Multi-way Divide and Conquer Parallel Programming based on PLists

Virginia Niculescu
Computer Science Department
Babeş-Bolyai University
Cluj-Napoca, Romania
vniculescu@cs.ubbcluj.ro

Darius Bufnea
Computer Science Department
Babeş-Bolyai University
Cluj-Napoca, Romania
bufny@cs.ubbcluj.ro

Adrian Sterca
Computer Science Department
Babeş-Bolyai University
Cluj-Napoca, Romania
forest@cs.ubbcluj.ro

Robert Silimon
Frequentis Romania
Cluj-Napoca, Romania
Ioan-Robert.SILIMON@frequentis.com

***Abstract*—Divide and Conquer with all its variants represents an important paradigm of parallel programming. In this paper we present an implementation of *PLists* data structures and functions, which is introduced as an extension of a Java parallel programming framework – JPLF. The JPLF framework was initially based on *PowerLists* and their associated theory. By using functions defined on *PLists*, we may easily define programs based on the multi-way Divide and Conquer paradigm. Also, their definition allows the description of any kind of embarrassingly parallel computation. By introducing *PLists* into the JPLF framework, its application domain is very much enlarged, and also the flexibility of choosing the best computation variants is increased. The sizes of the data lists are not constrained any more - as it is for *PowerLists* to a power of two – and the level of parallelism could be much easier controlled. The experiments done for several applications reveal important improvements of the obtained performance.**

***Index Terms*—parallel computation, divide&conquer, recursive data structures, performance, framework**

I. Introduction

Parallelism is now everywhere, in small handheld devices like smartphones, in regular consumer notebooks and in high performance computing clusters. But parallel programming still remains a difficult and error-prone job, and writing correct parallel programs from scratch is often a difficult goal. Powerful conceptual frameworks that could offer also flexibility are required in order to ease the development of parallel programs.

Because of its strong multithreading support, synchronization mechanisms and thread-safe data structures, Java represents an appealing programming language for writing parallel programs.

PowerLists and *PLists* introduced by J. Misra [19] and J. Kornerup [16] are data structures naturally built for exploiting the power of Divide & Conquer (DC) programming paradigm. They allow working at a high level of abstraction, especially because the index notations are not used. Their advantage over regular lists is that they provide two different views over the underlying data, simplifying the design of algorithms working on them. In order to support correctness verification of parallel programs, algebras and induction principles are defined on these special data structures.

In this paper we present an extension based on *PLists* of a Java parallel programming framework JPLF [20], which has been initially created to support only *PowerLists*. *PLists* bring the advantages of allowing definitions of multi-way divide&conquer programs, but also (when the arity list is formed by only one number) definitions of embarrassingly parallel programs. Together: *PowerList, ParList, PList* with their multidimensional counterparts could be used as a foundation for a general parallel programming model based on domain decomposition [21]; this analysis was leaded by the general characteristics that a model of parallel computation should have [22].

The JPLF framework was implemented following object-oriented design principles and based on design patterns [13] in order to be flexible and extensible. The shared memory execution environment is based on thread pools (the tested implementation uses the `ForkJoinPool` executor, but others could be used too) where the size of these pools depends on the system where the execution takes place.

This paper is organized as follows. In section III we give a general description of *PLists*, and a description of the JPLF design and *PList* implementation is given in section IV. Section V presents some use-cases and the practical experiments related to them. Related work is presented in section II, the conclusions together with future work being presented in the last section (sec. VI).

II. Related Works

Divide&Conquer represents one of the most important algorithmic skeletons. Algorithmic skeletons are considered the foundation of an important approach in defining high level parallel models [6]; they have been used for the development of various systems providing the application programmer with suitable abstractions, but also reliability.

Different approaches have been considered to facilitate general and easy usage of Divide&Conquer pattern in parallel context [11], [14], [15]. For most of the problems that could be solved using Divide and Conquer pattern, the tasks(subproblems) creations is leaded by the domain decomposition; *PowerLists* and *PLists* express very well this model.

There have been previous works that try to facilitate the definition of formal and efficient parallel programs based on the *PowerList* theory.
A representative example is the work presented in [1] where

transformation rules over *PowerLists* functions are introduced, in order to adapt the *PowerLists* programs for the *massively data parallel model*.
A functional implementation of *PowerList* functions has been done in BSML (Bulk Synchronous Parallel ML) [18].
Also *PowerLists* have also been used to capture parallelism and recursion succinctly for GPU computing [3].

Lately, there has been registered an increase in the use of Java in High Performance Computing area [23], and Java has been used as a support language for defining structured parallel programming environments based on skeletons, too. Some representative examples are those described in: *Lithium* [2], *Calcium* [8] and *Skandium* [17].

An important role in enabling functional programming in Java is played by Java Streams, which are also based on algorithmic skeletons. In [20] a more detailed comparison between the performance of selected algorithms' implementations using Java parallel streams and the JPLF *Powerlist* implementation is done. In comparison with them, the JPLF framework has the asset of contributing with additional support for applications that need more complicated data decomposition as that represented by the zip operator (e.g. Fast Fourier Transform).

In addition, with the *PLists* extension, the performance is improved while the domain of the applications that could be defined inside the framework is enlarged.

III. PLIST DATA STRUCTURES

The *PList* data structure was introduced in order to develop programs for the recursive problems which can be divided into any number of subproblems, numbers that could be different from one level to another [16]. It is a generalization of the *PowerList* data structure and it has three constructors for creating *Plists*: one that creates singletons from simple elements, one based on concatenation of two lists, and the other based on alternative combining of two 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, square brackets are used to denote ordered quantification. The expression

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

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

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

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

Formally, the *PList* constructors have the following types:

$$\begin{array}{lcl} <.> & : & X \to PList.X.1 \\ [\ |\ i : i \in \overline{n} : .] & : & (PList.X.n)^n \to PList.X.(n*m) \\ [\ \natural\ i : i \in \overline{n} : .] & : & (PList.X.n)^n \to PList.X.(n*m) \end{array} \quad (1)$$

where m is the length of the arguments, which are n similar *Plist*.

The *PList* axioms also define the existence of the unique decomposition of *PList* using constructors operators [16].

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 is more than one *PList* argument they all must have the same length). Functions over *PList* are only defined for certain pairs of these input values; to express the valid pairs, it is required that the specification of the function defines the predicate:

$$defined : ((PosList \times PList) \to X) \times PosList \times PList \to Bool \quad (2)$$

to characterize where the function is defined.

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 we find it convenient.

We illustrate functions' definitions with two examples: *reduction* and integration through *repeated rectangle formula*.

Reduction

This function computes the reduction of all elements of a *PList* using an associative binary operator $\oplus$:

$$\begin{array}{lcl} defined.red(\oplus).l.p & \equiv & prod.l = length.p \\ red(\oplus).[].<a> & = & a \\ red(\oplus).(x \triangleright l).[|i : i \in x : p.i] & = & (\oplus i : 0 \le i < x : red(\oplus).l.(p.i)) \end{array} \quad (3)$$

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

The addition of numbers is the most popular example of reduction; we denote $sum = red(+)$.

Map

Map function applies on each element of a *PList* a unary function f:

$$\begin{array}{lcl} defined.map(f).l.p & \equiv & prod.l = length.p \\ map(f).[].<a> & = & a \\ map(f).(x \triangleright l).[|i : i \in x : p.i] & = & (f(i) : 0 \le i < x : map(f).l.(p.i)) \end{array} \quad (4)$$

where $prod.l$, $length.p$, $[]$, and $\triangleright$ have the same meaning as for the reduce function. Similar to $reduce$, this function could also be defined using $\natural$ operator.

Numerical Integration with the Rectangle Formula

For a function $f : [a,b] \to \mathbb{R}$, the integral $I = \int_a^b f.xdx$ can be approximate by the following recursion [10]:

$$\begin{array}{l} Q_{D_0}.f = (b-a)f((a+b)/2) \\ Q_{D_k}.f = \frac{1}{3}Q_{D_{k-1}}.f + h\sum_{i=1}^{2m} f.x_i, \forall k > 0 \end{array} \quad (5)$$

where $h = \frac{b-a}{3^k}$, $m = 3^{k-1}$, and the x_i values are computed by the following formulas:

$$\begin{cases} x_1 & = & a + \frac{h}{2} \\ x_2 & = & a + \frac{5}{2}h \\ x_{2j+1} & = & x_1 + 2jh \\ x_{2j+2} & = & x_2 + 2jh,\ 1 \le j < 3^{k-1}. \end{cases} \quad (6)$$

The formula considers at each step a division into 3 equal parts, and the values of the function in three points of each interval.

We will define a *PList* function *drept*, that computes $(Q_{D_k}.f)$, for a given k.

If we consider a division on interval $[a, b]$ with $n = 3^k$ points:

$$\begin{array}{lcl} [x_0, \ldots, x_{n-1}] & = & [a_0, a_0 + \frac{h}{3^k}, \ldots, a_0 + \frac{3^k-1}{3^k}h], \\ \text{where } a_0 = a + \frac{h}{2}. & & \end{array} \tag{7}$$

It can be noticed that at the combine stage 3^{k-1} points are used for computation of $(Q_{D_{k-1}}.f)$ and $2 * 3^{k-1}$ intervene in computation of the second term of the sum that computes $(Q_{D_k}.f)$.

The function

$$drept : Real \times PosList \times PList.Real.n \to Real \tag{8}$$

defined by:

$$\begin{array}{l} defined.sum.l.p \equiv prod.l = length.p \\ drept.[].<x> = hk * x \\ drept.hk.(3 \triangleright l).[\natural i : i \in \overline{3} : p.i] = \\ \qquad \frac{1}{3} * drept.(3 * hk).l.(p.1) + hk * sum.(2 \triangleright l).(p.0 \natural p.2), \end{array} \tag{9}$$

has three arguments; the first $hk = \frac{b-a}{3^k}$ is the division step, the second is a list form by k values equal to 3, and the third is the *PList* that contains the function values in the specified points.

IV. *Plist* IMPLEMENTATION IN JPLF FRAMEWORK

The JPLF framework provides general support in Java for computing *PowerList* functions and starting from now also *PList* functions.

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

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

This separation of concerns allows us to modify them independently, offering the possibility of extension by providing new or improved ways for execution, for storage, or allowing other data structures to be included.

`IBasicList` is a type used for working with simple basic lists and it is also used as a unitary supertype of the specific types. They are also used for defining sequential nonrecursive functions, which will be specializations of `BasicListFunction` or `BasicListResultFunction`. They facilitate the definition of functions on lists that are based on iterations.

A. PList Implementations

When a *PList* is decomposed, the result is formed of a set of similar sublists. In order to avoid element copy, the storage of all sublists remains the same as that of the initial list, and only the *storage information* is updated. For each list `l`, the *storage information* `SI(l)` is composed of:

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

From a given list with storage information `SI(list)` being `{base, start, end, incr}`, the *tie* and *zip* deconstruction operators create a number of lists that have the same storage container – `base` and correspondent updated values for `(start, end, incr)`. For example if we split a *PList* into 3 sublists (provided that its length is divisible by 3), these are characterized by the following storage information:

Op.	Sublist	SI
tie	left	base,start, (start+end)/3,incr
	middle	base,(start+end)/3,2/3(start+end),incr
	right	base,2/3(start+end),end,incr
zip	left	base,start, end-2*incr,incr*3
	middle	base,start+incr,end-incr,incr*3
	right	base,start+2*incr,end,incr*3

As for *PowerLists*, there are two specializations of the `PList` type: `TiePList` and `ZipPList`. The operator type used for splitting a *PList* is determined by the specific type of that *PList* which could be either `TiePList` or `ZipPList`, and this enables polymorphic definitions of the splitting and combining operations.

B. PList Functions

A *PList* function expresses the specific computation by using *tie* or *zip* deconstruction operators for splitting the *PList* arguments, and its definition is directed by the two specific cases – the base case (for singletons) and the inductive case (for non-singleton lists). The correctness of the functions is proved using the associated structural induction principle.

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

We consider functions for which a certain *PList* argument is always split by using the same operator (and so it preserves its type – a `TiePowerList` or a `ZipPowerList`). Also, if the result is a *PList*, this is constructed at each step by using the same operator. Based on this assumption, in the framework, the construction and deconstruction operators are not explicitly specified for each function; instead they are implied by the *PList* types – if they are `TiePList`s, the *tie* operator is used, and if the type is `ZipPList`s then the *zip* operator is used. So, it is very important when a specific function is called, to prepare it in such a way that the types of the arguments are the types implied by the specific op_args sequence. The `PList` class provides two methods `toTiePList` and `toZipPList` that transforms a general `PList` into a specific one which has specific implementation for splitting and construction.

The result of a *PList* function could be a simple object or a *PList* data structure. The differentiation between these two cases is done by considering the following two types: `PFunction` (functions that return simple objects) and `PResultFunction`(functions that return *PLists*).

The `PFunction` class defines the template method `compute` that implements the divide&conquer solving strategy. The following code snippet (Code 1) shows the code of the template method `compute` defined for `PFunction`:

```
public Object compute() {
 if (test_basic_case()) {
  this.result = basic_case();
 }
 else {
  split_arg();
  List<PFunction<T>> sublists_functions =
     create_sublists_function();
  List<Object> res_sublist =
     new ArrayList<Object>();
  for (int i=0; i<sublists_functions.size();i++)
  {
   res_sublist.add(
     sublists_functions.get(i).compute());
  }
 this.result = combine(res_sublist);
 return this.result;
  }
```

Code 1: The code of the template method `compute` of the class `PFunction`.

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

- `basic_case`,
- `combine`,
- `create_sublists_function()`.

Still, it is not mandatory to provide implementations for all of them, their implicit definitions could be used. For example, for *map* (the function that applies an atomic function on each element of the list) we have to give a definition only for `basic_case()`, while for *reduce* we have to provide an implementation only for `combine()`.

Using this design, new *PList* functions could be defined by extending the `PFunction` or `PResultFunction` classes.

C. Multithreading Executors

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

The parallel execution is based on executors, and this allows modifications or specializations.

For *PList* specialized executor classes are created – `FJ_PFunctionExecutor` and `FJ_PFunctionComputationTask`

The class `FJ_PowerFunctionExecutor` provides now an implementation based on the `ForkJoinPool` Java executor (others could be considered too).

The implementation of the `compute` method of the `FJ_PFunctionComputationTask` class relies on the fact that the *PLists* functions are defined based on the *Template Method* pattern [13]. Its implementation follows the same skeleton as that used by the `compute` method defined for any *PList* function.

A special attribute `recursion_depth` is used by `FJ_PFunctionComputationTask` to control the creation of the parallel tasks – at each level after new tasks are forked to be executed in parallel, this parameter is decreased and when it is equal to 0 sequential computation is called (the `compute` method of the function).

D. List Transformer

The parallelism could be also bounded by transforming the argument lists into lists of sublists. If the sublists are `BasicLists` then for them sequential computation is done.

How the sublists are considered depends on which of the two operators, `tie` and `zip`, is applied. The transformation preserves the same storage of the elements, and only lists information is changed.

If *tie* is the operator used to transform a `PList` of n elements into a `PList` of p `BasicLists` then it is not mandatory to have $p|n$, but for *zip* this condition is required.

V. Applications and Experiments

In order to evaluate the usability of the *PLists* implementation we consider the applications – *Reduce*, *Map* and *Repeated Rectangle Formula* – for which we evaluate the performance.

In general we have considered three cases for the evaluation:

1) sequential execution;
2) unbounded parallel execution – multithreading execution for which parallel tasks are created until the base cases are attained;
3) bounded parallel execution – multithreading execution for which the number of parallel tasks is bounded through one of the following two mechanisms:
   a) the initial list is transformed into a list of `BasicLists`
   b) the parallel recursion depth is set to a lower value than the maximal recursion depth.

A. Reduce

The PowerList representation of the reduce computation is given in Sec. III. The definition of function red could be done either using *tie* or *zip* operator.

The `Reduce` class overrides the method `combine` that applies the associative operator on the results of the recursive calls on the sublists. The method `basic_case()` is overridden just to include also the case when the argument is a list of sublists, in which case the base case uses a sequential `Reduce` function on `BasicLists`.

For *Reduce* we conducted two experiments:

- *PLists* of random 10×10 matrices of real numbers, the length of the *PLists* are multiples of 5000 (Fig. 1);
- *PLists* of random 100×100 matrices of real numbers, the length of the *PLists* are powers of 2 (Fig. 2).

The figures emphasize the obtained speedups, which are computed as: $speedup = T_{sequential}/T_{parallel}$, where $T_{sequential}$ is the execution time of the sequential computation, and $T_{parallel}$ is the execution time of parallel computation.

Since for matrix addition the sequential computation is more efficient if an iterative (non-recursive) variant is considered, the bounded parallelism in this case was based on transforming the initial list of matrices into a `Plist` of `BasicLists` of matrices.

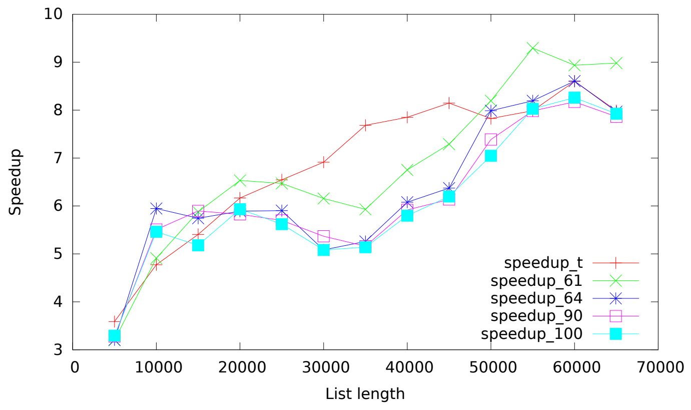

Fig. 1. *Reduce* - matrix addition: 10×10 matrices. *speedup_t* corresponds to unbounded parallelism variant, *speedup_n* correspond to bounded parallelism variant with PList with n elements of type BasicList.

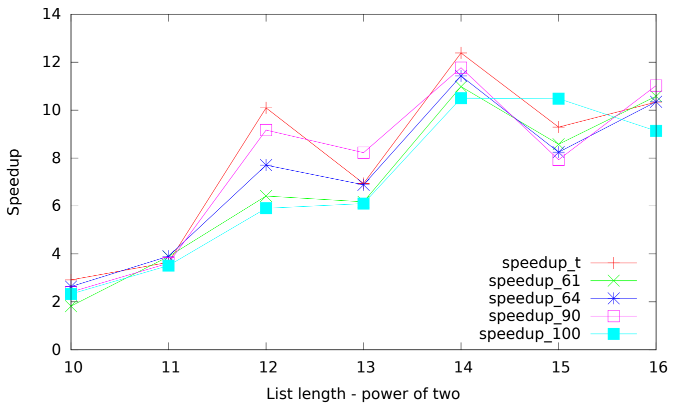

Fig. 3. *Map* – applying squaring on each element of a list of 100×100 matrices. *speedup_t* corresponds to unbounded parallelism variant, *speedup_n* correspond to bounded parallelism variant with PList with n elements of type BasicList.

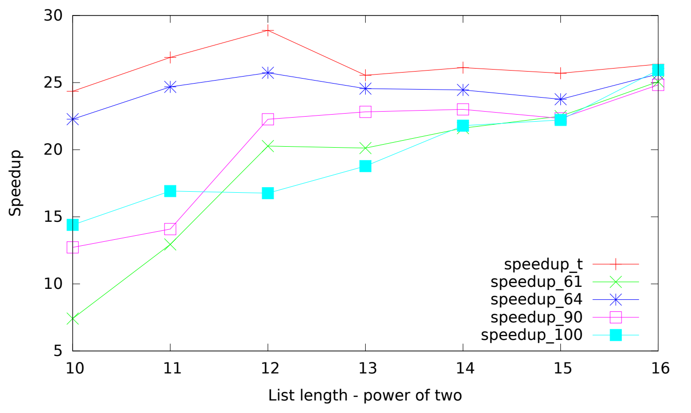

Fig. 2. *Reduce* - matrix addition: 100×100 matrices. *speedup_t* corresponds to unbounded parallelism variant, *speedup_n* correspond to bounded parallelism variant with PList with n elements of type BasicList.

For bounded parallelism, the best choice for the number of elements of type `BasicList` depends on:

- the initial list length,
- the possibility to obtain balanced length sublists,
- the decomposition into prime factors of the length of resulted `Plist` – the resulted *arity list*;
- the correlation between the maximal number of parallel recursive tasks and the number of the hardware cores.

For example:

- if the number of `BasicList`s inside the `PList` argument is equal to 61 the arity list is equal to $[61^1]$, and so 61 parallel tasks are split from the first level. Each task will compute sequentially the corresponding sum.
- if the number of `BasicList`s inside the `PList` argument is equal to 64 the arity list is equal to $[2^6]$, and then the `PList` will be split as a `PowerList`.
- if the number of `BasicList`s inside the `PList` argument is equal to 100 the arity list is equal to $[2^2, 5^2]$, and then there will be 2 levels that do the splitting into two equal size lists, and other two levels with a splitting operations into five sublists.

B. Map

Map emphasizes simple parallel computation, and the correspondent $PList$ function has been defined in Sec. III. The example considers matrices of size 100×100 for which we apply square operation (power of two) for each element. Fig. 3 emphasizes the results obtained for the executions with unbounded parallelism and with different levels of bounded parallelism – the initial lists being transformed into a `PList` with different numbers of `BasicList` elements.

By analysing the reduce and map examples, we can notice that for large data sets, or if the `basic_case` and/or the `combine` functions are computational intensive the difference between bounded and unbounded parallelism variants are not significant.

C. Repeated Rectangle Formula

As we have seen in Sec. III we have a simple *PList* function definition that approximates an integral using the repeated rectangle formula (eq. 8-9). This example emphasizes a multi-way divide&conquer program where the division has to be done always in 3 parts (subproblems). For this case the `basic_case` and `combine` functions are not computational intensive (this is important because in parallel cases we have to consider the overhead time of task creation, that we try to keep it lower than elementary operations).

The results of the experiments done for this example are illustrated in the Fig. 4.

The variant that considers bounded parallelism is based on the limitation of the recursion depth. For the presented test the `recursion_ level` has been set to 4 levels.

We may notice that for large sets of data the bounded parallelism variant becomes better since the overhead due to the task creation is limited.

Remark. The experiments have been performed on an IBM x3750 M4 machine, running CentOS 7, 64 bit kernel, Java

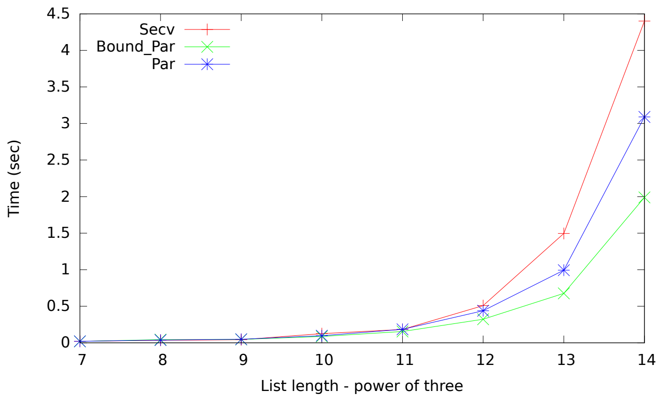

Fig. 4. *Repeated Rectangle Formula*: execution time – Sequential execution vs Parallel execution vs. Bounded Parallel execution

8 and equipped with 4 Intel Xeon E5-4610 v2 @ 2.30GHz CPUs (8 cores per CPU) and 64 gigabytes of RAM. Each test has been repeated 5 times, average execution time being considered.

VI. Conclusions

The *PList* notation is very rich – it includes the *PowerList* theory as a special case. While this generality is not always needed in order to describe parallel computations, it may be useful when the problem is stated in a different radix than 2 (e.g. repeated rectangle formula III), or in a mixed radix as it is for example the case of Fast Fourier Transform with arbitrary factors ($N = r_1, r_2 ... r_m$) [5]. Together with other examples this will be the focus of the associated further work.

The existence of the two different constructor operators differentiates *PList* data structures from other list data structures which are based on simple concatenation. In addition, the possibility to split at each step in a different number of sublists (and so subproblems) introduces an important level of flexibility that is useful also in order to choose the most performant partition of the problem. In extremis, the arity list could be considered as being formed of only one number equal to the size of list. This introduces the possibility to define any computation that fits into the "embarrassingly parallel" paradigm.

The ability to control the parallel recursion level and so to control the number of tasks that are going to be executed in parallel increases the ability to improve the practical performance.

An important advantage is brought by the possibility to work with lists of lists that allows us to combine the paradigms - e.g. a *PList* (or a *PowerList*) of *BasicLists* elements allows PAR-SEQ computation. Vice-versa is also possible - SEQ-PAR computation if *BasicLists* of *PLists* or *PowerLists* are used. The types of the lists imply the types of the combination and of execution. The combination could be done on multiple levels, and so the possibility to express different types of computation increases.

References

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

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

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

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

[5] I E. Oran Brigham. "The fast Fourier transform and its applications," Prentice-Hall, 1998.

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

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

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

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

[10] Gh. Coman, *Numerical Analysis*, Editura Libris, Cluj-Napoca, 1995 (in Romanian).

[11] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. "A Divide-and-Conquer Parallel Pattern Implementation for Multicores," In The Third International Workshop on Software Engineering for Parallel Systems (SEPS 2016) co-located with SPLASH 2016. Amsterdam, 2016, ACM, pp. 10-19.

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

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[14] C. H. Gonzalez and B. B. Fraguela. "A generic algorithm template for divide-and-conquer in multicore systems," In 12th International Conference on High Performance Computing and Communications, HPCC '10, Washington, DC, USA, 2010. IEEE Computer Society, pp. 79–88.

[15] C. A. Herrmann and C. Lengauer. 'A higher-order language for divide-and-conquer," Parallel Processing Letters, 10(02n03): 239–250, 2000.

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

[17] M. Leyton and J. M. Piquer, "Skandium: Multi-core Programming with Algorithmic Skeletons," In *PDP: Parallel, Distributed, and Network-Based Processing*. IEEE Computer Society, 2010, pp. 289–296.

[18] Frédéric Loulergue, Virginia Niculescu, Julien Tesson. "Implementing powerlists with Bulk Synchronous Parallel ML,". In 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC2014), Timisoara, Romania, 22-25 sept. 2014, IEEE Computer Society, 2014, pp 325-332

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

[20] V. Niculescu, F. Loulergue, D. Bufnea, A. Sterca, "A Java Framework for High Level Parallel Programming using Powerlists". In *18th International Conference on Parallel and Distributed Computing, Applications and Technologies* (PDCAT) 18-20 Dec. 2017, pp.255-262.

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

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

[23] G. L. Taboada and S. Ramos and R. R. Expósito and J. Touriño and R.Doallo, "Java in the High Performance Computing arena: Research, practice and experience,". Science of Comput. Program., vol. 78, no 5, pp. 425–444, 2013.