MPI Scaling Up for Powerlist Based Parallel Programs

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

***Abstract*—**

Powerlists are recursive data structures that together with their associated algebraic theories could offer both a methodology to design parallel algorithms and parallel programming abstractions to ease the development of parallel applications This has been also proved by a concrete development of such a framework that allows easy, efficient, and reliable implementation of Java parallel programs on shared memory systems.

The paper presents a highly scalable version of this framework by extending it to distributed memory systems based on an MPI implementation. Through this extension we may use the framework to develop Java parallel programs also on distributed memory systems such as clusters. The design of the framework enables flexibility in defining the appropriate execution type depending on the execution system and its characteristics. Therefore, it is possible to choose MPI execution (that also could be combined with multithreading) if the available system includes an MPI platform, or simple multithreading execution.

Examples are given and performance experiments are conducted. The performance analysis of these applications emphasises the utility and the efficiency of this framework extension.

***Index Terms*—parallel programming; scaling; recursive structures; Java; MPI; performance; models.**

I. Introduction

Since nowadays Java could be accepted as an alternative for High Performance Computing, Java parallel programming frameworks could be developed to achieve very high levels of performance and scalability. Such a framework is *JPLF* [11], which is based on *powerlist* data structures and multithreading programming. We present in this paper an MPI extension of this framework in order to allow execution also on cluster architectures. The multithreaded variant is based on using thread pools (the tested implementation uses `ForkJoinPool` executor, but others could be used too) where the size of these pools are depended on the system on which it is executed. By moving to a cluster type system, we may increase the level of parallelization by involving many more processing units into executing the same application.

In this paper we presents the design decisions on which this MPI extension is based, together with applications and experiments that emphasize the advantages introduced by this extension.

The paper is organized as it follows. In section II we give a general description of powerlists, and then some general aspects about the framework design and implementation are given in section III. Section IV presents the MPI extension of the framework that supports Java implementations of *powerlists* parallel programs. Section V presents some applications and the practical experiments related to them, together with the associated performance analysis. Related work is presented in section VI, and we present the conclusions section VII.

II. Powerlist Theory

The powerlists data structures and their associated theory have been introduced by J. Misra [10], and especially because the index notations are not used, they allow defining divide&conquer programs at a high level of abstraction. The functions on powerlists are defined recursively by splitting their arguments based on two deconstruction operators. A powerlist is a linear data structure with elements of the same type, with the specific characteristic that the length of a powerlist is always a power of two.

We denote by $PL\langle X, n\rangle$ the type of a powerlist that has 2^n elements each being of type X. A powerlist with a single element a is called a *singleton*, and it is denoted by $[a]$. Two powerlists are called *similar* if they have the same length and elements of the same type.

Two *similar* powerlists can be combined into a new, double length, powerlist 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.

Therefore, the constructor operators for powerlists are:

$$\begin{array}{lcl} [.] & : & X \to PL\langle X, 0\rangle \\ .|. & : & PL\langle X, n\rangle \times PL\langle X, n\rangle \to PL\langle X, n+1\rangle \\ .\natural. & : & PL\langle X, n\rangle \times PL\langle X, n\rangle \to PL\langle X, n+1\rangle \end{array} \quad (1)$$

Powerlist algebra is defined by these operators and by axioms that assure the existence of a unique decomposition of a powerlist, using one of *tie* or *zip* operators; and the fact that the *tie* and *zip* operators commute. The proofs of properties on powerlists are based on a structural induction principle defined on powerlists, which consider a base case (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, or a combination of two powerlists, a function on a powerlist can be defined recursively by cases. For example, the high order function map, which

applies a scalar function to each element of a powerlist, is defined as follows:

$$\begin{array}{lcl} map(f,\ [a]) & = & [f(a)] \\ map(f,\ p\,|\,q) & = & map(f,\ p)\,|\,map(f,\ q) \end{array} \quad (2)$$

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

$$\begin{array}{lcl} red(\oplus,\ [a]) & = & [a] \\ red(\oplus,\ p\,|\,q) & = & red(\oplus,\ p) \oplus (red(\oplus,\ q) \end{array} \quad (3)$$

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

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

$$\begin{array}{lcl} inv:\ PL\langle X,n\rangle \to PL\langle X,n\rangle & & \\ inv([a]) & = & [a] \\ inv(p\,|\,q) & = & inv(p) \natural inv(q) \end{array} \quad (4)$$

Many other functions, for example the Fast Fourier Transform (section V-C) benefit from the existence of the two operators *tie* and *zip*.

The parallelism of the functions is implicit: each application of a deconstruction operator (*zip* or *tie*) implies two independent computations that may be performed in two processes (programs) that could run in parallel.

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

III. The PowerList Java Framework

The *JPLF* framework provides general support implementations for computing powerlist functions in Java. The design of this framework was guided by the types defined in the powerlist theory and by their specific properties and operations.

The two characteristic operations: *tie* and *zip* are used to split a powerlist, but they could also be used as constructors. The theory considers that the powerlist functions are defined by applying one deconstruction operator (for divide operation) for each input powerlist, and a combining function. If the result is also a powerlist, then the combining function is based on a powerlist construction operator, too. The differentiation between these two cases is done by considering the following two types: `PowerFunction` and `PowerResultFunction`.

The operator type used for splitting a powerlist is determined by the specific type of that powerlist that could be either `TiePowerList` or `ZipPowerList`; and this enables polymorphic definitions of the splitting and combining operations. The associated class diagram that corresponds to the list data structure types is shown in Figure 1.

When a powerlist is split, the result is formed by two similar sublists. In order to avoid element copy, the storage of both sublists remains the same as that of the initial list, and only the *storage information* is updated. For

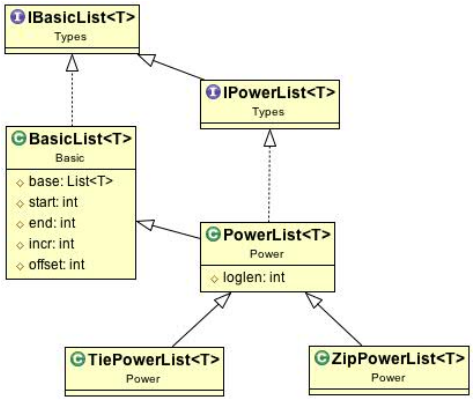

Fig. 1: The class diagram of the classes corresponding to lists implementation.

each list `l`, the *storage information* `SI(l)` is formed of: `[base, start, end,incr]` where `base` is the reference to the storage container, `start` is the start index,`end` is the end index, and `incr` the incrementation step, which is used for list iteration.

From a given list with storage information `SI(list)` being `{base, start, end, incr}`, *tie* and *zip* deconstruction operators each create two lists `left_list` and `right_list` that have the same storage container – `base`, and correspondent updated values for `(start, end, incr)`.

The definition of the divide&conquer functions over powerlists is done based on the *template method* design pattern [7]. The `PowerFunction` class defines the template method `compute` that implements the solving strategy. For any new function, the user should provide implementations for the following methods: `basic_case`, `combine`, `create_left_function`, and `create_right_function`.

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

For the shared memory case, the framework's efficiency and usability are emphasized in detail in [11].

In order to offer a better scalability, another execution type is proposed: a distributed execution based on MPI. This improved scalability could be necessary in various situations. The framework is oriented on applications that use regular data sets of large sizes, and so, it should be possible to use for execution multiple computational nodes.

IV. MPI Based Extension

For the shared memory variant, the cost of splitting and combining powerlist data structure is reduced to minimum by eliminating the necessity of copying the elements of the lists; only the characteristics of the list are changed $(start, end, incr)$.

For an MPI variant we have to work with distributed memory, which is accessed from different processes, and so,

we had to treat the cost associated to splitting and combining the list very carefully. It is well known that the associated costs for transmitting data are much higher than the simple computation costs. Because of this we have tried to eliminate the transmission cost where it was possible.

In order to achieve this, we had to analyze the powerlist function definition from the data point of view. Powerlist functions are defined on list data structures and each time we apply the recursive function definition, the data is split into two new data structures. If we assume that the computation associated to one of these structures will be computed on another process, this splitting step implies also communication operations. Similarly, the combining stage could apply operations on the corresponding results of the two recursive calls, which also implies communication.

The analysis of powerlist function execution leads to the following three phases:

1) *Descending/splitting phase* that considers the operations needed to split the list arguments, and additional operations, if they exist.
2) *Leaf phase* that considers the operations executed on singletons.
3) *Ascending/combining phase* that considers the operations needed to combine the list arguments, and additional operations, if they exist.

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

There are in fact very few cases when the input is transformed at the descending phase. Such cases involve some additional computation on the sublists obtained at each step. Also, there are many cases when function transformations could be applied – such as tupling – in order to eliminate these additional computations [12].

From this analysis, we may identify different classes of powerlist functions:

1) $splitting \equiv data\_distribution$
   functions for which the splitting phase implies only data distribution – examples: $map, reduce, fft$;
2) $splitting \not\equiv data\_distribution$
   functions for which the splitting phase involves also additional computation besides the data distribution – examples: $f(p \natural q) = f(p+q) \natural f(p-q)$;
3) $combining \equiv data\_composition$
   functions for which the combining phase implies only the data composition based on construction operator (tie, or zip) of the results obtained in the leaves – examples: map;
4) $combining \not\equiv data\_composition$
   functions for which the combining phase involves specific computation used in order to obtained the final result – examples: $reduce, fft$.

As it can be noticed, these classes are not disjunctive, and so, instead of defining different types of functions, we split the function execution into three sections and we use *template method* pattern [7] in order to let these parts vary independently. Adding the corresponding case is done using the *decorator* pattern [7].

For MPI execution, we need to specify different computational tasks that should be executed for each phase. This could be done by adding specific wrappings such as: `MPI_PowerCT_split`, `MPI_PowerCT_compose`, `MPI_PowerResultCT_compose`, `MPI_PowerCT_read`, `MPI_PowerCT_write`.

Since for domain decomposition parallel applications, the input/output data sizes are very large, usually these data are stored into files. This introduces another variation in defining the kind of partial operations that one function should define. These variations are given by the way the input data are taken, or/and how the result is given, in combination with the previously described cases.

If the data are taken from a file, then:

- the case 1) implies concurrent file reads of the appropriate data done by each process;
- the case 2) implies a reading done by the process 0, followed by an implementation of the decomposition phase based on MPI communications.

The concurrent read for each input parameter of powerlist type is possible since each process reads different data, and these data depends on: 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.

Decomposition based on tie operator is very simple and direct – each process receives a `filepointer` from where it starts reading the same number of data elements.

For zip decomposition, the file reading involves a little bit more complex operations: each process also receives a starting `filepointer` and a number of data elements that should be read, but each new reading needs also a seek operation which is based on a bit reverse operation applied on the indices of the data elements.

A similar situation is encountered for the combining phase together with the results writing. Concurrent writing is possible if the output file is already created and each process will write values on different positions that are computed based on the operator type, element data size, and the process rank.

Since all the framework's classes are generic and also almost all MPI Java implementations need simple data types to be used in communication operations, we have used byte array transformation of the data through serialization. This implies specific operations that serialize/deserialize specific data types. This also helps the reading/writing mechanism that is defined at the byte level.

The MPI execution of a powerlist function is very simple: in order to have an MPI execution of an powerlist function, it is not necessary to define specific MPI functions for each particular function, but just to specify if the function needs a `split` operation or a simple `read` operation, and a `compose` operation or a `write` operation, etc.

The next code snippet shows the construction of the MPI computational tasks for `map`, when the input data are read from a file, and the result is also written into a file.

```
sizes[0] = n;
elem_sizes[0] = elem_size;
files[0] = "matrix.in";
I_MPI_CTOperations<Matrix> exec =
   new MPI_PowerResultCT_write<Matrix>(
    new MPI_PowerCT_read<Matrix>(
     new MPI_PowerResultFunctionCT<Matrix>(
      new Map<Matrix>(f, rlist.toTiePowerList(),
                  list.toTiePowerList()), files, sizes,
                  elem_sizes), "matrix.out");
```

The operations `read`, `write`, `compose`, etc. are generic because they are based on the template method operations defined for each powerlist function. Also, they are dependent on the total number of processes and the rank of each process.

V. Applications and Experiments

In order to evaluate the usability of the MPI extension, we consider three classical problems – *Map, Reduce*, and *Fast Fourier Transform* – for which we evaluate the performance of their implementations developed using the presented MPI extension. We will also discuss the performance implications of the execution parameters by conducting some experiments for these problems, considering different input sizes and different computing solutions.

All the experiments presented in this paper have been executed on a IBM NextScale cluster with a connectivity of 56 Gb/s (Infiniband Mellanox FDR switch SX6512 with 216 ports, 1:1 subscription rate). Each node is equipped with two Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz, and each of these two processors has 10 physical computing cores and 128 GB RAM memory. The machine was running Red Hat Enterprise Linux Server release 7.0 (Maipo) on 64bit and Java 8. The MPI library used was Intel(R) MPI Library for Linux* OS, Version 2018 Update 1 which contains a Java MPI binding. We have used 4 nodes, and all the experiments have been repeated three times, average time being considered.

In general we considered three cases for the evaluation: sequential execution, multithreaded execution based on the usage of the executor constructed over *ForkJoinPool* Java executor, and MPI execution combined with multithreaded execution. For MPI execution we have considered different variants based on the total number of MPI processes ($proc$), and based on the spreading of these processes on the considered physical nodes of the cluster (ppn = the numbers of processes per cluster-node). Each MPI process executes the leaf computation using a multithreaded executor (based on *ForkJoinPool*).

In the performance graphics presented in this section, we have considered the data size expressed as powers of two, so only the corresponding powers are represented, and the time is expressed in nanoseconds. Visibility reasons determined different maximum sizes represented in the graphics.

A. Map

The representation of the powerlist *map* computation is given in Section II. The definition of function map could be done either using *tie* or *zip* operator (in the experiments the *tie* variant was used).

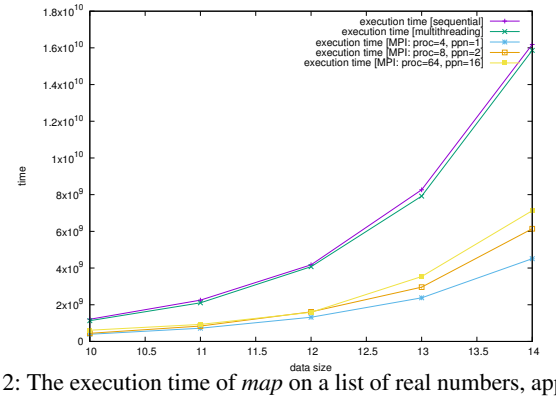

Fig. 2: The execution time of *map* on a list of real numbers, applying square – when the input file reading and output file writing times are included.

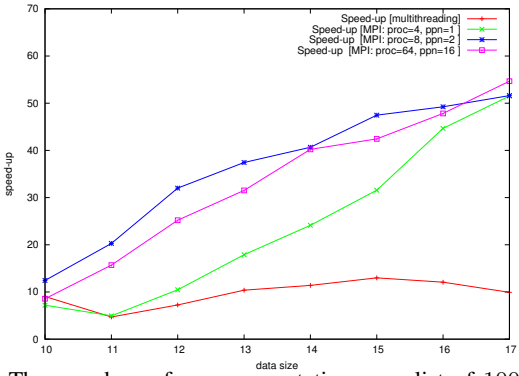

Fig. 3: The speed-up of *map* computation on a list of 100×100 matrices, applying square – when the input file reading and output file writing times are not included.

Since the result of a map computation is a powerlist, the corresponding class `Map<T>` extends the class `PowerResultFunction<T>`. Some details about the MPI definition for this class were given in the previous section.

The comparative analysis considers the sequential execution which is based on the type `BasicList` that implies an iterative application on the function argument of `map` on each list element, instead of the sequential execution of the *powerlist* variant that involves recursion which could increase the computation time.

For `map`, both input reading and the result writing could be done in parallel by each process, as described in the previous section. This leads to a very good performance even when `map` is tested for simple lists of real numbers with square operation – Figure 2. It can be noticed that, in this case, the multithreaded variant is not better than the sequential one; this is due to the fact that the improvement obtained through the parallelization is covered by the added time due to the involved recursion, and task creation.

Still, we wanted to evaluate also the performance brought by MPI for the computation phase only. For this purpose, some experiments with matrices of different orders have been conducted. Here we presents only the results for matrices

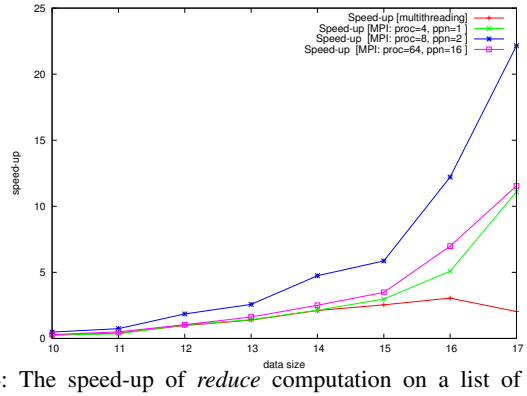

Fig. 4: The speed-up of *reduce* computation on a list of 100×100 matrices, applying addition – when the file reading time is not included.

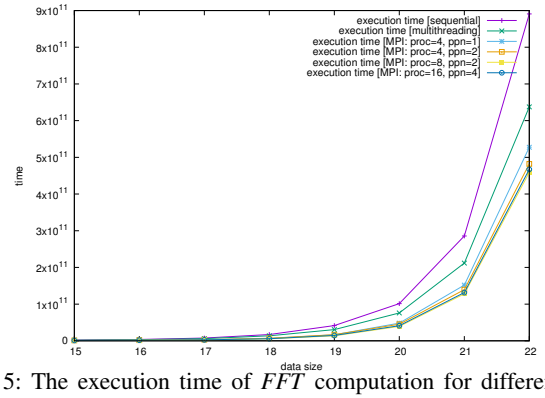

Fig. 5: The execution time of *FFT* computation for different data sizes when the reading time of the input data is included.

of order 100×100, but it should be mentioned that the peformance is improved as the order of matrices and the data size increase.

B. Reduce

As for `map`, the powerlist representation of the `reduce` computation is given in Section II.

Corresponding to this definition, since *reduce* returns a single value and not a powerlist, a class `Reduce<T>` that extends the class `PowerFunction<T>` is used.

For `reduce` we have done several experiments, and we present here the performance obtained for adding matrices of order 100 considering the analysis that does not include the time for reading the input data – Figure 4. The speed-up increases with the data size, and the performance obtained by using MPI variant is much better than the one that uses only multithreading.

C. Fast Fourier Transform

For a polynomial p with complex coefficients, Fourier Transform could be obtained by evaluating p on a specific sequence of points: $(W\ p)$. If the polynomial is given as a powerlist of its coefficients, the points where the values should be computed form also a powerlist $(W\ p) = (\omega^0, \omega^1, \ldots, \omega^{n-1})$, where n is the length of p and w is the nth principal root of1.

Since $(W\ p)$ contains powers of the nth principal root of 1, and since they have special relations with the roots of 1 of lower order, the Fourier Transform can be recursively computed in $O(n \log n)$ steps, using the well known Fast Fourier Transform algorithm [6]. The powerlist representation of this algorithm, proved in [10], is:

$$\begin{cases} \mathit{fft}([a]) & = & [a] \\ \mathit{fft}(p \natural q) & = & (P + u \times Q) \,|\, (P - u \times Q) \end{cases} \quad (5)$$

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

The result of the function $\mathit{powers}(p)$ is the powerlist $(w^0, w^1, .., w^{n-1})$ where n is the length of p and w is the $(2 \times n)$th principal root of 1.

The operators used in the *fft* definition are extension of the addition, substraction, and multiplication operators on powerlists. They have simple definitions that consider as an input two similar powerlists, and specify that the elements on the similar positions are combined using the corresponding scalar operator. The theoretical parallel time-complexity of *fft* computation using this powerlist definition is $O(\log n)$ parallel steps using $O(n)$ processors.

In order to have Fast Fourier Transform also for the sequential case, the sequential implementation that has been tested it is also based on `PowerLists`, but without using any parallel executor. A sequential variant based on `BasicList` does not correspond to Fast Fourier Transform.

For FFT, we need to apply `compose` operation between the intermediary results of each process; in this way the final result – even if it is a powerlist – is obtained in one process, and so the time for writing the result cannot be parallelized as for the *map* case.

Figure 5 shows the obtained performance when the input file reading time is included into the total execution time. For *fft* the impact of including the file reading into the performance analysis is not so predominant especially because the reading is a *zip* type reading that implies a skip operation before each reading, and this diminishes the performance (not contiguous block reading).

The experiments described here prove that the MPI extension of the framework brings important improvements over the multithreaded variant.

From all the experiments, we have noticed that the variant with a number of MPI processes equal to the physical processors number, and these processes evenly distributed on the nodes was the best choice for almost all the cases.

We have done also experiments with more processes distributed on one node (so for pairs $[proc, ppn]$ equal to - $[8, 4], [16, 8], \ldots$ etc.) but they did not proved to be better. In these cases the number of threads used by the `ForkJoinExecutor` was set smaller than the implicit value given by `ForkJoinExecutor.commonPool()` that depends on the number of cores of the processor. These results are also theoretically confirmed by the fact that in a shared memory context it is more efficient to use threads, instead of MPI processes.

VI. Related work

An important approach in defining high level parallel models is based on algorithmic skeletons [4]; they have been

used for the development of various systems providing the application programmer with suitable abstractions. Powerlists and their theory also offer a skeleton based approach in construction of parallel programming models.

The powerlist theory has been used in other works that try to facilitate the definition of formal and efficient parallel programs. For example, in [1] transformation rules to parallelize divide-and-conquer (DC) algorithms over powerlists are presented. The goal of this work was to derive programs for the *massively data parallel model*. In [3] powerlists are used to capture both parallelism and recursion succinctly, and automatically schedule partitioned matrices over a GPU cluster.

Lithium [2] is implemented as a Java package and represents both the first skeleton based programming environment in Java, and the first complete skeleton based Java environment exploiting macro-data flow implementation techniques. *Calcium* [5] and *Skandium* [9] are two others Java skeleton frameworks.

Java 8 Streams are playing an important role in bringing functional programming to Java, and they are also based on algorithmic skeletons. In [11] a comparison between the performance of some algorithms' implementations in Java parallel streams and in the presented analyzed framework is done.

Also there are studies that emphasize that Java could also be a good candidate for MPI implementation. There are several reasonably good Java implementation of MPI: Java Intel MPI [14], Java OpenMPI [13], or MPJ Express [8]. Each of these could be used for the presented scaling up, even if there are few syntactic differences between these Java MPI implementations. We have also used MPJ Express in few experiments, and the results have been similar to those obtained based on Intel Java MPI. The expected further improvements of these Java MPI bindings will implicitly lead to improvements of the presented framework.

VII. Conclusions and Future Work

We have presented an MPI extension for distributed memory system execution of a Java parallel programming framework based on powerlists.

The framework design is based on design patterns that provide easy definition of the new concrete programs, but also the possibility to extend the framework to accept other similar data structures (such as *ParList* and *PList*), and also other execution models – these will be the focus of the associated further work.

It was important to identify and to analyze the computation phases of a powerlist function: *descend, leaf*, and *ascend*, for which correspondent generic classes have been defined. These allow efficient MPI execution of the powerlist functions by wrapping their computation with specific operations depending on the functions' characteristics: `read`, `split`, `compose`, etc.

The set of analyzed examples includes: *map, reduce*, and *Fast Fourier Transform*, and the experiments shows that the obtained performance inside the *JPLF* framework is very much improved using the MPI execution.

MPI based execution could be combined with multithreading computation, and the right way of choosing the parameters (MPI number of processes, number of process on each node, number of threads used by each process) depends on the architecture specific characteristic. More concretely, the experiments emphasize that the best choice is to consider a number of MPI processes equal to the double of the number of physical nodes and these processes to be evenly distributed over the nodes.

From all the examples, the comparison between the parallel execution based only on multithreading and the one which also involves distributed parallel computation based on MPI, emphasizes that, if the corresponding architecture is available, we can gain important advantages by using MPI.

References

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

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

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

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

[5] D. Caromel and M. Leyton, "Fine Tuning Algorithmic Skeletons" Euro-ParParallel Processing, 13th International Euro-Par Conference, Rennes, France, pp. 28-31, 2007.

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

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

[8] A. Javed, B. Qamar, M. Jameel, A. Shafi and B. Carpenter. "Towards Scalable Java HPC with Hybrid and Native Communication Devices in MPJ Express," *International Journal of Parallel Programming (IJPP)* 2015 - Springer.

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

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

[11] V. Niculescu, F. Loulergue, D. Bufnea, and A. Sterca. "A Java Framework for High Level Parallel Programming using Powerlists," in *Parallel and Distributed Computing, Applications and Technologies* (PDCAT). IEEE, Taipei, Taiwan. 2017, pp. 255-262.

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

[13] O. Vega-Gisbert, J.E. Roman, J.M. Squyres. "Design and implementation of Java bindings in Open MPI," *Parallel Computing* 59, pp. 1-20 (2016).

[14] "Intel MPI Library Developer Reference for Linux OS: Java Bindings for MPI-2 Routines," [Online] https://software.intel.com/en-us/mpi-developer-reference-linux-java-bindings-for-mpi-2-routines, accessed: 2018-05-10.