Pattern-driven Design of a Multiparadigm Parallel Programming Framework

Virginia Niculescu[1][a], Frédéric Loulergue[2,3][b], Darius Bufnea[1][c] and Adrian Sterca[1][d]
[1]*Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania*
[2]*School of Informatics, Computing and Cyber Systems, Northern Arizona University, USA*
[3]*Univ. Orleans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France*
{*vniculescu, bufny, forest*}*@cs.ubbcluj.ro, Frederic.Loulergue@nau.edu*

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

Abstract: Parallel programming is more complex than sequential programming. It is therefore more difficult to achieve the same software quality in a parallel context. High-level parallel programming approaches are intermediate approaches where users are offered simplified APIs. There is a trade-off between expressivity and programming productivity, while still offering good performance. By being less error-prone, high-level approaches can improve application quality. From the API user point of view, such approaches should provide ease of programming without hindering performance. From the API implementor point of view, such approaches should be portable across parallel paradigms and extensible. JPLF is a framework for the Java language based on the theory of Powerlists, which are parallel recursive data structures. It is a high-level parallel programming approach that possesses the qualities mentioned above. This paper reflects on the design of JPLF: it explains the design choices and highlights the design patterns and design principles applied to build JPLF.

1 INTRODUCTION

Even if nowadays parallel programming is used in almost all software applications, writing correct parallel programs from scratch is very often a difficult task. The designers of parallel programming APIs or languages face three conflicting challenges. First, they need to provide the users an API that is as easy as possible to use, and as high-level as possible to make programmers productive in writing quality software. Secondly, they need to provide an API that offers good performances. Finally, as parallel architectures are numerous and evolve, they need to provide an API that is flexible enough to accommodate change in the supported parallel paradigms.

We have been designing and developing a Java API, named JFPL, for Java Framework for Power Lists (Niculescu et al., 2017; Niculescu et al., 2019). By being based on the PowerList theory introduced by J. Misra (Misra, 1994), JFPL is a high-level parallel programming framework that allows building parallel programs that follow multi-way divide and conquer parallel programming patterns with good execution performances both on shared and distributed memory architectures.

The shared memory execution environment is based on thread pools where the size of these pools implicitly depends on the system where the execution takes place. The current implementation uses a Java `ForkJoinPool` executor (Oracle,), but others could be used too. For distributed memory systems, we use MPI (Message Passing Interface) (MPI,) in order to distribute processing units on computing nodes. However, MPI-based computations imply a different parallel programming paradigm. JFPL therefore supports both multi-threading in a shared memory context and multi-processing in a distributed memory context.

Allowing the support of multiple paradigms requires the API to be flexible and extensible. The framework was implemented following object-oriented design principles in order to possess these characteristics. Specifically, we have employed separations of concerns in order to facilitate changing the low-level storage and the parallel execution environ-

[a] https://orcid.org/0000-0002-9981-0139
[b] https://orcid.org/0000-0001-9301-7829
[c] https://orcid.org/0000-0003-0935-3243
[d] https://orcid.org/0000-0002-5911-0269

ment. In order to overcome the challenges brought by the multiparadigm support, we have used different design patterns, decoupling patterns having a defining role.

Contribution: While our previous work was more focused on presenting the API and performances, the contribution of this paper is a reflection on the design of the API. The paper details the design choices made to build JFPL and the rationale behind them.

Outline: The remaining of the paper is organized as follows. In Section 2, we give an overview of PowerLists. Section 3 is devoted to a complex analysis of the JFPL framework design and implementation. Related work is discussed in Section 4. We conclude in Section 5.

2 AN OVERVIEW OF POWERLIST THEORY

The *PowerList* data structure and its associated theory have been introduced by J. Misra (Misra, 1994). PowerLists provide a high level of abstraction especially because the array index notations are not used.

PowerLists are homonogeously typed sequences of elements. The size of a PowerList is a power of two. A *singleton* is a PowerList containing a single element. It is denoted by $[v]$ for a value v. Two PowerLists of the same size and same type for elements are *similar*.

Two different constructors exist to combine two similar PowerLists. The resulting PowerList has a size that is the double of the size of the input PowerLists:

- the operator *tie* yields a PowerList containing the elements of p followed by elements of q;
- the operator *zip* returns a PowerList containing alternatively the elements of p and q.

tie is denoted by $|$, and *zip* by $\natural$.

The functions on Powerlists are recursive functions defined based on a structural induction. Usually, the base case considers the singleton list. The recursive case may use either $|$, or $\natural$, or both for decomposing and composing the PowerLists.

The higher-order function *map*, applies a scalar function f_s to each element of a PowerList. It can be defined as follows:

$$\begin{cases} map(f_s,\ [v]) & = & [f_s(v)] \\ map(f_s,\ pl_1 \,|\, pl_2) & = & map(f_s,\ pl_1) \,|\, map(f,\ pl_2) \end{cases}$$

Another correct definition of *map* could be also obtained in a similar way, by using $\natural$ instead of $|$.

Considering an associative operator $\oplus$, the reduction of a PowerList can be defined as:

$$\begin{cases} red(\oplus,\ [v]) & = & [v] \\ red(\oplus,\ pl_1 \,|\, pl_2) & = & red(\oplus,\ pl_1) \oplus (red(\oplus,\ pl_2) \end{cases}$$

and *red* could be defined using $\natural$, too.

The implementation of the Fast Fourier Transform (FFT) algorithm (Cooley and Tukey, 1965) on PowerLists requires both $|$ and $\natural$, and the FFT computations on Powerlists needs $O(n \log n)$ steps:

$$\begin{cases} fft([x]) & = & [x] \\ fft(p \natural q) & = & (F_p + u \times F_q) \,|\, (F_p - u \times F_q) \end{cases}$$

where $F_p = fft(p)$, $F_q = fft(q)$, $u = powers(p)$, and $+, -$ are the correspondent associative binary operators extended to Powerlists. $powers(p)$ is the PowerList $[w^0, w^1, .., w^{|p|-1}]$ where $|p|$ denotes the size of p and w denotes the $2n^{\text{th}}$ principal root of 1.

The parallelism in all these functions is *implicit*. Each application of an operator (*zip* or *tie*), used to *deconstruct* the input PowerList, implies two independent computations that may be independently performed in parallel.

Having both $|$ and $\natural$ makes the PowerLists theory different from other theories of lists. Many parallel algorithms benefit from being expressible using both operators. However, the increased expressive power offered to users of an API based on these theories, induces some difficulties when these high-level constructs have to be implemented.

The considered class of PowerList functions is characterized by the fact that the functions could be computed recursively based on their values on the split argument lists. This class includes a broad class of functions as it has been proved in (Misra, 1994) and (Kornerup, 1997): sorting algorithms (Batcher, Bitonic, Odd-Even), prefix sum, Gray codes, to cite a few.

PowerLists are restricted to lists whose length is a power of two. The PList extension of PowerLists (Kornerup, 1997) lifts this restriction. PLists are also defined based on *tie* and *zip* operators. For PLists, both *tie* and *zip*, as constructors, are generalized to take as arguments an arbitrary number of similar PLists. The functions defined on PLists are similarly defined as those over PowerLists. The difference is that they receive in addition a list of arities. Each time an input PList needs to be split, the first element of the list of arities, gives the number of PLists that should be created. The product of all these arities should be equal to the length of the PList argument, and the length of this list gives the recursion depth.

We write $\bar{n}$ the set of numbers i such that $0 \leq i < n$. For n similar PLists p_i, with $i \in \bar{n}$, the generalized

tie is written $|_{i\in\bar{n}}\, p_i$, and the generalized *zip* is written $\natural_{i\in\bar{n}}\, p_i$.
The *map* function on PLists can then be defined as:

$$\begin{cases} map(f,\ [],\ [a]) & = & [f(a)] \\ map(f,\ n::l, |_{i\in\bar{n}}\, p_i) & = & |_{i\in\bar{n}} map(f,\ l,\ p_i) \end{cases}$$

where $n::l$ denotes the list with head element n, and with tail l (a list).

3 JPLF FRAMEWORK DESIGN

The framework's design is based on the PowerList theory, mainly on the types that this theory defines, but also on the specific operations and properties that these types have.

The main components of the framework are interconnected, although they have different responsibilities, such as:

- data structures implementations,
- functions implementations,
- functions executors.

Design choice 1. *Impose separate definitions for these components allowing them to vary independently.*
The idea behind 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. Other data structures such as PList are defined similarly as specializations of `BasicList` class.

3.1 PowerList Implementations

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 *(Gamma et al., 1995) to decouple the definition of the special lists' types from their storage. Storage could be a classical predefined list container.*

The storage is a container object 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 distance could exist between the locations of the two elements.

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 of two similar sub-lists but the initial list storage remains 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 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 $(0 \le k < p)$ 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`

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 to be 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.

The corresponding list data structure types are depicted in the class diagram shown in Figure 1.

3.2 PowerList Functions

The structure of the computation for a PowerList function is expressed by specifying *tie* or *zip* decon-

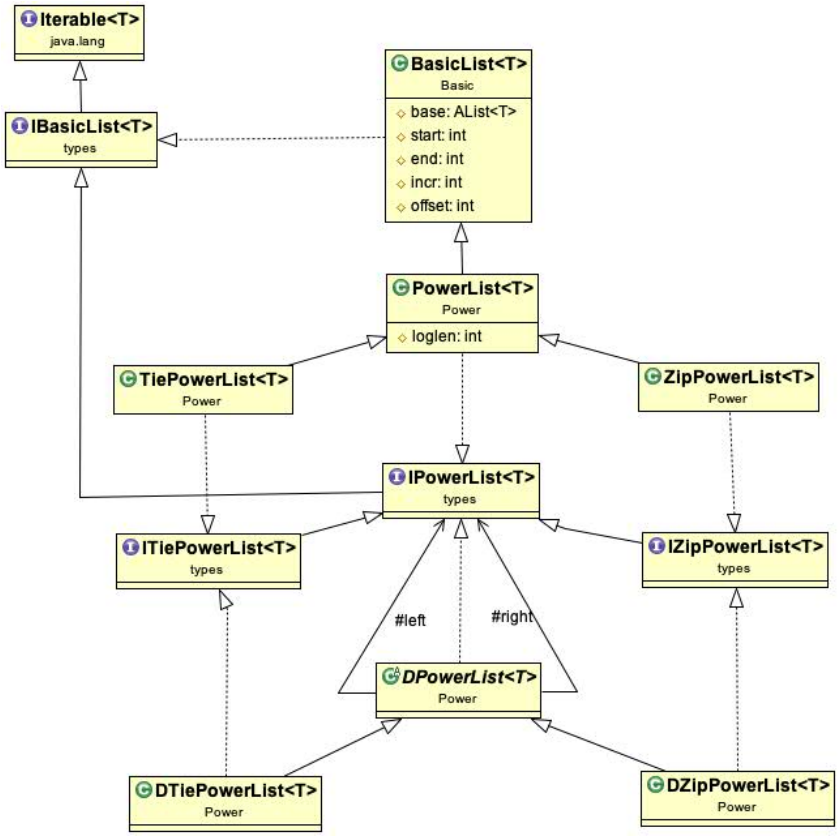

Figure 1: The class diagram of the classes corresponding to lists implementation.

struction operators for splitting the PowerList arguments and by the composing operator in case the result is also a PowerList.

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, also the same operator is 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 `TiePowerList`s and the *zip* operator is used in case the type is `ZipPowerList`s. 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 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 3. *In order to allow the implementation of the divide and conquer functions over PowerLists, use the* Template Method *design pattern (Gamma et al., 1995).*

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 below:

```
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;
}
```

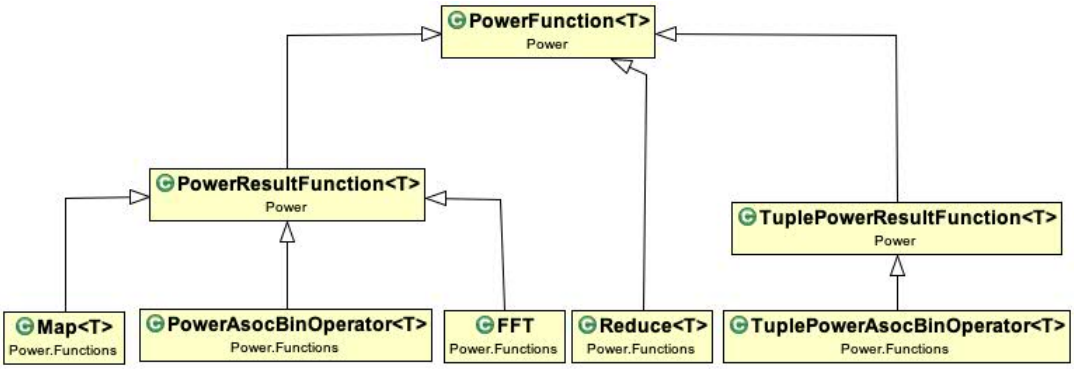

Figure 2: The class diagram of classes corresponding to functions on PowerLists and their execution.

The primitive methods:

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

are the only ones that need an implementation for the definition of a new function.

The functions `create_right_function` and `create_left_function` specialized implementations should be provided to guarantee that the new created functions correspond to the specific definition. For the other two, there are implicit definitions to release the user to provide implementations for all. 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` automatically verifies if the PowerList argument is a singleton, as per the specification of the PowerList theory. There is also the possibility to override this method and force to end the recursion before singleton lists are encountered.

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

Figure 2 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 has 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 stream of computation. This has been used for the FFT computation case.

3.3 Multithreading Executors

The basic sequential execution of a PowerList function is done simply by invoking the corresponding `compute` method.

In order to allow further modification or specialization, the definition of the parallel execution of a PowerList function is done separately. `IPowerFunctionExecutor` is the type 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 divide and conquer pattern could be used for such an execution.

Design choice 4. *Define separate executor classes that rely on the same operations as the primitive methods used for the PowerList function definition.*

The class `FJ_PowerFunctionExecutor` implementation relies on the `ForkJoinPool` Java executor, which is an implementation of the `ExecutorService` interface. This class has been designed to be used for computation that can be split recursively into smaller batches.

Other implementations can be easily developed, in order to allow the usage of other executors. Figure 3 shows the implemented classes corresponding to the multithreading executions based on `ForkJoinPool`.

The simple definition of the recursive tasks that we choose to execute in parallel is enabled by this executor: new tasks are created each time a split operation is done.

As the PowerLists functions are built

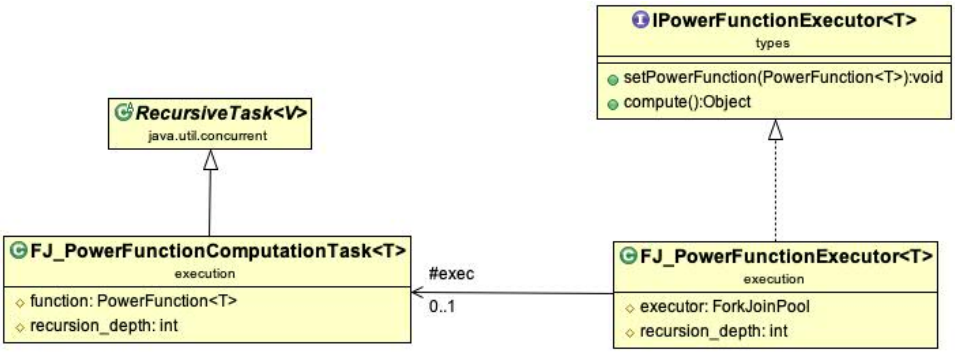

Figure 3: The class diagram of classes corresponding to the multithreading executions based on ForkJoinPool.

```
public Object compute() {
 Object result =null;
 if (function.te st_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;
}
```

Figure 4: `compute` for `PowerFunctionComputationTask`

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 4.

In this example, separate execution tasks wraps 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.

3.4 MPI Execution

There is an obvious need for scalability for a framework that works with regular data sets of very large sizes. The ability to use multiple cluster nodes could be attained by introducing MPI based execution of the functions (Niculescu et al., 2019).
The command for launching a MPI execution has in general the following form:

```
mpirun -n 20 TestPowerListReduce_MPI.class
```

where the `-n` argument defines the number of MPI processes (20 is just an example) that are going to be created.

So, 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` that are used by the implementation code.

In the case of the execution on shared memory systems, and so based on multithreading, the list splitting and combining operations were reduced to a constant time $O(1)$ since only the storage information $SI(l)$ characteristics of the new created lists should be computed.

On a distributed memory system, based on an MPI execution, the list splitting and combining costs could not be kept small because data communication between processes is needed. Since the cost for data communication is much higher than the simple computation costs, we had to analyze very carefully when these communications could be avoided.

PowerList functions are recursively defined on list data structures, and each time we apply the definition 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 also could need communication, since for combining stage we need to

apply operations on the corresponding results of the two recursive calls.

In order to identify 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 of 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 to distribute 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 (Niculescu and Loulergue., 2018).

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 obtained the final result.

   Examples: *reduce*, *fft*.

One direct solution to treat these classes of function 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 5. *Decompose the execution of the PowerList function into phases: reading, splitting, leaf, combining, and writing.*

Apply the Template method *pattern (Gamma et al., 1995) in order to allow the specified phases to vary independently. Apply the* Decorator *pattern (Gamma et al., 1995) in order to add specific corresponding cases.*

The Figure 5 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 6. The class `MPI_CTOperations` provides a `compute` template method and empty implementations for the different step operations: `read`, `split`, `compose`, ...

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

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

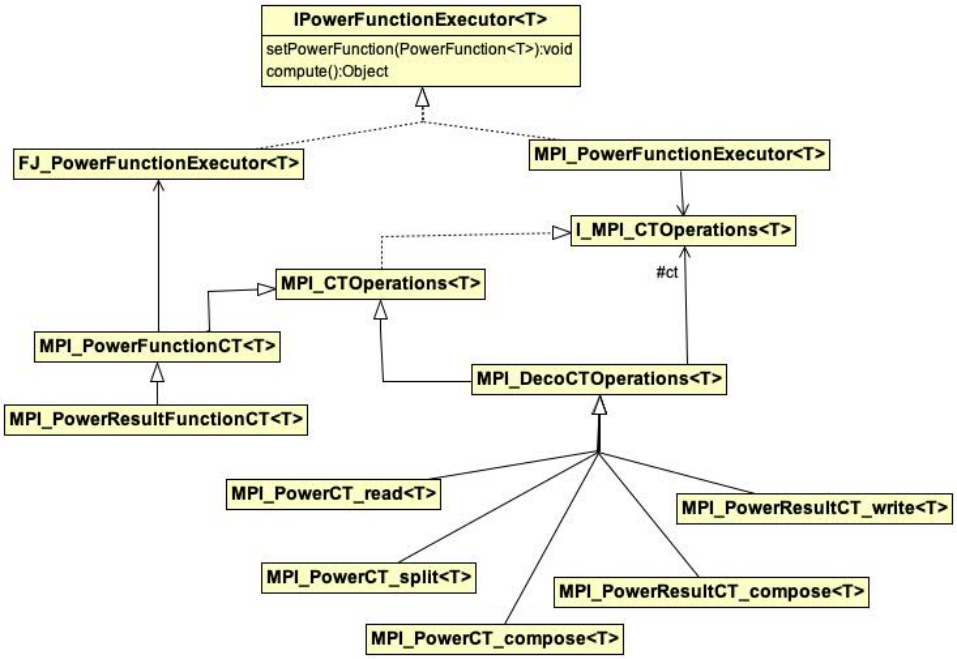

Figure 5: The classes used for different types of execution of a PowerList function.

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 is 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 to have 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 6. *Use the* Broker *design pattern in order to define specialized classes for reading and writing data (`FileReaderWriter`) and for serializing/deserializing the data (`ByteSerialization`).*

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 also 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

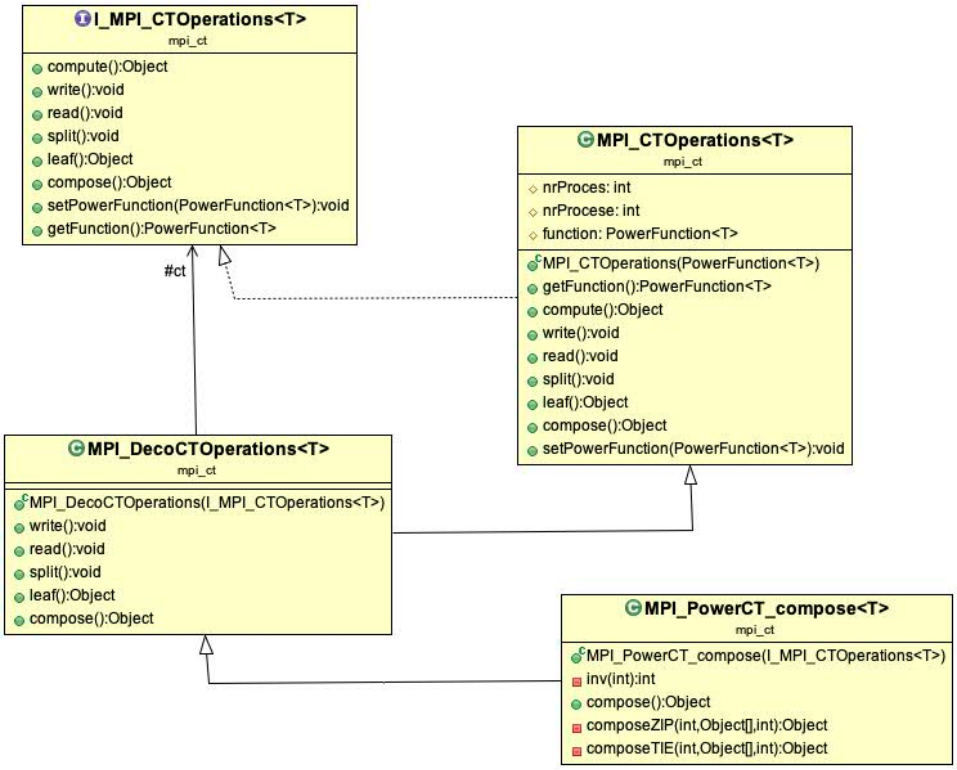

Figure 6: Implementation details of some of the classes involved in the definition of MPI execution.

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.

In order to better explain the MPI execution we will consider the case of the `Reduce` function (Section 2). The following test case considers a reduction on a list of matrices using addition. The code snippet in Figure 7 emphasizes what is needed for the MPI execution of the `Reduce` function.

As it can be noticed from the code, 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.

```
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();
```

Figure 7: The `Reduce` function

Design choice 7. *Apply the* Factory Method *pattern (Gamma et al., 1995), in order to simplify the specifications/creation of the most common functions.*

3.5 Increasing Granularity

Ideally, describing parallel programs using 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, so that for each element of the input list there is 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.

However, in many situations, 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, 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.

Still, 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. 2) 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 considered 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 should be used (Gamma et al., 1995).*

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 `IPowerList` type), a simple sequential execution of the function on that sub-list is called.

In our framework, sequential execution of functions on sub-lists is 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 will be used instead.

Remark. 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 are applied as in the PowerList case.

4 RELATED WORK

Algorithmic skeletons are considered an important approach in defining high level parallel models (Cole, 1991; Pelagatti, 1998). PowerLists 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 (Marlow, 2010) with skeletons implemented using its GpH extension (Hammond and Portillo, 1999). Multi-paradigm programming languages such as OCaml (Minsky, 2011) are also considered: OCamlP3L (Cosmo et al., 2008) and its successor Sklml offer a set of a few data and task parallel skeletons and parmap (Di Cosmo and Danelutto, 2012). Although OCaml is a functional, imperative and object oriented language, only the functional and imperative paradigms are used in these libraries.

Of course, objected oriented programming languages such as C++, Python and Java are host languages for high-level parallel programming approaches. 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 (Philippe and Loulergue, 2019) that relies on a rewriting approach for optimization (Loulergue and Philippe, 2019). The patterns used for the design of JFPL, are also mostly absent from many C++ skeleton libraries such as Quaff (Falcou et al., 2006) or OSL (Légaux et al., 2013). These libraries focus on the template feature of C++ to enable optimization at compile time though template meta-programming (Veldhuizen, 2000). Still, there are also very complex C++ skeleton based frameworks – e.g. FastFlow (Danelutto and Torquati, 2015) – that are built using a layered architecture and which target networked multi cores possibly equipped with GPUs systems.

One of the programming languages suitable for implementing structured parallel programming environments that use skeletons as their foundation is Java. The first skeleton based programming environment developed in Java, which exploits macro-data flow implementation techniques, is the RMI-based *Lithium* (Aldinucci et al., 2003). *Calcium* (based on ProActive, a Grid middleware) (Caromel and Leyton, 2007) and *Skandium* (Leyton and Piquer, 2010) (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 (Vega-Gisbert et al., 2016) and Intel MPI (Intel, 2019). There are also 100% pure Java implementations of MPI such as MPJ Express (Qamar et al., 2014; Javed et al., 2016). Although there are some syntactic differences between them, all of these implementations are suitable for MPI execution. In our past experiments, we used Intel Java MPI and MPJ Express and the obtained results were similar.

5 CONCLUSIONS

The framework presented in this paper has been architectured using design patterns. Based on this architecture, new concrete problems can be easily implemented and resolved. Also, the framework could be easily extended with additional data structures (such as ParList or PowerArray (Kornerup, 1997)).

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 for 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.

REFERENCES

Aldinucci, M., Danelutto, M., and Teti, P. (2003). An advanced environment supporting structured parallel programming in Java. *Future Generation Comp. Syst.*, 19(5):611–626.

Caromel, D. and Leyton, M. (2007). Fine tuning algorithmic skeletons. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes,*

France, August 28-31, 2007, Proceedings, pages 72–81.

Cole, M. (1991). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA.

Cooley, J. and Tukey, J. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301.

Cosmo, R. D., Li, Z., Pelagatti, S., and Weis, P. (2008). Skeletal Parallel Programming with OcamlP3l 2.0. 18(1):149–164.

Danelutto, M. and Torquati, M. (2015). Structured parallel programming with "core" fastflow. *Central European Functional Programming School. CEFP 2013. Lecture Notes in Computer Science*, 8606:29–75.

Di Cosmo, R. and Danelutto, M. (2012). A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In *International Conference on Computational Science (ICCS)*, volume 9, pages 1837–1846. Elsevier.

Falcou, J., Sérot, J., Chateau, T., and Lapresté, J.-T. (2006). Quaff: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32:604–615.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hammond, K. and Portillo, Á. J. R. (1999). Haskskel: Algorithmic skeletons in haskell. In *IFL*, volume 1868 of *LNCS*, pages 181–198. Springer.

Intel (2019). Intel MPI library developer reference for Linux OS: Java bindings for MPI-2 routines. Accessed: 20-November-2019.

Javed, A., Qamar, B., Jameel, M., Shafi, A., and Carpenter, B. (2016). Towards scalable Java HPC with hybrid and native communication devices in MPJ Express. *International Journal of Parallel Programming*, 44(6):1142–1172.

Kornerup, J. (1997). *Data Structures for Parallel Recursion*. Ph.d. dissertation, University of Texas.

Légaux, J., Loulergue, F., and Jubertie, S. (2013). OSL: an algorithmic skeleton library with exceptions. In *International Conference on Computational Science (ICCS)*, pages 260–269, Barcelona, Spain. Elsevier.

Leyton, M. and Piquer, J. M. (2010). Skandium: Multicore programming with algorithmic skeletons. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 289–296. IEEE Computer Society.

Loulergue, F. and Philippe, J. (2019). Automatic Optimization of Python Skeletal Parallel Programs. In *Algorithms and Architectures for Parallel Processing (ICA3PP)*, LNCS, pages 183–197, Melbourne, Australia. Springer.

Marlow, S., editor (2010). *Haskell 2010 Language Report*.

Minsky, Y. (2011). OCaml for the masses. *Communications of the ACM*, 54(11):53–58.

Misra, J. (1994). Powerlist: A structure for parallel recursion. *ACM Trans. Program. Lang. Syst.*, 16(6):1737–1767.

MPI. Mpi: A message-passing interface standard. `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`. Accessed: 20-November-2019.

Niculescu, V., Bufnea, D., and Sterca, A. (2019). 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*, pages 199–204. IEEE.

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

Niculescu, V., Loulergue, F., Bufnea, D., and Sterca, A. (2017). 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*, pages 255–262. IEEE.

Oracle. The Java tutorials: ForkJoinPool. `https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`. Accessed: 20-November-2019.

Pelagatti, S. (1998). *Structured Development of Parallel Programs*. Taylor & Francis.

Philippe, J. and Loulergue, F. (2019). PySke: Algorithmic skeletons for Python. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 40–47. IEEE.

Qamar, B., Javed, A., Jameel, M., Shafi, A., and Carpenter, B. (2014). Design and implementation of hybrid and native communication devices for Java HPC. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, pages 184–197.

Vega-Gisbert, O., Román, J. E., and Squyres, J. M. (2016). Design and implementation of Java bindings in Open MPI. *Parallel Computing*, 59:1–20.

Veldhuizen, T. (2000). Techniques for Scientific C++. Computer science technical report 542, Indiana University.