated theories, and it respects the analysed requirements. The design of JPLF is analysed by explaining the design choices and highlighting the design patterns and design principles applied.

Keywords: Parallel Programming, Frameworks, Software Engineering, Separation of Concerns, Design Patterns, Recursive Data Structures

1 Introduction

Nowadays, in order to leverage the full computing power of current processors and also because the computing demand is increasing more and more, parallel programming is used in almost all software applications. Parallel programming requires considerably more skills than sequential programming since it introduces an additional layer of complexity and new types of errors. One way to master this complexity is to use frameworks and specialized APIs that make the programmers more productive in writing quality software. Besides performance, these should also provide reliability and flexibility that assures support for various system paradigms.

Since the frameworks for parallel programming are generally built around models of parallel computation, the analysis of requirements for an efficient framework has to start from the general requirements of a good model of parallel

computation. We intend to provide in this paper an analysis of these requirements in relation to the software engineering that allow us to put them in practice. A general architecture is proposed for this kind of frameworks – MEDUGA (Model-Executors-DataManager-UserInteracter-GranularityBalancer-metricsAnalyser). This design is exemplified on a concrete framework –*JPLF: Java Parallel Lists Framework* [29, 27]. This paper is an extension of the conference paper [30], in which the patterns and software development principles used for the JPLF implementation were analysed.

By being based on the *PowerList* theory introduced by J. Misra [25], JPLF is a high-level parallel programming framework that allows building parallel programs that follow the multi-way divide-and-conquer parallel programming skeleton with good execution performances both on shared and distributed memory architectures.

The provided shared memory execution is based on thread pools; the current implementation uses a Java `ForkJoinPool` executor [40], but others could be used too. For distributed memory systems, we considered MPI (Message Passing Interface) [39] in order to distribute processing units on computing nodes. So, JPLF is a multiparadigm framework that supports both multi-threading in a shared memory context and multi-processing in a distributed memory context, and it is open for other types of execution systems, too.

Allowing the support of multiple paradigms requires the framework to be flexible and extensible. In order to achieve these characteristics, the framework was implemented following object-oriented design principles. More specifically, we have employed separations of concerns in order to facilitate changing the low-level storage and the parallel execution environment, and in order to overcome the challenges brought by the multiparadigm support, we have used different design patterns, decoupling patterns having a defining role.

Outline: The remaining of the paper is organized as follows. First we analyse the general requirements for an efficient parallel programming framework in section 2. Section 3 is devoted to a complex analysis of the JPLF framework design and implementation. Related work is discussed in Section 4. We give the conclusions and the specification of further work in section 5.

2 Requirements for a multiparadigm parallel programming framework

In [36] Skillicorn and Talia analyse the usefulness requirements for a model of parallel computation. This kind of models should address both abstraction and effectiveness, which are summarized in a set of specific requirements: abstractness, software development methodology, architecture independence, cost measures, no preferred scale of granularity, efficiently implementable.

In computer programming, a model is seen as an abstract machine providing certain operations to the programming level above and requiring implementations for each of these operations on all of the architectures below. It is designed

to separate software-development concerns from effective execution. We need models because we need both abstraction to assure easy development, but also stability to assure reliability. In general, models are valuable if they are theoretically consistent, fit the real world, and have predictive power.

A software framework is considered an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications. A software framework provides software with generic functionality that can be specialized by additional user-written code, and thus providing application-specific software. It provides a structured way to build and deploy applications from a particular area or domain, and the use of frameworks has been shown to be effective in improving software productivity and quality. In parallel programming they are very important due to the complexity of the parallel execution that make parallel programming writing difficult and error prone. In contrast to libraries, frameworks are characterized by: inversion of control – the flow of control is not dictated by the user, but by the framework; default behaviour; extensibility — through concrete software extension points (usually entitled “hot-spots”); non-modifiable code (usually entitled “frozen-spots”) [34, 20].

In relation to the models of computation, software frameworks come with a more empirical meaning, as they have the role to put the models into practice. They come as a further layer in the development process, providing the structure needed to implement and use a model.

So, in order to build a useful parallel programming framework, we have to rely on a model of parallel computation and provide the context of a concrete implementation for it. This means that we should carefully analyze the requirements of a model for parallel computation and the challenges imposed by the need to put them into practice on actual systems.

2.1 Requirements for a model of parallel computation

We will analyse the requirements stated in [36], and based on them we emphasize what further requirements are implied for the corresponding frameworks.

- *Easy to Understand and Program.* A model should be easy to understand in order to secure a large mass of programmers that could embrace it. If parallel programming models are able to hide the complexities and offer an easy interface, they have a greater chance of being adopted. Parallel execution is a very complex process and a model must hide most of the execution details from programmers in order to allow productivity and reliability. In the same time it is well known that the most challenging issue, for such a computational model is to find a good trade-off between abstraction/readability and performance/efficiency [11].
  The corresponding frameworks should provide well defined constructs that assure easy definition of the computation. Generic code facilitates this very much, but also creational patterns (e.g. *Abstract Factory* or *Builder*) could provide mechanisms that allow the user to easily define correct and efficient programs inside the framework.

- *Architecture-Independence.* Parallel systems are not only very different and non-homogeneous, but are also highly modifiable, being in a continuous process of improvement. We may notice here the big change brought by the GPU devices that modified a lot the world of parallel implementators. So, the models should be independent on this low level of execution, but this rises big challenges for the parallel programming frameworks based on models.

  There are now, many parallel systems that define many computation paradigms, each having particular characteristics. The most used architecture classes are shared-memory, and distributed memory systems, together with their hybrid variants. The hybrid variants may include very important accelerators, as GPUs and FPGAs, that introduce other computing paradigms.

  A very clear separation between the level of specification and definition of the computation and the execution level should be provided by the framework. The model provides the tasks, and the 'executors', that are separately defined, should have the ability of executing the tasks in an efficient way. These executors represent a "hot-spot" in the framework, since they should be specialized for different platforms, and/or improved in time.
- *Software Development Methodology.* This is needed because rising the level of abstraction leads to a gap between the semantic structure of the program, and the detailed structure required for its execution. In order to bridge this gap a solid semantic foundation on which transformation techniques can be built is needed. Correctness by construction is very important in the parallel programming setting, since in this case debugging is a very difficult task. This requirement is intrinsically related to the abstract machine defined by the model.

  The concrete execution should be provided by the framework, and if the model provides a good development methodology, this could be used for the executors definition. They should assure correct execution of the task generated inside the model for different types of parallel system.
- *Guaranteed Performance.* Even if it is not expected to extract absolutely all the performance potential when implementing a model on a particular architecture, the model should assure the possibility to obtain a good implementation on each architecture type. If the corresponding abstract machine associated to the model imposes restrictions about the data access, communications, or other low level computational aspects, then it doesn't qualify as a good general model.

  Also, the model doesn't have to impose certain levels of granularity since the systems could come with different scales of dependencies on granularity.

  A framework built based on a model has to be flexible enough to provide the possibility to improve and adapt the execution to new conditions brought by various systems. This requires flexibility and adaptability. The model being theoretical, could and should emphasize the maximum level of parallelism for a computation. But this maximum level of parallelism could lead to a very fine granularity that may not be appropriate for the concrete system. The adaptation of the task granularity to different system granularity levels

is very important and the framework should tackle this issue, by providing a functionality that could inject the desired level of granularity.
Another important issue in achieving performance is related to data management. Parallel computation is in many cases related to huge volumes of data that have to be read, computed and stored. The computation defined inside the model should be connected to the data, and these operations could have a huge impact on the final cost of computation. Considering the impact of data management on computational costs, this has to be reflected in a well defined framework component, that could be transformed and improved in time.

- *Cost Measures.* The most important goal in a parallel program design and construct is increasing the performance. Execution time is the most important of the concerns, but there are others such as processor utilization or even the development costs that are important too. They all describe the cost of a program, and the model should provide cost measures. At the abstract level, the model provides measures that are independent on the concrete execution level, but these measures should be parameterized well enough, such that they are able to assess the real costs that could be achieved on real machines.
Since the best solution is many times obtained through empirical tests (and not only based on a theoretical analysis inside the model), it would be desirable for the framework to provide functions that automatically gives the evaluation of some metrics – e.g. number of certain functions calls, number of threads/processes created, execution time, etc – and correlate them with some concrete platform system parameters.

By summarizing the previous requirements, we may emphasise the main components of the architecture of a good framework for parallel programming:

- *Model* – the implementation of the model of parallel computation that satisfies the specified requirements;
- *Executors* – a component that treats the execution on different types of parallel systems;
- *Data Manager* – a data management component that deals with data acquisition and management;
- *User Interacter* – a component that facilitates the easy development of new programs inside the framework.
- *Granularity Balancer* – a component able to adjust the granularity of the tasks that should be executed by the executors.
- *Metrics Analyser* – a component that could automatically provide cost measures of the execution in order to evaluate the performance.

The model should be efficiently connected with the other components: it provides the necessary information (e.g. executors receive tasks), but also uses the components functionality when needed (e.g. receive data from Data Manager). The defined components represent the “hot-spots” of the framework. Figure 1 emphasises the proposed architecture of a framework for parallel programming –

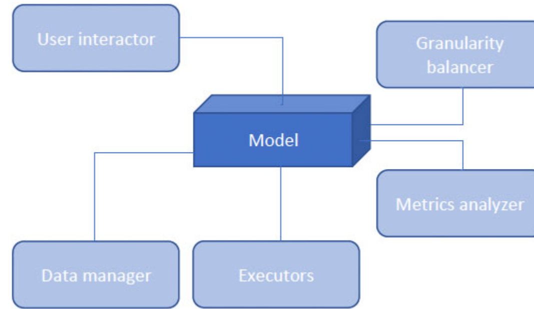

Fig. 1. MEDUGA - architecture scheme of a framework built based on a model of parallel computation.

MEDUGA – Model-Executors-DataManager-UserInteracter-GranularityBalancer-MetricsAnalyser. This architecture could assure a good level of extensibility (through the component), but also stability (through the model).

2.2 The importance on relying on software engineering methodologies and patterns

In general, software engineering methodologies proved to be essential in the development of quality software of any kind. In order to satisfy the analysed requirements, the development of a framework that is based on a model of parallel computation, even more important than in the sequential case, must rely on appropriate software engineering methodologies, due to the complexity of such systems.

Parallel programming emphasizes specific parallel programming patterns and they are mainly related to the parallel programming paradigms that have been inventoried [23]. The model should provide well defined relations with these patterns, and their possible implementations. Many popular parallel computation models are defined based on skeletons [3], which structure and simplify the computational process. They are in direct connection to the parallel design patterns, which offer essential advantages in the development of software processes. They could provide the necessary flexibility and adaptability much needed in this case.

Besides the parallel programming patterns, general design patterns used in the common software design and software development methodologies are also important because the framework should assure correct separation of concerns, flexibility, adaptability and extensibility.

3 The *JPLF* Framework

The JPLF framework has been built following the requirements analysed in the previous section. *PowerList* and associated theories [25, 17] have been selected as

a model of computation, and this model was proved to fulfill the general requirements for a model of parallel computation in [26]. *PowerLists* allow an efficient and correct derivation of programs of divide-and-conquer type; *PList* extensions allow multi-way divide-and-conquer computation, but also "embarrassingly parallel computation"[12]. Extensions with *PowerArray*s, resp. *PArray*s are possible, in order to move to multidimensional data organisation.

3.1 *PowerList* theory as a model of parallel computation

The theory of *Powerlists* data structures introduced by J. Misra [25] offers an elegant way for defining divide-and-conquer programs at a high level of abstraction. This is especially due to the fact that the index notations are not used, and because it allows reasoning about the algorithm correctness based on a formal defined algebra.

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. The functions on *PowerList*s are defined recursively by splitting their arguments based on two deconstruction operators (tie and zip).

Similar theories such as *ParLists* and *PLists* were defined [17], for working also with lists with non power-of-two lengths, and divide-and-conquer functions that split the problem in any number of subproblems. They extend the set of computation skeletons that could be defined using these data structures.

Besides the fact the inside these theories we have a solid software methodology that allows proving program correctness, 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), p and q, can be combined into a new, double length, power list data structure, in two different ways:

- using the operator *tie*, written $p \mid q$, 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.

To prove the correctness of properties on *PowerList*s a structural induction principle is used: this considers 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 *PowerList*s, 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:

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

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.

Moreover, the existence of the two decosntruction operators could be essential for the definition of certain functions. An important example is represented by the algorithm that computes the Fast Fourier Transform defined by Cooley and Tukey [4]; this has a very simple *PowerList* representation, which has been proved correct in [25]:

$$\begin{aligned} fft([a]) &= [a] \\ fft(p \natural q) &= (P + u \times Q) \,|(P - u \times Q) \end{aligned} \quad (2)$$

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 *PowerList*s. They have simple definitions that consider as an input two similar *PowerList*s, and specify that the elements on the similar positions are combined using the corresponding scalar operator.

The parallelism of the functions is implicitly defined: each application of a deconstruction operator (*zip* or *tie*) implies two independent computations that may be performed independently in two processes (programs) that could run in parallel. So, we obtain a tree decomposition, which is specific to divide-and-conquer programs. The existence of 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 [17]. It is a generalisation of the *PowerList* data structure and it has also three constructors: 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 *PList*s $p.i, i \in \overline{n}$ 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.

The *PList* axioms, also define the existence of a unique decomposition of a *PList* using constructors operators. Functions over *PList* are defined using two arguments. The first argument is a list of arities: $PosList$, and the second is the *PList* argument (if there are more than one *PList* argument, they all must have the same length).

Usually the arity list is formed of the prime factors obtained through the decomposition of the list length into prime factors. Still, we may combine these factors, if convenient. The functions could be defined only if the product of the numbers in the arity list is equal to the *PList* argument length. If the arity list is reduced to one element – the *PList* argument length – the decomposition is done only once, and we arrive to an 'embarrassingly parallel computation' type.

We illustrate *PList* functions' definitions with a simple example: the *reduction* function that computes the reduction of all elements of a *PList* using an associative binary operator $\oplus$:

$$\begin{aligned} &defined.red(\oplus).l.p && \equiv prod.l = length.p \\ &red(\oplus).[].[a] && = a \\ &red(\oplus).(x \triangleright l).[|i : i \in x : p.i] && = (\oplus i : 0 \leq i < x : red(\oplus).l.(p.i)) \end{aligned} \tag{3}$$

where *prod.l* computes the product of the elements of list l, *length.p* is length of p, [] denotes the empty list, and $\triangleright$ denotes `cons` operator on simple lists. This function could also be defined using the $\natural$ operator because the $\oplus$ operator is associative.

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.

3.2 *JPLF* design and implementation

We will present in this section some details about the design and implementation of the major components of the framework.

Model Implementation The main elements of the model are interconnected, although they have different responsibilities, such as:

- data structures implementations,
- functions implementations.

Design choice 1. *Impose separate definitions for these elements allowing them to vary independently.*

The motivation of this design choice is that separation of concerns enables independent modifications and extensions of the components by providing alternative options for storage or for execution.

PowerList Data Structures The type used when dealing with simple basic lists is `IBasicList`. In relation to the *PowerList* theory, this type is also used as a unitary super-type of specific types defined inside the theory. The framework extension with types that match the *PList* and *ParList* data structures is also enabled by this.

Design choice 2. *Use the pattern* Bridge *[10] to decouple the definition of the special lists' types from their storage. Storage could be:*

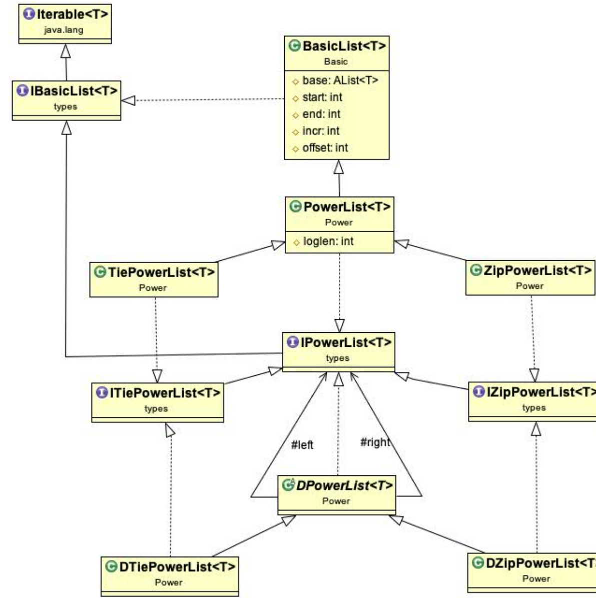

Fig. 2. The class diagram of the classes corresponding to lists implementation [30].

- *a classical predefined list container where all the elements of a list are stored, but this doesn't necessarily mean that two neighbor elements of the same list are actually stored into neighbor locations in this storage: some byte distance could exist between the locations of the two elements;*
- *a list of sub-storages (containers) that are combined using* tie *or* zip *depending on the list type (*Composite *storage).*

The storage part belongs to the ***DataManager*** *component of the framework, since it may vary from one platform to another, and could be extended, too.*

The reason for this design decision is to allow the same storage being used in different ways, but most importantly to avoid the data being copied when a split operation is applied. This is a very important design decision that influences dramatically the obtained performance for the *PowerList* functions execution.

The result of splitting a *PowerList* is formed by two similar sub-lists but the initial list storage could remain the same for both sub-lists having only the *storage information* updated (in order to avoid element copy). Having a list (l), the *storage information* $SI(l)$ is composed of: the reference to the storage container `base`, the start index `start`, the end index `end`, the increment `incr`.

From a given list with storage information $SI(list) =$ `(base, start, end, incr)`, two sub-lists (`left_list` and `right_list`) will be created when either *tie* and *zip* deconstruction operators are applied. The two sub-lists have the same storage container `base` and correspondent updated values for `(start, end, incr)`.

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`

If we have a *PList* instead of a *PowerList* the splitting operation could be defined similarly by updating `SI` for each new created sub-list. If we split the list into p sub-lists then the kth $(0 \leq k < p)$ sub-list has $size = (end - start)/p$ and SI is:

Op.	Sub-List	SI
tie	kth	`base, start+size*(k/p),` `start+size*((k+1)/p), incr`
zip	kth	`base, start+k*incr,` `end-(p-k-1)*incr, incr*k`

The operators *tie* and *zip* are the two characteristic operations used to split a list, but they could also be used as constructors. This is reflected into the constructors definition.

There are two main specializations of the `PowerList` type: `TiePowerList` and `ZipPowerList`. Polymorphic definitions of the splitting and combining operations are defined for each of these types, which determine which operator is used. Since a PowerList could also be seen as a composition of two other PowerLists, two specializations with similar names: `DTiePowerList` and `DZipPowerList` are defined in order to allow the definition of a *PowerList* from two sub-lists that don't share the same storage. This is particularly important for executions on distributed memory platforms. The *Composite* design pattern is used for this.

The corresponding list data structure types are depicted in the class diagram shown in Figure 2.

PowerList Functions A *PowerList* function is expressed in our model by specifying the *tie* or *zip* deconstruction operators for splitting the *PowerList* arguments and by a composing operator in case the result is also a *PowerList*.

Design choice 3. *Use a type driven implementation for* PowerList *functions: if an argument's type is* `TiePowerList`*, then the* tie *operator is used for splitting that argument, and if an argument's type is* `ZipPowerList`*, the* zip *operator is used for it.*

This is possible because for the considered *PowerList* functions, one *PowerList* argument is always split by using the same operator (and so it preserves its type – a `TiePowerList` or a `ZipPowerList`). In case the result is a *PowerList*, the same operator (depending on the concrete type) is also used at each step of the construction of the result.

PowerList functions may have more than one *PowerList* argument, each having a particular type: `TiePowerList` or `ZipPowerList`. The *PowerList* functions don't need to explicitly specify the deconstruction operators. They are determined by the arguments' types: the *tie* operator is automatically used for

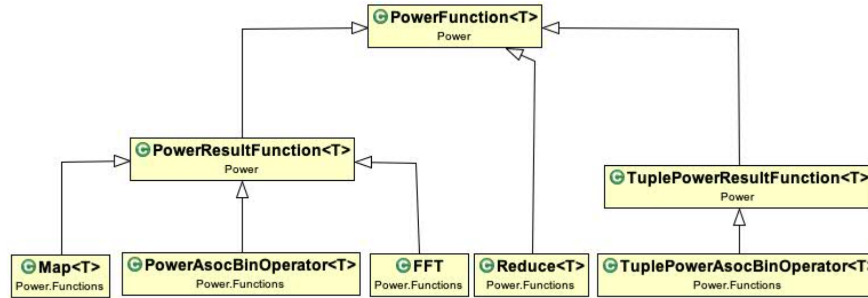

Fig. 3. The class diagram of classes corresponding to functions on PowerLists and their execution [30].

`TiePowerLists` and the *zip* operator is used in case the type is `ZipPowerLists`. It is very important when invoking a specific function, to call it in such a way that the types of its actual parameters are the appropriate types expected by the specific splitting operators. The two methods `toTiePowerList` and `toZipPowerList`, provided by the `PowerList` class, transform a general `PowerList` into a specific one.

The result of a *PowerList* function could be either a singleton or a *PowerList*. For the functions that return a *PowerList*, a specialization is defined – `PowerResultFunction` – for which the result list type is specified. This is important in order to specify the operator used for composing the result.

Design choice 4. *In order to support the implementation of the divide-and-conquer functions over* PowerLists*, use the* Template Method *pattern [10].*

The divide-and-conquer solving strategy is implemented in the template method `compute` of the `PowerFunction` class. `PowerFunction`'s `compute` method code snippet is presented in Fig. 4.

The primitive methods:

- `combine`
- `basic_case`
- `create_right_function` and `create_left_function`

are the only ones that need to be implemented in order to define a new *PowerList* function.

For the `create_right_function` and `create_left_function` functions we should provide specialized implementations to guarantee that the newly created subfunctions (left and right) correspond to the function being computed. For the other two, there are implicit definitions in order to free the user from providing implementations for all of them. For example, for *map* we have to provide a definition only for `basic_case`, whereas only a `combine` implementation is required for *reduce*.

The function `test_basic_case` implicitly verifies if the *PowerList* argument is a singleton, but there is the possibility to override this method and force an end of the recursion before singleton lists are encountered.

```
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;
}
```

Fig. 4. The template method `compute` for *PowerList* function computation.

The `compute` method should be overridden only for functions that do not follow the classical definition of the divide-and-conquer pattern on *PowerLists*.

Figure 3 emphasizes the classes used for *PowerList* functions and some concrete implemented functions: `Map`, `Reduce`, `FFT`. The class `PowerAssocBinOperator` corresponds to associative binary operators (e.g. $+, *$ etc.) extended to *PowerLists*. `TuplePowerResultFunction` has been defined in order to allow the definition of tuple functions, which combine a group of functions that have the same input lists and a similar structure of computation. Combining the computations of such kind of functions could lead to important improvements of the performance. For example, if we need to compute extended *PowerList* operators $< +, *, -, / >$ on the same pair of input arguments, they could be combined and computed in a single computation stream. This has been used for the FFT computation case [28].

3.3 Executors

Multithreading Executors The simple sequential execution of a *PowerList* function is done simply by invoking the corresponding `compute` method.

In order to allow further modifications or specializations, the definition of the parallel execution of a *PowerList* function is done separately. The executors' supertype is the `IPowerFunctionExecutor` interface that covers the responsibility of executing a *PowerList* function. This type provides a `compute` method and also the methods for setting and getting the function that is going to be executed. Any function that complies with the defined divide-and-conquer pattern could be used for such an execution.

Design choice 5. *Define separate executor classes that rely on the same operations as the primitive methods used for the* PowerList *function definition.*

The class `FJ_PowerFunctionExecutor` relies on the `ForkJoinPool` Java executor, which is an implementation of the `ExecutorService` interface. Figure 5 shows the implemented classes corresponding to the multithreading executions based on `ForkJoinPool`. A `FJ_PowerFunctionExecutor` uses a `ForkJoinPool` to execute a `FJ_PowerFunctionComputationTask` that is created to compute a *PowerList* function. The simple definition of the recursive tasks that we choose to execute in parallel is enabled by this executor: new parallel tasks are created each

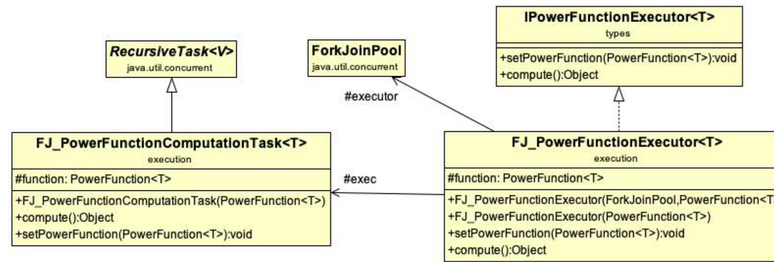

Fig. 5. The class diagram of classes corresponding to the multithreading executions based on ForkJoinPool.

time a split operation is done. As the *PowerLists* functions are built based on the `Template Method` pattern, the implementation of the `compute` method of the `FJ_PowerFunctionComputationTask` is done similarly. The same skeleton, is used in this implementation, too. The code of the `compute` template method inside the `FJ_PowerFunctionComputationTask` is shown in the code snippet of Figure 6.

```
public Object compute() {
 Object result =null;
 if (function.test_basic_case()){ result = function.basic_case(); }
 else{
    function.split_arg();
    PowerFunction<T> left_function =   function.create_left_function();
    PowerFunction<T> right_function =   function.create_right_function();
  //wrap the functions into recursive tasks
    if (recursion_depth == 0){ result = function.compute(); }
    else{
        FJ_PowerFunctionComputationTask<T> left_function_exec =
            new FJ_PowerFunctionComputationTask<T>(left_function, recursion_depth-1);
        FJ_PowerFunctionComputationTask<T> right_function_exec =
            new FJ_PowerFunctionComputationTask<T>(right_function, recursion_depth-1);
        right_function_exec.fork();
        Object result_left = left_function_exec.compute();
        Object result_right = right_function_exec.join();
        result = function.combine(result_left, result_right);
    }
  }
  return result;
}
```

Fig. 6. The `compute` method in `PowerFunctionComputationTask`

In this example, separate execution tasks wrap the two `PowerFunctions` that have been created inside the `compute` method of the `PowerFunction` class (`right` and `left`). A forked execution is called for the task `right_function_exec`, while the calling task is the one computing the `left_function_exec` task.

In order to define other kinds of executors, we need to define a new class that implements `IPowerFunctionExecutor`, and define its `compute` method based on the methods defined by the `PowerFunction` class.

MPI Execution The ability to use multiple cluster nodes for execution could be attained by introducing MPI based execution of the functions [27]. This assures the needed scalability for a framework that works with regular data sets of very large sizes.

The command for launching a MPI execution has, generally the following is:

```
mpirun -n 20 TestPowerListReduce_MPI
```

where the `-n` argument defines the number of MPI processes (20 is just an example) that are going to be created. It could be easily observed that the MPI execution is radically different from the multithreading execution: each process executes the same Java code and the differentiation is done through the `process_rank` and the `number_of_processes` variables that are used in the implementing code.

The advantage brought by list splitting and combining without element moving (just changing the storage information `SI`), which is possible for the execution on shared memory systems, is no longer possible for the distributed memory execution paradigm. On a distributed memory system, based on an MPI execution, the list splitting and combining costs could not be kept so small because data communication between processes (sometimes on different machines) is needed. Since the cost for data communication is much higher than the simple computation costs, we had to analyze very carefully when this communication could be avoided.

During the *PowerList* functions computation when we apply the definition of the function on non-singleton input lists, each input list is split into two new lists. In order to distribute the work, we need to transfer one part of the split data to another process. Similarly, the combining stage could also need communication, since for combining stages, we need to apply operations on the corresponding results of the two recursive calls.

For identifying the cases when the data communication could be avoided, the phases of PowerList function computation were analyzed in details:

1. *Descending/splitting phase* that includes the operations for splitting the list arguments and the additional operations, if they exist.
2. *Leaf phase* that is formed only by the operations executed on singletons.
3. *Ascending/combining phase* that includes the operations for combining the list arguments and the additional operations, if they exist.

The complexity of each of these stages is different for particular functions.

For example, for *map*, *reduce* or even for *fft*, the descending phase does not include any additional operations. It has only the role of distributing the input data to the processing elements. The input data is not transformed during this process.

There are very few functions where the input is transformed during the descending phase. For some of these cases it is possible to apply some function transformation — as tupling — in order to reduce the additional computations. This had been investigated in [28].

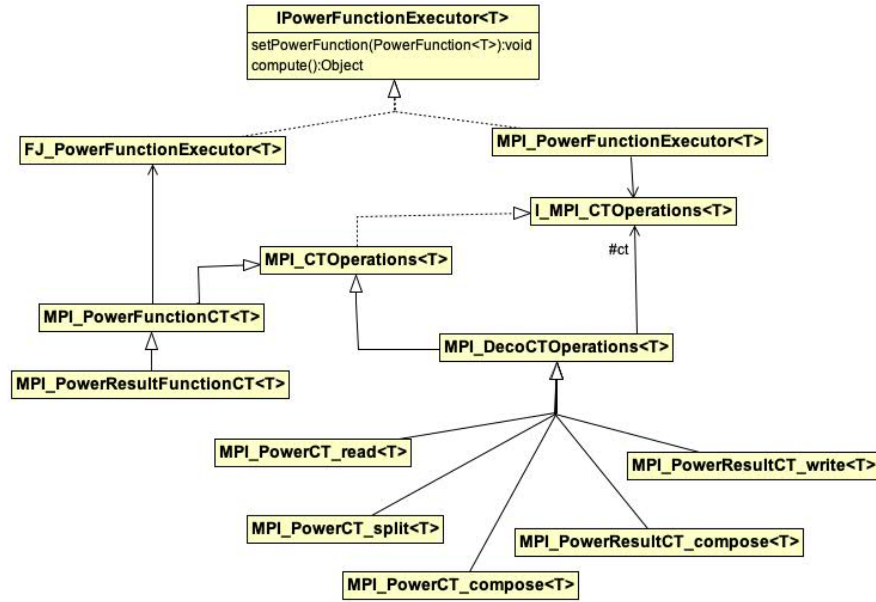

Fig. 7. The classes used for different types of execution of a PowerList function [30].

Similarly, we may analyze the functions for which the combining phase implies only data composition (as *map*) or also some additional operations (as *reduce*).

Through the combination of these situations we obtained the following classes of functions:

1. $splitting \equiv data_distribution$
   The class of functions for which the splitting phase needs only data distribution.
   Examples: *map*, *reduce*, *fft*
2. $splitting \not\equiv data_distribution$
   The class of functions for which the splitting phase needs also additional computation besides the data distribution.
   Example: $f(p \natural q) = f(p+q) \natural f(p-q)$
3. $combining \equiv data_composition$
   The class of functions for which the combining phase needs only the data composition based on the construction operator (*tie* or *zip*) being applied to the results obtained in the leaves.
   Example: *map*
4. $combining \not\equiv data_composition$
   The class of functions for which the combining phase needs specific computation used in order to obtain the final result.
   Examples: *reduce*, *fft*.

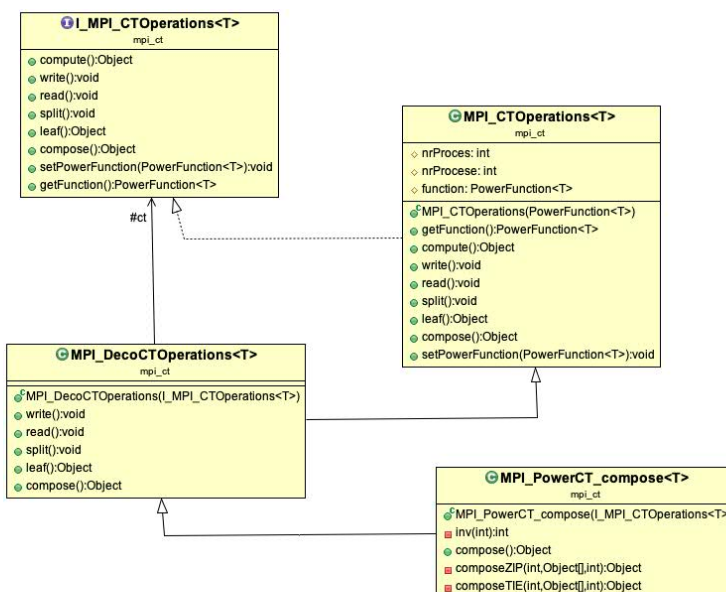

Fig. 8. Implementation details of some of the classes involved in the definition of MPI execution [30].

One direct solution to treat these function classes as efficiently as possible would be to define distinct types for each of them. But the challenge was that these classes are not disjunctive. The solution was to split the function execution into sections, instead of defining different types of functions.

Design choice 6. *Decompose the execution of the PowerList function into phases: reading, splitting, leaf, combining, and writing.*
Apply the Template method *pattern in order to allow the specified phases to vary independently.*
Apply the Decorator *pattern [10] in order to add specific corresponding cases.*

The Figure 7 emphasizes the operations' types corresponding to the different phases. For MPI execution, we associated a different computational task (CT) for each phase. The computational tasks are defined as decorators, they are specific to each phase, and they are different for functions that return PowerLists by those that return simple types (`PowerResultFunction` vs. `PowerFunction`):

- `MPI_PowerCT_split`,
- `MPI_PowerCT_compose`, resp. `MPI_PowerResultCT_compose`,
- `MPI_PowerCT_read`,
- `MPI_PowerResultCT_write`.

Some details about the implementations of these classes are presented in Figure 8. The class `MPI_CTOperations` provides a `compute` template method and empty implementations for the different step operations: `read`, `split`, `compose`, . . . (details are given in Fig. 9).

```
// compute method of MPI_CTOperations class
public Object compute(){
  Object result;
  read();
  split();
  result = leaf();
  result = compose();
  write();
  return result;
}
```

Fig. 9. The template method `compute` for MPI execution.

The `leaf` operation encapsulates the effective computation that is performed in each process. It can be based on multithreading and this is why it could use `FJ_PowerFunctionExecutor` (the association between `MPI_PowerFunctionCT` and `FJ_PowerFunctionExecutor`). Hence, an MPI execution is a implicitly a combination of MPI and multithreading execution.
The `compose` operations in `MPI_PowerCT_compose` and `MPI_PowerResultCT_compose` are defined based on the `combine` operation of the wrapped PowerList function.

The input/output data for domain decomposition of parallel applications are in general very large, and so these are usually stored into files. This introduces other new phases in function computation if reading and writing are added as additional phases (case 1), or if they are combined with splitting (resp. combining phases; in this case they introduce new variations of the function computation phases (case 2).

If the data are taken from a file, then:

case 1: a reading is done by the process 0, followed by an implementation of the decomposition phase based on MPI communications;

case 2: concurrent file reads of the appropriate data are done by each process.

The possibility of having concurrent read of the input data is given by the fact each process needs to read data from different positions on the input file, and also because the data depends on known parameters: the type of the input data (`TiePowerList` or `ZipPowerList`), the total number of elements, the number of processes, the rank of each process, and the data element size (expressed in bytes).

The difficulty raised from the fact that all the framework's classes are generic and also almost all MPI Java implementations need simple data types to be used in communication operations. The chosen solution was to use byte array transformations of the data through *serialization*.

Design choice 7. *Use the* Broker *design pattern in order to define specialized classes for reading and writing data (`FileReaderWriter`) and for serializing/deserializing the data (`ByteSerialization`).*
These specialised classes belong to ***DataManger*** *component of the framework.*

When the decomposition is based on the *tie* operator, reading a file is very simple and direct: each process receives a `filePointer` that depends on its rank from where it starts reading the same number of data elements.

When the decomposition is based on the *zip* operator, file reading requires a little bit more complex operations: each process receives a starting `filePointer` and a number of data elements that should be read, but for each next reading, another seek operation should be done. The starting `filePointer` is based on bit reverse (to the right) operation applied on the process number.

For example, for a list equal to $[1, 2, 3, 4, 5, 6, 7, 8]$ a *zip* decomposition on 4 processes leads to the following distribution: $[\ [1, 5],\ [3, 7],\ [2, 6],\ [4, 8]\ ]$.

In order to fuse the combining phase together with writing, we applied a similar strategy. The conditions that allow concurrent writing are: the output file to be already created and each process writes values on different positions, these positions are computed based on the process rank, the operator type, the total number of elements, the number of processes, and the data element size.

Using this MPI extension of the framework, we don't need to define specific MPI function for each *PowerList* function. We just define an executor by adding the needed decorators for each specific function: a `read` operation, or a `split` operation, and a `compose` operation or a `write` operation, etc. The order in which they are added is not important. In the same time the operations: `read`, `write`, `compose`, etc. are based on the primitives operations defined for each *PowerList* function (which are used in the `compute` template method). Also, they are dependent on the total number of processes and the rank of each process.

To give better insights of the MPI execution we will present the case of the `Reduce` function (section 3.1). The following test case considers a reduction on a list of matrices using addition. The code snippet in Figure 10 emphasizes what is needed for the MPI execution of the `Reduce` function.

```
ArrayList<Matrix> base = new ArrayList<Matrix>(n);
AsocBinOperator<Matrix> op = new SumOperator<Matrix>();
TiePowerList<Matrix> pow_list = new TiePowerList<Matrix>(base,0, n-1, 1);
PowerFunction<Matrix> mf = new Power.Functions.Reduce(op, pow_list);
int [] sizes = new int[1]; sizes[0] = n;
int [] elem_sizes = new int[1];
elem_sizes[0] = ByteSerialization.byte_serialization_len(new Matrix(0));
String [] files = new String[1];
files[0] = "date_matrix.in";
MPI_CTOperations<Matrix> exec =
  new MPI_PowerCT_compose<Matrix>(
    new MPI_PowerCT_read<Matrix>(
      new MPI_PowerFunctionCT<Matrix>(mf, ForkJoinPool.commonPool()),
      files, sizes, elem_sizes) );
Object result = exec.compute();
```

Fig. 10. The MPI execution of the `Reduce` function

3.4 Granularity Balancer

In an ideal case, the execution of parallel programs defined based on *PowerLists* implies the decomposition of the input data using the *tie* or *zip* operator and

each application of *tie* and *zip* creates two new processes running in parallel, such that for each element of the input list will be a corresponding parallel process.

If we consider the `FJ_PowerFunctionExecutor`, this executor implicitly creates a new task that handles the `right_part_function`. So, the number of created tasks grows linearly with the data size. This leads to a logarithmic time-complexity that depends on the *loglen* of the input list.

Adopting this fine granularity of creating a parallel process per element may hinder the performance of the whole program. One possible improvement would be to bound the number of parallel tasks/processes, i.e. to specify a certain level until which a new parallel task is created:

Design choice 8. *Introduce an argument –* `recursion_depth` *– for the* `Executor` *constructors; the default value of this argument is equal to the logarithmic length of the input list (loglen l) and the associated precondition specifies that its value should be less or equal to loglen l. When a new recursive parallel task is created this new task will receive a* `recursion_depth` *decremented with* 1*. The recursion stops when this* `recursion_depth` *reaches zero.*

This solution will lead to a parallel recursive decomposition until a certain level and then each task will simply execute the corresponding *PowerList* function sequentially.

In the same time, there are situations when for a sequential computation of the requested problem, a non recursive variant is more efficient than the recursive one. For example, for *map*, an efficient sequential execution will just iterate through the values of the input list and apply the argument function. The equivalent recursive variant (Eq. 1) is not so efficient since recursion comes with additional costs.

In this case we have to transform the input list by performing a data distribution. A list of length n is transformed into a list of p sub-lists, each having n/p elements. If the sub-lists have the type `BasicList` then the corresponding `BasicListFunction` is called. In the framework, this responsibility is solved by the following design decision:

Design choice 9. *Define a class* `Transformer` *that has the following responsibilities:*

- *transforming a list of atomic elements into a list of sub-lists and,*
- *transforming a list of sub-lists into a list of atomic elements(flat operation).*

How the sub-lists are computed depends on the two operators `tie` *and* `zip`*, and the transformation should preserve the same storage of the elements.*
For the `Transformer` *class implementation the* Singleton *pattern[10] should be used.*

The transformation described above does not imply any element copy and it preserves the same storage container for the list. Every new list created has p `BasicList` elements with the same storage. On creation, the storage information SI is initialized for each new sub-list according to which decomposition operator was used (*tie* or *zip*) to create this new sub-list. The time-complexity associated to this operation is $O(p)$. The `Transformer` class has the following important functions:

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

The execution model for these lists of sub-lists is very similar and only differs for the basic case. If an element of a singleton list, that corresponds to the basic case is a sub-list (i.e. has the `IBasicList` type), a simple sequential execution of the function on that sub-list is called. Sequential execution of functions on sub-lists is implicitly based on recursion which is not very efficient in Java. If an equivalent function defined over `IBasicList` (based on iterations) could be defined, then this should be used instead.

3.5 User Interactor

In order to define a new program/function the user need only to specialize: `combine`; `basic_case`; `create_right_function` and `create_left_function`; as it was described in section 3.2. For the first two functions there are implicit definitions, such that they should be overwritten, only when `combine` is not a simple concatenation, and when `basic_case` is different from identity function.

As specified before, *PowerList* functions don't have to explicitly specify the deconstruction operators since they are determined by the arguments' types; this simplify very very much the definition of new functions. For the parallel execution, different executor types could be used and the user have to choose one depending on the available platform.

In Fig. 11 we present the steps needed to execute the function *map*, which applies square on a list of matrices; the sequential execution based on an iterative list traversal is directed by the use of `BasicList` type, recursive sequential execution is directed by `TiePowerList` (`ZipPowerLis` also could be used), and for parallel multithreading execution an executor based on Java `ForkJoinPool` is created, and the function is executed through it. As it can be noticed from the code represented in Fig. 10, for an MPI execution of a *PowerList* function we need only to specify the 'decorators', and the files' characteristics (if it is the case). The general form of a *Powerlist* function has a list of *PowerLists* arguments. The reading should be possible for any number of *PowerLists* arguments. This is why we have arrays for the files' names and lists' and elements' sizes. For *reduce* we have only one input list.

Design choice 10. *Apply the* Factory Method *pattern [10], in order to simplify the specifications/creation of the most common functions.*

The most common functions, as *map*, *reduce*, or *scan* are provided since they have many applications, and many other functions could be obtained through their composition.

3.6 Metrics Analyser

For testing we have used external scripts (under Linux OS) that allow us to executes several times one program with different parameters: number of MPI processes, number of threads of each process, recursion granularity, and depth of

```
    int limit =1<<5 ; // size of the list
    // function to be applied on each element
    Function f = new SquareFunctionFieldElem<Matrix>(new Matrix(1));
  ArrayList<Matrix> base = new ArrayList<Matrix>(limit); //storage of the list
    // populate the list
      //[...]
//sequential execution
//define the list for sequential computation
    BasicList<Matrix> list = new BasicList<Matrix>(base, 0, limit-1);
//sequential function definition
    BasicListResultFunction<Matrix> bmf =  new Basic.Functions.Map<Matrix>(f, list);
//iterative sequential computation
     Object result = bmf.compute();
//mulithreading execution
    //define the list for parallel computation
     TiePowerList<Matrix> pow_list = new TiePowerList<Matrix>(base,0, limit-1);
  //parallel function definition
     PowerResultFunction<Matrix> mf =
         new Power.Functions.Map<Matrix>(f, pow_list);
  //recursive sequential computation
    Object result = mf.compute();
  //executor definition
     FJ_PowerFunctionExecutor<Matrix> executor = new FJ_PowerFunctionExecutor<Matrix>(mf);
  // parallel multithreading computation
     result = executor.compute();
```

Fig. 11. Sequential and multithreading execution of `Map` function – squaring applied on a list of matrices

the data list (as explained in section 3.4). The performance results were written into files. The parameterization has been done through command line arguments, and so, we may consider that we have used a very simple form of dependency injection.

3.7 Extensions

For *PList*, the functions and their possible multiparadigm executors are defined in a similar way to those for *PowerList*. *PList* is a generalization of *PowerList* allowing the splitting and the composition to be done into/from more than two sub-lists. So, instead of having the two functions: `create_right_function` and `create_left_function`, we need to have an array of (sub)functions. Still, the same principles and patterns are applied as in the *PowerList* case.

*PowerArray*s and *PArray*s are defined similarly to the unidimensional counterparts, and so are their corresponding functions, too. Including them into the framework could be done based on the same principles as those followed for *PList*s.

4 Related work

Algorithmic skeletons are considered an important approach in defining high level parallel models [3, 32]. *PowerList*s and their associated theory could be used as a foundation for a domain decomposition divide-and-conquer skeleton based approach.

There are numerous algorithmic skeleton programming approaches. Most often, they are implemented as libraries for a host language. This languages include functional languages such a Haskell [22] with skeletons implemented using its GpH extension [13]. Multi-paradigm programming languages such as OCaml [24] are also considered: OCamlP3L [5] and its successor Sklml offer a set of a few data and task parallel skeletons and parmap [7]. Although OCaml is a functional, imperative and object oriented language, only the functional and imperative paradigms are used in these libraries.

Object-oriented programming languages such as C++, Java, or even Python are host languages for high-level parallel programming approaches. Very often object-oriented features are used in a very functional programming style. Basically classes for data structures are used in the abstract data-type style, with a type and its operations, sometimes only non-mutable. This is the approach taken by the PySke library for Python [33] that relies on a rewriting approach for optimization [21]. The patterns used for the design of JFPL, are also mostly absent from many C++ skeleton libraries such as Quaff [8] or OSL [18]. These libraries focus on the template feature of C++ to enable optimization at compile time though template meta-programming [38]. Still, there are also very complex C++ skeleton based frameworks – e.g. FastFlow [6] – that are built using a layered architecture and which target networked multi-cores possibly equipped with GPUs systems.

Java is one of the programming languages chosen often for implementing structured parallel programming environments that use skeletons as their foundation. The first skeleton based programming environment developed in Java, which exploits macro-data flow implementation techniques, is the RMI-based *Lithium* [1]. *Calcium* (based on ProActive, a Grid middleware) [2] and *Skandium* [19] (multi-core oriented) are two others Java skeleton frameworks. Compared with the aforementioned frameworks, JPLF could be used on both shared and distributed memory platforms.

Unrelated to architectural concerns, but related to the implementation of JFPL is that Java has been considered as a supported language by some MPI implementations which offer Java bindings. Such implementations are OpenMPI [37] and Intel MPI [41]. There are also 100% pure Java implementations of MPI such as MPJ Express [35, 14]. Although there are some syntactic differences between them, all of these implementations are suitable for MPI execution. We have also used Intel Java MPI and MPJ Express and the obtained results were similar.

There are many works that emphasize the need of using well defined software engineering concepts and methodologies for increasing the reliability and productivity in parallel software development [15, 16, 31, 23]. They refer either methodologies as I. Foster in [9], or patterns as the high impact book "A Pattern Language for Parallel Programming" [23], or both. Structured approaches are necessary since the technologies are various, there are many execution platforms, and also, the parallel software development is difficult.

5 Conclusions and further work

Starting from an analysis of the requirements for a reliable parallel programming framework, we tried to identify the main components of such a framework and we arrived to an architecture that is based on a model of parallel computation, but contains also well defined "hot-spot" as components – MEDUGA(Model-Executors-DataManager-UserInteractor-GranularityBalancer-metricsAnalyser).

We emphasized also how this was applied on the development of a concrete framework - JPLF.

The JPLF framework has been architectured using design patterns. Based on the proposed architecture, new concrete problems can be easily implemented and resolved in parallel. Also, the framework could be easily extended with additional data structures (such as ParList or PowerArray [17]).

The most important benefit of the framework's internal architecture is that the parallel execution is controlled independently of the *PowerList* function definition. Primitive operations are the foundation for the executors' definitions, this allowing multiple execution variants for the same *PowerList* program. For example, sequential execution, MPI execution, multithreading using `ForkJoinPool` execution or some other execution model can be easily implemented. If we have a definition of a PowerList function we may use it for multithreading or MPI execution without any other specific adaptation of that particular function.

For the MPI computation model it was mandatory to properly manage the computation steps of a *PowerList* function: *descend, leaf*, and *ascend*. These computation steps were defined within a *Decorator* pattern based approach.

Many frameworks are oriented either on shared memory or on distributed memory platforms. The possibility to use the same base of computation and associate then the execution variants depending on the concrete execution systems brings important advantages.

The separation of concerns principle has been intensively used. This facilitated the data-structures' behavior to be separated from their storage, and to ensure the separation of the definition of functions from their execution.

As a further work we propose to enhance the metrics analyser component of the framework by allowing the injection of some metrics evaluation into the computation. Through this, the computation would be to be augmented with the required metrics computation.
Several executions have to be done, overhead regions identification could improve the performance very much, resource utilization evaluation (e.g. number of threads that are created/used) may improve the efficiency, etc.

As we presented in section 4 there are many parallel programming libraries (that could be assimilated to frameworks), which are based on skeletons, and which provide implementations of the considered parallel skeletons on different systems. It would be interesting to investigate the measure in which they have been built following software engineering methodologies that assure the expected levels of software quality. How this aspects were affected performance but also maintainability is another interesting subject of study.

References

1. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Generation Comp. Syst. **19**(5), 611–626 (2003)
2. Caromel, D., Leyton, M.: Fine tuning algorithmic skeletons. In: Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings. pp. 72–81 (2007)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA, USA (1991)
4. Cooley, J., Tukey, J.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation **19**(90), 297–301 (1965)
5. Cosmo, R.D., Li, Z., Pelagatti, S., Weis, P.: Skeletal Parallel Programming with OcamlP3l 2.0. Par. Proc. Letters **18**(1), 149–164 (2008)
6. Danelutto, M., Torquati, M.: Structured parallel programming with "core" fastflow. Central European Functional Programming School. CEFP 2013. Lecture Notes in Computer Science **8606**, 29–75 (2015)
7. Di Cosmo, R., Danelutto, M.: A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In: Proceedings of the InternationalConference on Computational Science. vol. 9, pp. 1837–1846. Elsevier (2012)
8. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: Efficient C++ Design for Parallel Skeletons. Parallel Computing **32**, 604–615 (2006)
9. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc. (1995)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
11. Gorlatch, S., Lengauer, C.: Abstraction and performance in the design of parallel programs: An overview of the sat approach. Acta Inf. **36**(9–10), 761–803 (Jan 2000)
12. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison Wesley (2003)
13. Hammond, K., Portillo, Á.J.R.: Haskskel: Algorithmic skeletons in haskell. In: IFL. LNCS, vol. 1868, pp. 181–198. Springer (1999)
14. Javed, A., Qamar, B., Jameel, M., Shafi, A., Carpenter, B.: Towards scalable Java HPC with hybrid and native communication devices in MPJ Express. International Journal of Parallel Programming **44**(6), 1142–1172 (2016). https://doi.org/10.1007/s10766-015-0375-4
15. Jelly, I., Gorton, I.: Software engineering for parallel systems. Information and Software Technology **36**(7), 381 – 396 (1994), software Engineering for Parallel Systems
16. Kiefer, M.A., Warzel, D., Tichy, W.: An empirical study on parallelism in modern open-source projects. In: SEPS 2015 (2015)
17. Kornerup, J.: Data Structures for Parallel Recursion. Ph.d. dissertation, University of Texas (1997)
18. Légaux, J., Loulergue, F., Jubertie, S.: OSL: an algorithmic skeleton library with exceptions. In: Proceedings of the International Conference on Computational Science. pp. 260–269. Elsevier, Barcelona, Spain (2013)
19. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 289–296. IEEE Computer Society (2010)

20. Lopes, S.F., Afonso, F., Tavares, A., Monteiro, J.: Framework characteristics - a starting point for addressing reuse difficulties. 2009 Fourth International Conference on Software Engineering Advances pp. 256–264 (2009)
21. Loulergue, F., Philippe, J.: Automatic Optimization of Python Skeletal Parallel Programs. In: Algorithms and Architectures for Parallel Processing (ICA3PP). pp. 183–197. LNCS, Springer, Melbourne, Australia (2019)
22. Marlow, S. (ed.): Haskell 2010 Language Report (2010), https://www.haskell.org/definition/haskell2010.pdf
23. Massingill, B.L., Mattson, T.G., Sanders, B.A.: A Pattern Language for Parallel Programming. Addison Wesley, Software Patterns Series (2004)
24. Minsky, Y.: OCaml for the masses. Commun. ACM **54**(11), 53–58 (2011)
25. Misra, J.: Powerlist: A structure for parallel recursion. ACM Trans. Program. Lang. Syst. **16**(6), 1737–1767 (1994)
26. Niculescu, V.: Pares – a model for parallel recursive programs. Romanian Journal of Information Science and Technology (ROMJIST) pp. 159–182 (2012)
27. Niculescu, V., Bufnea, D., Sterca, A.: MPI scaling up for powerlist based parallel programs. In: 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019. pp. 199–204. IEEE (2019)
28. Niculescu, V., Loulergue., F.: Transforming powerlist based divide&conquer programs for an improved execution model. J. Supercomput **76** (2020)
29. Niculescu, V., Loulergue, F., Bufnea, D., Sterca, A.: A Java framework for high level parallel programming using powerlists. In: 18th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2017, Taipei, Taiwan, December 18-20, 2017. pp. 255–262. IEEE (2017)
30. Niculescu., V., Loulergue., F., Bufnea., D., Sterca., A.: Pattern-driven design of a multiparadigm parallel programming framework. In: Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,. pp. 50–61. INSTICC, SciTePress (2020)
31. Pankratius, V.: Software engineering in the era of parallelism. In: KIT-Nachwuchswissenschaftler-Symposium (2010)
32. Pelagatti, S.: Structured Development of Parallel Programs. Taylor & Francis (1998)
33. Philippe, J., Loulergue, F.: PySke: Algorithmic skeletons for Python. In: International Conference on High Performance Computing and Simulation (HPCS). pp. 40–47. IEEE (2019)
34. Pressman, R.: Software Engineering:A Practitioner's Approach, 7th edition. McGraw-HillScience (2009)
35. Qamar, B., Javed, A., Jameel, M., Shafi, A., Carpenter, B.: Design and implementation of hybrid and native communication devices for Java HPC. In: Proceedings of ICCS'2014, Cairns, Queensland, Australia, 10-12 June, 2014. pp. 184–197 (2014)
36. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Comput. Surv. **30**(2), 123–169 (Jun 1998)
37. Vega-Gisbert, O., Román, J.E., Squyres, J.M.: Design and implementation of Java bindings in Open MPI. Parallel Computing **59**, 1–20 (2016)
38. Veldhuizen, T.: Techniques for Scientific C++. Computer science technical report 542, Indiana University (2000)
39. X***: Mpi: A message-passing interface standard. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, accessed: 20-November-2019

40. X***: Oracle: The Java tutorials: ForkJoinPool. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, accessed: 20-November-2019
41. X***: Intel MPI library developer reference for Linux OS: Java bindings for MPI-2 routines (2019), https://software.intel.com/en-us/mpi-developer-reference-linux-java-bindings-for-mpi-2-routines, accessed: 20-November-2019