PList-based Divide and Conquer Parallel Programming

Virginia Niculescu, Darius Bufnea, and Adrian Sterca

***Abstract*—This paper details an extension of a Java parallel programming framework – JPLF. The JPLF framework is a programming framework that helps programmers build parallel programs using existing building blocks. The framework is based on *PowerLists* and *PList* theories and it naturally supports multi-way Divide and Conquer. By using this framework, the programmer is exempted from dealing with all the complexities of writting parallel programs from scratch. This extension of the JPLF framework adds *PLists* support to the framework and so, it enlarges the applicability of the framework to a larger set of parallel solvable problems. Using this extension, we may apply more flexible data division strategies, and the length of the input lists no longer has to be a power of two – as required by the *PowerLists* theory. In this paper we unveil new applications that emphasize the new class of computations that can be executed within the JPLF framework. We also give a detailed description of the data structures and functions involved in the *PLists* extension of the JPLF, and extended performance experiments are described and analyzed.**

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

I. Introduction

The computing world has been undoubtable multi-core for many years. From high performance computing (HPC) servers, to personal computers and smartphones, all have multi-core processors. The world of programming must and has taken advantage of these multi-core architectures. Developing parallel programs from scratch is cumbersome due to many difficulties that should be carefully treated, such as balanced decomposition, synchronization, or communication. Because of this, parallel frameworks, which isolate part of these issues and help the programmer writing efficient and correct parallel programs, were developed. These parallel frameworks offer the programmer some building blocks which can be used to construct a parallel software, so that the skeleton of the parallel program is already coded and the programmer only needs to address the specifics of the problem at hand.

One of the most popular paradigms for developing parallel programs is Divide and Conquer where the input data is split into parts that are processed in parallel. The JPLF framework [1] is a parallel framework written in Java that eases the development of parallel programs by providing data structures (i.e. lists) that naturally support divide and conquer computation and functions that take advantage of these parallel data structures. More specifically, the JPLF is an implementation in Java of the *PowerLists* theory [2] and of some executors required for parallel execution. Powerlists are data structures (i.e. lists) introduced by J. Misra [2] that have natural support for Divide and Conquer processing. They offer a higher level of abstraction because operating on these lists do not require low-level indexing operations. Their advantage over regular lists is that they provide two different views over the underlying data, simplifying the design of the algorithms working on them. In order to support correctness, algebras and induction principles are defined on these special data structures.

The Java programming language was chosen for the implementation due to its popularity, strong object-oriented paradigm, multi-threading and synchronization support, but also its good networking facilities. The framework was developed following object-oriented design principles like separation of concerns and applying design patterns [3], so that it would be highly flexible and end easily extensible. Besides parallel data structures, the JPLF framework also has parallel execution support for the shared memory systems (multi-threading), but also MPI support for execution on the distributed memory systems. Due to its flexible design, other execution models can be added to the framework.

This paper is an extended version of our work [4] which presents an extension of the JPLF framework by adding *PLists* support. *PLists* are a generalization of *PowerLists* introduced by J. Kornerup [5] that do not have the constraint of the data size (i.e. length of the list) being a power of two. *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. The current paper emphasizes the applicability of the framework to a larger set of parallel solvable problems through different data division strategies. We outline and review the JPLF extension from [4], define and add another application example of the extension (i.e. Fast Fourier Transform) and we specify new implementation details for achieving bounded parallelism in the JPLF *PList* extension. Also, in the current paper we extend the experimental evaluation by performing additional evaluation tests. Together: *PowerList, PList*

Manuscript received March 9, 2020; revised April 16, 2020. Date of publication May 27, 2020. Date of current version May 27, 2020. The associate editor prof. Dinko Begušić has been coordinating the review of this manuscript and approved it for publication.

The part of this paper was presented at the International Conference on Software, Telecommunications and Computer Networks (SoftCOM) 2019.

Authors are with the Computer Science Department, Babeş-Bolyai University, Cluj-Napoca, Romania (e-mails: [vniculescu, bufny, forest]@cs.ubbcluj.ro).

Digital Object Identifier (DOI): 10.24138/jcomss.v16i2.1029

1845-6421/06/1029 © 2020 CCIS

with their multidimensional counterparts could be used as a foundation for a general parallel programming model based on domain decomposition [6]; this analysis was leaded by the general characteristics that a model of parallel computation should have [7].

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

Parallel algorithmic skeletons like pipe, farm, map, reduce, ease the development of parallel programs because they provide a higher level of abstraction and they allow defining high level parallel models [8]. Frameworks like FastFlow [9] and Stapl [10] that implement parallel algorithmic skeletons simplifies the application programmer's job by dealing with the synchronization issues between sequential parts of the program and managing parallel processes themselves. Many of these parallel frameworks are implemented in C++, but Java is becoming popular in the High Performance Computing area [11]. Examples of parallel programming frameworks that implement algorithmic skeletons are: *Lithium* [12], *Calcium* [13] and *Skandium* [14].

Divide & Conquer is one of the most well known and used algorithmic skeletons in parallel programming. Different approaches have been considered to facilitate general and easy usage of Divide&Conquer pattern in parallel context [15]–[17]. The Divide & Conquer pattern can either be applied to the data through domain decomposition or to the functional tasks. Two parallel data model abstractions are *PowerLists* and *PLists*. They provide natural support for Divide & Conquer patterns based on the data domain decomposition. Prior existing work that tries to make use of the *PowerList* theory powerful abstraction are [18], [19] and [20]. In [18] authors introduce transformation rules over *PowerList*s functions in order to adapt the *PowerList*s programs for the *massively data parallel model*. [19] shows a functional implementation of *PowerList* functions in BSML (Bulk Synchronous Parallel ML). Also *PowerLists* have also been used to capture parallelism and recursion succinctly for GPU computing [20].

Java Streams are powerful functional constructs of the Java programming language that can be used in parallel programming. They are too based on algorithmic skeletons. In [1] a more detailed comparison between the performance of selected algorithms' implementations using Java parallel streams and the JPLF *Powerlist* implementation is done.

In communication, more specifically network routing dynamic programming algorithms are mostly used (e.g. Dijkstra, Bellman-Ford). These algorithms can be implemented using Divide & Conquer [21], [22] and our JPLF framework can be used to implement the Divide & Conquer programs.

Compared to the aforementioned frameworks, our JPLF framework has 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 [5]. It is a generalization of the *PowerList* data structure, which is a linear data structure whose elements are all of the same type, and with the length equal to a power of two. A *PowerList* with a single element a is called *singleton*, and it is denoted by $< a >$; if two $PowerList$ structures have the same length and elements of the same type, they are called *similar*. Two *similar* $PowerList$s can be combined into a $PowerList$ data structure with double length, using two constructors: *tie* ($p \mid q$) and *zip* ($p \natural q$), yielding, respectively, the concatenation and interleaving of two similar lists.

For *PList*s data structures we also have three constructors: one that creates singletons from simple elements, one based on concatenation, and the other based on alternative combining of two or more 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

$$[\ |\ i : i \in \overline{n} : p.i] \tag{1}$$

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

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

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

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

$$\begin{array}{lcl} <.> & : & X \rightarrow PList.X.1 \\ [\ |\ i : i \in \overline{n} : .] & : & (PList.X.m)^n \rightarrow PList.X.(n*m) \\ [\ \natural\ i : i \in \overline{n} : .] & : & (PList.X.m)^n \rightarrow PList.X.(n*m) \end{array} \tag{3}$$

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 [5].

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 following predicate:

$$defined : ((PosList \times PList) \to X) \times PosList \times PList \to Bool \quad (4)$$

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 three examples: *reduction, map* and integration through *repeated rectangle formula*. Another example for Fast Fourier Transform is presented in order to emphasize the differences between *PowerList* and *PList* functions.

Reduction

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

$$\begin{array}{lcl} defined.red(\oplus).l.p & \equiv & prod.l = length.p \\ red(\oplus).[].<a> & = & a \\ red(\oplus).(x \triangleright l).[|i : i \in x : p.i] & = & (\oplus i : 0 \leq i < x : red(\oplus).l.(p.i)) \end{array} \quad (5)$$

where $prod.l$ computes the product of the elements of the 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* an unary function f:

$$\begin{array}{lcl} defined.map(f).l.p & \equiv & prod.l = length.p \\ map(f).[].<a> & = & a \\ map(f).(x \triangleright l).[|i : i \in x : p.i] & = & (f(i) : 0 \leq i < x : map(f).l.(p.i)) \end{array} \quad (6)$$

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

Numerical Integration with the Rectangle Formula

For a function $f : [a, b] \to \mathbb{R}$, the integral

$$I = \int_a^b f.xdx \quad (7)$$

can be approximated by the following recursion [23]:

$$\begin{array}{l} Q_{D_0}.f = (b-a)f((a+b)/2) \\ Q_{D_k}.f = \frac{1}{3}Q_{D_{k-1}}.f + h\sum_{i=1}^{2m} f.x_i, \forall k > 0 \end{array} \quad (8)$$

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

$$\begin{cases} x_1 & = & a + \frac{h}{2} \\ x_2 & = & a + \frac{5}{2}h \\ x_{2j+1} & = & x_1 + 2jh \\ x_{2j+2} & = & x_2 + 2jh, \ 1 \leq j < 3^{k-1}. \end{cases} \quad (9)$$

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 the interval $[a, b]$ with $n = 3^k$ points, we have the following list:

$$[x_0, \ldots, x_{n-1}] = [a_0, a_0 + \frac{h}{3^k}, \ldots, a_0 + \frac{3^k-1}{3^k}h], \text{ where } a_0 = a + \frac{h}{2}. \quad (10)$$

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

The function

$$drept : Real \times PosList \times PList.Real.n \to Real \quad (11)$$

defined by:

$$\begin{array}{l} defined.sum.l.p \equiv prod.l = length.p \\ drept.[].<x> = hk * x \\ drept.hk.(3 \triangleright l).[\natural i : i \in \overline{3} : p.i] = \\ \quad \frac{1}{3} * drept.(3 * hk).l.(p.1) + hk * sum.(2 \triangleright l).(p.0 \natural p.2) \end{array} \quad (12)$$

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

Fast Fourier Transform

Discrete Fourier Transform is an important tool used in many scientific applications. By this transformation, the polynomial representation with coefficients $(a_i, 0 \leq i < n)$ is changed to another that consists of a list of n values, which are the polynomial values in the nth order unity roots $(w_j, 0 \leq j < n)$. The degree of the polynomial, and so the number of coefficients – n, leads to three cases:

- n is a power of two – $PowerList$ definition,
- n is a prime number – simple sequential list definition,
- n is a product of different factors – $PList$ definition.

The function $root : \mathbb{N} \to \mathbb{C}$ applied to n returns the principal nth order unity root:

$$root.n = e^{\frac{2\pi i}{n}} \quad (13)$$

Case $n = 2^k$

A formula that computes the polynomial value in $w_j (w_j = (root.n)^j)$ is:

$$f.w_j = \sum_{l=0}^{2^{k-1}-1} a_{2l} * e^{\frac{2\pi ijl}{2^{k-1}}} + e^{\frac{2\pi ij}{2^k}} \sum_{l=0}^{2^{k-1}-1} a_{2l+1} * e^{\frac{2\pi ijl}{2^{k-1}}}, \text{ where } 0 \leq j < n \quad (14)$$

PowerList data structures can be used, in this case, for the parallel program specification. An additional function is used; it returns a *PowerList* of the same length as p, containing the powers of x from 0 up to the length of p. Its *PowerList* definition is:

$$\begin{array}{l} powers.x.[a] = [x^0] \\ powers.x.(p \natural q) = powers.x^2.p \natural <x*>.(powers.x^2.q)) \end{array} \quad (15)$$

where $< x* >$ means the function that multiplies each list element with x (it is a specialization of the *map* function).

The function $fft : PowerList.\mathbb{C}.n \rightarrow PowerList.\mathbb{C}.n$ can be defined as:

$$\begin{array}{lcl} fft.<a> & = & <a> \\ fft.(p \natural q) & = & (r + u * s) \mid (r - u * s) \end{array} \tag{16}$$

where

$$\begin{array}{ll} r = fft.p, & s = fft.q \\ u = powers.z.p, & z = root.\,(length.\,(p \natural q)) \end{array}$$

The case n prime

In this case, it is necessary to directly and sequentially compute the polynomial values:

$$\begin{array}{l} fft : ParList.\mathbb{C}.n \rightarrow ParList.\mathbb{C}.n \\ fft.p = vp.p.(powers.z.p) \end{array} \tag{17}$$

where vp is a function that compute the values of a polynomial (the first argument) on a list of points (the second argument):

$$\begin{array}{l} vp.p.[v|w] = vp.p.v|vp.p.w \\ vp.(a \triangleright p).<x> = a + x * vp.p.<x> \\ vp.<a>.<x> = a \end{array} \tag{18}$$

The case $n = r_1 * \cdots * r_k$

If n is not a power of two, but is a product of two numbers r_1 and r_2, the formula from the first case can be generalized in this way:

$$\begin{array}{l} f.w_j = \sum_{k=0}^{r_1-1} \left\{ \sum_{t=0}^{r_2-1} a_{tr_1+k} e^{\frac{2\pi ijt}{r_2}} \right\} e^{\frac{2\pi ijk}{n}}, \\ 0 \leq j < n. \end{array} \tag{19}$$

So, a recursive algorithm, that combines r_1 FFT, can be used. Recursively, this can be generalized for a product of type $n = r_1 * \cdots * r_k$.

Therefore, for the specification of the parallel algorithm, it is possible to use the decomposition in prime factors $n = r_1 * \cdots * r_k$. In this case the $PList$ data structures are appropriate to be used, with a *PosList* formed by the prime factors of n: $[r_1, r_2, \ldots, r_k]$.

In this case, we have a new expression for fft, based on $PList$ structural induction principle:

$$\begin{array}{l} fft : PosList \times PList.\mathbb{C}.n \rightarrow PList.\mathbb{C}.n \\ defined.fft.l.p \equiv (prod.l = length.p) \\ fft.<x>[\natural i : i \in \overline{x} : [a.i]] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : a.i * z^{(i*j)})], \\ fft.(x \triangleright l).[\natural i : i \in \overline{x} : p.i] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : r.i * u.i.j)], \\ \text{where} \end{array} \tag{20}$$

$$\begin{array}{l} r.i = fft.l.(p.i) \\ u.i.j = < z^{(ij*\frac{n}{x})} * > .powers.(z^i).l \\ z = root.n, \quad n = length.[\natural i : i \in \overline{x} : p.i] \end{array}$$

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 TiePLists, the *tie* operator is used, and if the type is ZipPLists 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 the argument
      split_arg();
      //create the sub_functions
      List<PFunction<T>> sublists_functions =
         create_sublists_function();
      //set the partial results
      List<Object> res_sublist =
         new ArrayList<Object>();
      for (int i=0; i<sublists_functions.size();i++)
      {
         res_sublist.add(
            sublists_functions.get(i).compute());
      }
      //combine the partial results
      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 (i.e. the method depicted in Listing Code 1).

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_PFunctionExecutor provides now an implementation based on the ForkJoinPool Java executor but others could be considered, too.
- In the context of FJ_PFunctionExecutor executor, the FJ_PFunctionComputationTask class extends the RecursiveTask class, and provides the parallelization mechanism by overriding the method compute.

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 as seen in Listing Code 2. Its implementation follows the same skeleton as that used by the compute method defined for any *PList* function.

D. Recursion Depth

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).

This is visible in the code from listing 2.

E. List Transformer

Another way of reducing the parallelism, besides using the recursion_depth special attribute, is to transform the list argument into lists of sublists. If each sublist is a BasicList, the computation for these basic lists is performed sequentially.

These newly created sublists can be created using the tie or zip operators and as such, they can be tie-type lists or zip-type lists. Please remember that after applying these operators, the storage of the elements remains the same, only lists information is changed.

If we use the *tie* operator in order to transform a PList of n elements into a PList of p BasicLists, it is not mandatory to have $p|n$, but if we use *zip* instead, this condition is required.

V. Applications and Experiments

All the experiments presented in this section have been performed on a machine that supports a high level of parallelisation: an IBM x3750 M4 system equipped with 64 gigabytes of RAM and 4 Intel Xeon E5-4610 v2 @ 2.30GHz CPUs (each CPU having 8 cores). This system is running CentOS 7, a 64 bit kernel and Java 8. Each of the following tests has been repeated 5 times, the average execution time being considered.

To demonstrate de usability of our *PLists* implementation and to test the performance of our framework, the following applications were considered: *Map*, *Reduce*, and *Repeated*

```
protected Object compute() {
   Object result = null;
   if (this.function.test_basic_case()) {
      result = this.function.basic_case();
   } else {
      if (this.recursion_depth == 0) {
         //sequential execution if recursion_depth=0
         result = this.function.compute();
      } else {
         //split the argument
         this.function.split_arg();
         //create the sub_functions
         List<PFunction<T>> sub_functions =
          this.function.create_sublists_function();
         // wrap the sub_functions into executor
             tasks
         List<FJ_PFunctionComputationTask<T>>
           sub_func_exec=
             new ArrayList<
                FJ_PFunctionComputationTask<T>>();
         for ( PFunction<T> f : sub_functions) {
            sub_func_exec.add(
               new FJ_PFunctionComputationTask<T>
                 (f, this.recursion_depth - 1));
         }
         // fork the tasks
         for (int i=1; i<sub_func_exec.size(); i++){
            sub_func_exec.get(i).fork();
         }
         // set the partial results into sub_results
         // ....
         // combine the partial results
         result=this.function.combine(sub_results);
      }
   }
   return result;
}
```

Code 2: The code of the template method `compute` of the class `FJ_PFunctionComputationTask`.

Rectangle Formula. For each of the above aforementioned applications, 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 Reduce operation described in terms of the PList theory is specified in Eq. 5 in Section III. The *red* function can be described using either the *tie* or *zip* operator.

In the `Reduce` class implementation, two methods are overridden, `combine()` and `basic_case()`. The `combine()` method is overriden so that it applies the associative operator on the results obtained by performing the computation on each sublist through recoursive calls. And the `basic_case()` method is overriden just for the case when the list argument is a list of `BasicList` sublists and the Reduce computation is performed sequentially on each BasicList.

For *Reduce* we conducted several experiments:

1) *PLists* of random 10×10 matrices of real numbers, the length of the *PLists* are multiples of 5000, bounded parallelism is obtained by converting the initial PList argument to a list with BasicList sublists (Fig. 1);
2) *PLists* of random 100×100 matrices of real numbers, the length of the *PLists* are powers of 2, bounded parallelism is obtained by converting the initial PList argument to a list with BasicList sublists (Fig. 2);
3) *PLists* of random 10×10 matrices of real numbers, the length of the *PLists* are powers of 3: $3^7, 3^8, .., 3^{12}$, bounded parallelism is obtained by converting the initial PList argument to a list with BasicList sublists (Fig. 3);
4) *PLists* of random 10×10 matrices of real numbers, the length of the *PLists* are powers of 3: $3^7, 3^8, .., 3^{12}$, bounded parallelism is obtained by limiting the parallel recursion depth (Fig. 4 and 5).

Bounded parallelism was obtained in figures Fig. 1, Fig. 2 and Fig. 3 by converting the initial PList argument to a list with BasicList sublists, while the bounded parallelism used in Fig. 4 and Fig. 5 was obtained by limiting the parallel recursion depth.

For example:

- if the number of `BasicLists` 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 `BasicLists` 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 `BasicLists` 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.

The figures Fig. 1 and Fig. 2 depicts 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.

Theoretically, for the addition of a list of matrices the sequential computation is more efficient if an iterative (non-recursive) variant is considered, the bounded parallelism based on transforming the initial list of matrices into a Plist of BasicLists of matrices, should lead to better performance.

In Fig. 3 we plot the results obtained in an experiment with 10×10 matrix additions. The initial PList argument has the list length as powers of 3. We can see in this figure the execution time (in seconds) for sequential execution, unbounded parallel execution and bounded parallel execution with several values

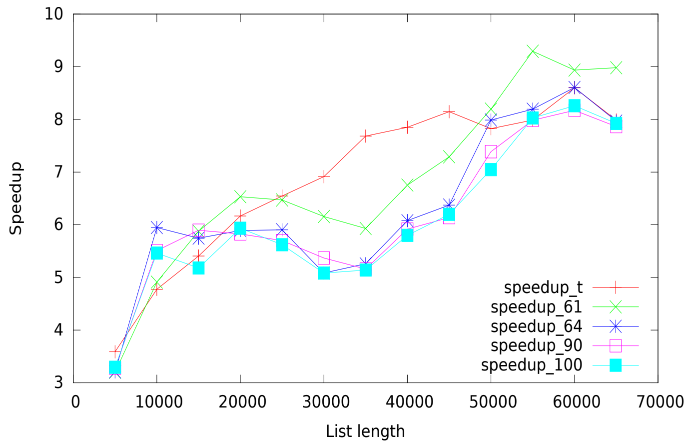


Fig. 1. *Reduce* - matrix addition: 10×10 matrices. *speedup_t* corresponds to unbounded parallelism variant, *speedup_n* correspond to bounded parallelism variant with PList with n elements of type BasicList.

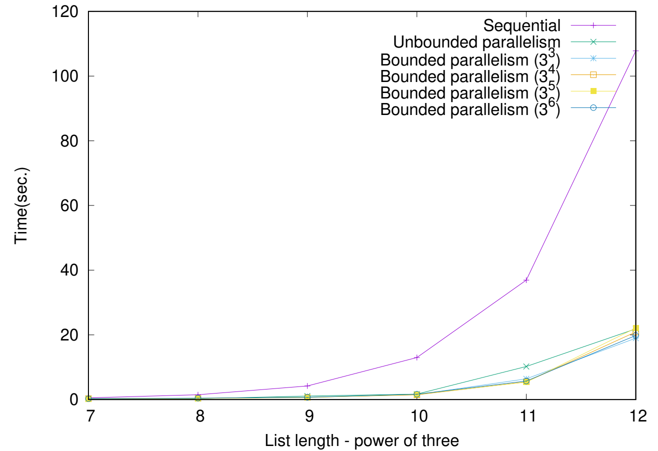


Fig. 3. *Reduce* - execution times for 10×10 matrices addition. OX axis shows the list length as powers of 3: $3^7, 3^8, .., 3^{12}$. Bounded parallelism is obtained by splitting the initial PList argument into n (n is power of 3) elements of type BasicList.

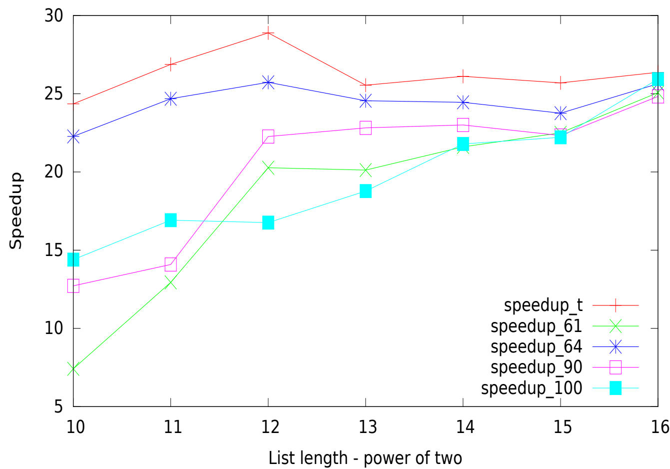


Fig. 2. *Reduce* - matrix addition: 100×100 matrices. *speedup_t* corresponds to unbounded parallelism variant, *speedup_n* correspond to bounded parallelism variant with PList with n elements of type BasicList.

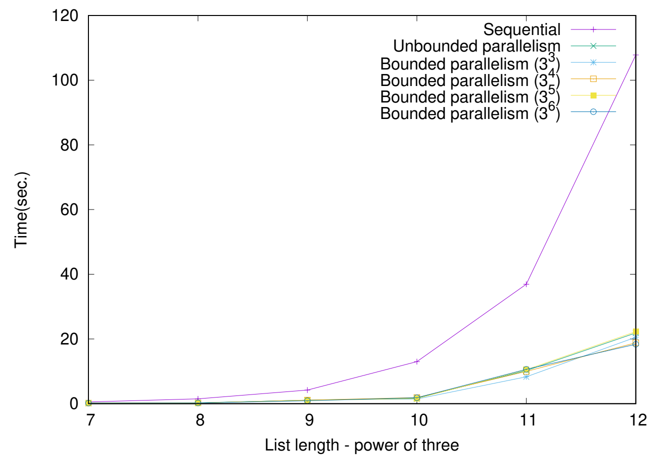


Fig. 4. *Reduce* - execution times for 10×10 matrices addition. OX axis shows the list length as powers of 3: $3^7, 3^8, .., 3^{12}$. Bounded parallelism is obtained by limitting the parallel recursion depth.

of the number n of BasicLists that the initial PList argument is split into. We can see that the parallel executions are much better than the sequential one, and the differences between unbounded parallelism and the bounded parallelism variants are not very high.

Fig. 4 shows the results of another experiment similar to the previous one: 10×10 matrix additions, initial PList argument has the list length as powers of 3. But this time bounded parallelism execution is obtained by limiting the parallel recursion depth (which is still powers of 3 – the number written in brackets in this figure). We can see here that unbounded parallel execution and all variants of bounded parallel execution performed better than sequential execution as expected. Experiment 4 emphasizes the fact that bounding the parallelism just by setting parallel recursion depth, doesn't limit the performance very much, especially for the cases when the list length is not very big. In order to compare unbounded parallelism with bounded parallelism, we replotted the same data without the sequential execution time in Fig. 5. We can

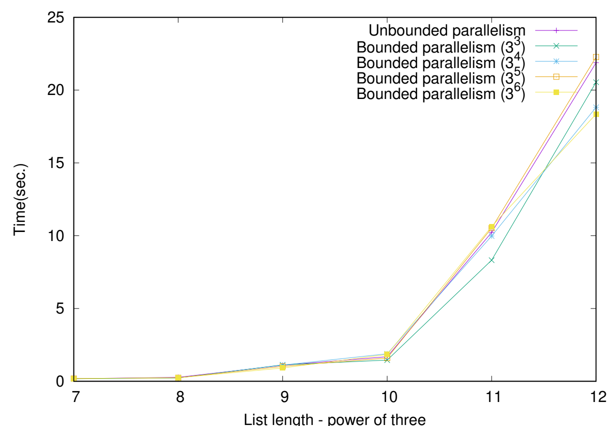


Fig. 5. *Reduce* - execution times for 10×10 matrices addition without sequential execution. OX axis shows the list length as powers of 3: $3^7, 3^8, .., 3^{12}$. Bounded parallelism is obtained by limitting the parallel recursion depth.

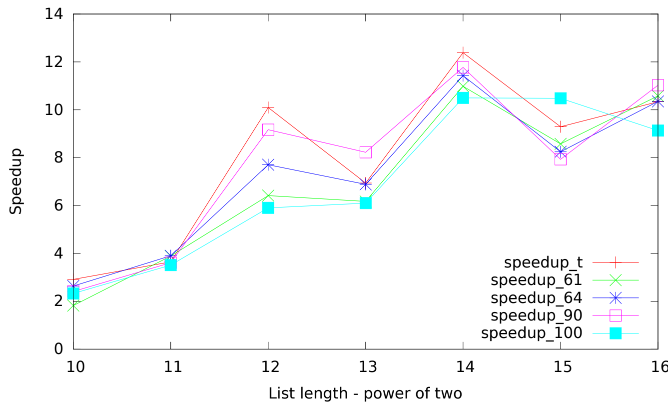


Fig. 6. *Map* – applying squaring on each element of a list of 100×100 matrices. *speedup_t* corresponds to unbounded parallelism variant, *speedup_n* correspond to bounded parallelism variant with PList with n elements of type BasicList.

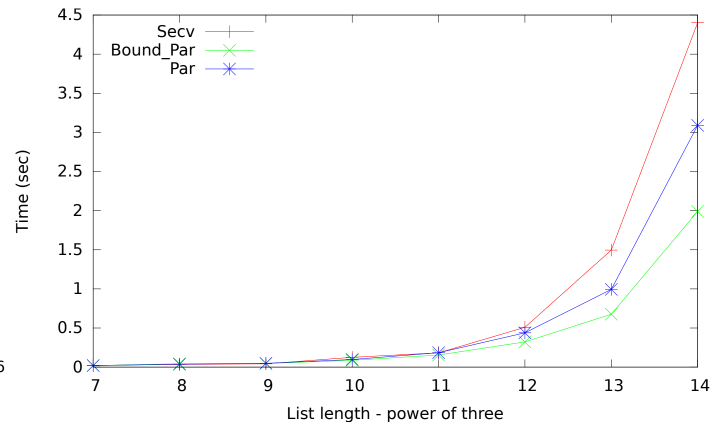


Fig. 7. *Repeated Rectangle Formula*: execution time –
Sequential execution vs Parallel execution vs. Bounded Parallel execution

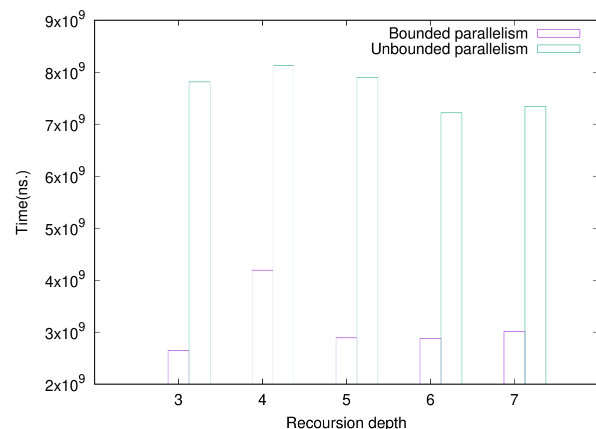


Fig. 8. *Repeated Rectangle Formula*: execution time for data size 2^{13} and various parallel recursion depths –
Bounded Parallel execution vs. Unbounded Parallel execution

see here that for list lengths larger than 3^{10} bounded parallelism has greater performance that unbounded parallelism and the bound level seems to matter: the 3^3 bounded parallelism obtained the smallest execution time (except the value obtained for list length 3^{12}) – which makes sense because 27 is the bound level closest to the number of actual cores in the system (i.e. 32).

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.

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. 6 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. For the *Map* experiments, the number of parallel tasks is bounded in the same way as for the *Reduce* tests, because as in the matrix addition case the sequential computation of map is more efficient for large data sets if done iteratively (non-recursive). We can notice in this figure, as we saw for the Reduce experiments that for large data sets or if the *basic_case* and/or the *combine* functions are computational intensive, the difference in performance between bounded and unbounded parallelism variants is 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. 7.

The variant that considers bounded parallelism is based on the limitation of the parallel recursion depth. For the presented test the `recursion_depth` argument in the FJ_PFunctionExecutor constructor, 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.

In Fig. 8 we can see the effect of different parallel recursion depths on the performance of bounded parallel execution over unbounded parallel execution. We can see in this figure that all recursion depth levels for bounded parallelism achieved an execution time much less than the one obtained by unbounded parallel execution and there is no exact recursion depth level which performed significantly better than the others (although

we can see that the recursion depth level of 4 achieved the worse results - we have no explanation why this happened).

VI. Conclusions

The *PList* theory is very powerful, being an extension and a generalization of the *PowerList* theory. *PList* data structures can be split into a different number of sublists, and the functions defined on them specify multi-way divide-and-conquer algorithms, which allows not only splitting a problem into a number of subproblems greater than two, but also variants for which at each level a different number of subproblems are split. For some problems this ability is mandatory – as it is the case of *Repeated Rectangle Formula*, or the case of *Fast Fourier Transform* with arbitrary factors. But, this ability is very important also for the cases when splitting can be done in any number of subproblems (e.g. map and $reduce$ functions), because it allows choosing the best variant from the performance point of view. The fact that a *PList* data structure can be constructed in two different ways, differentiate this from other list based data structures that rely on simple concatenation for building new instances. From the implementation point of view, the fact that at each step a list can be split into a different number of sublists (and so subproblems), brings a degree of flexibility that is useful when one has to choose the most performant partition of the problem. In extremis, the arity list could be considered as being formed of only one element equal to the size of list. In this way, any computation that fits into the "embarrassingly parallel" pattern might be defined based on this theory. One special feature of the described framework is its capability to accept different recursion levels of parallelisation. In this way, the number of tasks that are executed in parallel is controlled, with direct impact on the practical performance. Another extremely important feature of the framework is its ability to work with lists of lists. It allows different combinations of list types, the programmer being able to combine the associated computation paradigms. For example, a *BasicList* of *PowerLists* or *PLists* elements allows SEQ-PAR computation, vice-versa being also possible - PAR-SEQ computation, if *PowerList* (or a *PList*) of *BasicLists* is used. By using different list types, different types of execution are performed. This combination can also be done on different or multiple levels, increasing in this way the possibility to express different types of computation.

References

[1] V. Niculescu, F. Loulergue, D. Bufnea, A. Sterca, "A Java Framework for High Level Parallel Programming using Powerlists", in *Proceedings of the 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'17)*, pp. 255–262, DOI: 10.1109/PDCAT.2017.00049, December 18-20, 2017, Taipei, Taiwan.

[2] J. Misra, "Powerlist: A structure for parallel recursion", in *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1737–1767, DOI: 10.1145/197320.197356, November 1994.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design patterns: elements of reusable object-oriented software", Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN:978-0-201-63361-0.

[4] V. Niculescu, D. Bufnea, A. Sterca, "Multi-way Divide and Conquer Parallel Programming based on PLists", in *Proceedings of the 27th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–6, DOI: 10.23919/SOFTCOM.2019.8903794, September 19-21, 2019, Split, Croatia.

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

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

[7] D. Skillicorn, D. Talia, "Models and languages for parallel computation", *Computing Surveys*, vol. 30, no. 2, pp. 123–169, DOI: 10.1145/280277.280278, June 1998.

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

[9] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, "FastFlow: high-level and efficient streaming on multi-core", in *Programming Multi-core and Many-core Computing Systems*, S. Pllana and F. Xhafa, Ed., Wiley, 2014.

[10] M. Zandifar, M. Abduljabbar, A. Majidi, D. Keyes, N. Amato, L. Rauchwerger, "Composing Algorithmic Skeletons to Express High-Performance Scientific Applications", in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 415-–424, DOI: 10.1145/2751205.2751241, 2015.

[11] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, R.Doallo, "Java in the High Performance Computing arena: Research, practice and experience", *Science of Comput. Program.*, vol. 78, no 5, pp. 425–444, DOI: 10.1016/j.scico.2011.06.002, 2013.

[12] M. Aldinucci, M. Danelutto, P. Teti, "An advanced environment supporting structured parallel programming in Java", *Future Generation Computer Systems*, vol. 19, pp. 611–626, DOI: 10.1016/S0167-739X(02)00172-3, 2003.

[13] D. Caromel, M. Leyton, "A transparent non-invasive file data model for algorithmic skeletons", in *Proceedings of 22nd IEEE International Symposium on Parallel and Distributed Processing*, IPDPS 2008, pp. 1–10, DOI: 10.1109/IPDPS.2008.4536276, Miami, Florida USA, April 14-18, 2008.

[14] M. Leyton, J. M. Piquer, "Skandium: Multi-core Programming with Algorithmic Skeletons", in *PDP: Parallel, Distributed, and Network-Based Processing*, pp. 289–296, DOI: 10.1109/PDP.2010.26, IEEE Computer Society, 2010.

[15] M. Danelutto, T. De Matteis, G. Mencagli, 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, ACM, pp. 10–19, DOI: 10.1145/3002125.3002128, Amsterdam, 2016.

[16] C. H. Gonzalez, B. B. Fraguela. "A generic algorithm template for divide-and-conquer in multicore systems", in *Proceedings of 12th International Conference on High Performance Computing and Communications*, HPCC '10, pp. 79—88, DOI: 10.1109/HPCC.2010.24, Washington, DC, USA, 2010, IEEE Computer Society.

[17] C. A. Herrmann, C. Lengauer. '$\mathcal{HDC}$: A higher-order language for divide-and-conquer", in *Parallel Processing Letters*, Vol. 10, No. 02n03, pp. 239–250, DOI: 10.1142/S0129626400000238, 2000.

[18] K. Achatz, W. Schulte, "Architecture independent massive parallelization of divide-and-conquer algorithms", in *Proceedings of International Conference on Mathematics of Program Construction*, MPC 1995, Lecture Notes in Computer Science, vol 947, pp. 97–127, DOI: 10.1007/3-540-60117-1_7, Springer, Berlin, Heidelberg.

[19] F. Loulergue, V. Niculescu, J. Tesson, "Implementing powerlists with Bulk Synchronous Parallel ML", in *Proceedings of 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, (SYNASC2014), Timisoara, Romania, 22-25 sept. 2014, pp. 325–332, DOI: 10.1109/SYNASC.2014.51, IEEE Computer Society.

[20] A. S. Anand, R. K. Shyamasundarn, "Scaling computation on GPUs using powerlists," in *Proceedings of the 22nd International Conference on High Performance Computing Workshops (HiPCW)*, pp. 34–43, DOI: 10.1109/HiPCW.2015.14, Bangalore, 2015.

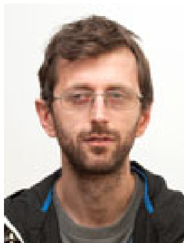
[21] Y. Zhang, Y. Mei, "Divide-and-Conquer Large Scale Capacitated Arc Routing Problems with Route Cutting Off Decomposition", arXiv:1912.12667, 2019.

[22] S. Itzhaky, R. Singh, A. Solar-Lezama, K. Yessenov, Y. Lu, C. Leiserson, R. Chowdhury, "Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations", in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 145—164, DOI: 10.1145/3022671.2983993, Amsterdam, Netherlands.

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

Virginia Niculescu is an associate professor in the Department of Computer Science at (UBB) Babeş-Bolyai University of Cluj-Napoca. She received her PhD in Computer Science (2002) from the same university, where she also completed her bachelor degree (1994). Between the topics taught by her are parallel and distributed computating, data structures and algorithms, object-oriented programming and design, and worflow systems. She coordinates UBB Research Center of Modeling, Optimization and Simulation, and the master programme High Performance Computing and Big Data Analytics. Her research interests are focused mainly on models of parallel and distributed computing, scientific computation, but also on design patterns of different programming paradigms.

Darius Bufnea is a lecturer at the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Romania. He received the Ph.D. degree from Babeş-Bolyai University in 2008. He was the director of one research grant funded by the Romanian funding agency and member in other four. He has authored more than 30 research papers on networking, congestion control, information retrieval, parallel computing and web technologies. He is the coauthor of four books that cover different computer science related topics. His current focus and interest is on cybersecurity.



Adrian Sterca is a lecturer at the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Romania. He received the Ph.D. degree from Babeş-Bolyai University in 2009. He directed 2 research grants founded by the Romanian funding agency and was a member in other four grants. He has authored several research papers on networking, multimedia systems and information retrieval. His current research interests are networking, multimedia streaming, image processing, information retrieval and parallel computing.