ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **14**, Number 2, 2011, 159–182

PARES – A Model for Parallel Recursive Programs

Virginia NICULESCU

Faculty of Mathematics and Computer Science, Babeş-Bolyai University,
1 M. Kogălniceanu, Cluj-Napoca, România

E-mail: vniculescu@cs.ubbcluj.ro

Abstract. *PowerList*, *ParList*, and *PList* theories and their multidimensional extensions *PowerArray*, *ParArray*, and *PArray* are well suited to express recursive, data-parallel algorithms. Their abstractness is very high and assures simple and correct design of parallel programs. Base on these theories we define a model of parallel computation with a very high level of abstraction – PARES (Parallel Recursive Structures). A model of parallel computation, to be useful must address the following set of requirements: abstractness, software development methodology, architecture independence, cost measures, no preferred scale of granularity, efficiently implementable. We show in this paper that all these requirements are fulfilled for the proposed model.

Key-words: parallel programming, model, data-structures, divide&conquer, recursion.

1. Introduction

PowerList, *ParList*, and *PList* are data structures introduced by J. Misra [12] and J. Kornerup [9], which can be successfully used in a simple and provable correct, functional description of parallel programs, which are *divide and conquer* in nature. They allow working at a high level of abstraction, especially because the index notations are not used. To assure methods that verifies the correctness of the parallel programs, algebras and structural induction principles are defined on these data structures. Based on the structural induction principles, functions and operators, which represent the parallel programs, are defined. These data structures can be easily extended to the multidimensional case, and the resulted structures are *PowerArray*,

ParArray, and *PArray*. They allow the specification of recursive parallel programs that works with multidimensional data in more efficient and suggestive way. Working with multidimensional arrays instead of nested lists also brings important advantages at the design phase.

These theories can be considered the base for a model of parallel computation with a very high level of abstraction. The proposed model PARES (*Parallel Recursive Structures*) includes *PowerList*, *ParList*, and *PList* theories and their multidimensional extensions *PowerArray*, *ParArray*, and *PArray* together with the data-distributions defined on them, that allow the definition of parallel programs with different scale of granularity. The programs are defined as functions on the specified structures and their correctness is assured by using structural induction principles.

A model of parallel computation, to be useful must address both issues, abstraction and effectiveness, which are summarized in the following set of requirements: abstractness, software development methodology, architecture independence, cost measures, no preferred scale of granularity, efficiently implementable [19]. We will prove that all these requirements are fulfilled for the PARES model defined based on these theories.

The first three requirements are easily proved based on the definitions of data structures and corresponding algebras – these are analyzed in Section **??**. In this kind of abstract models the parallelism is implicit, and hence the decomposition, communication, synchronization and mapping are implicit (if the models classification specified in [19] is considered). So, generally, unbounded parallelism (the number of processes is not limited) is analyzed using these structures. Still, the most practical approach of bounded parallelism can be introduced, and so, the distributions, too. Also, based on these, realistic cost measures can be rigorously defined. Section **??** presents how distributions may be defined on these special kinds of data structures, and also how the functions defined on them could be transformed to accept distributions. In this way, the high level of abstraction of these theories is reconciled with the performance, and so, together, they could form a parallel computation model.

For other similar theories, these kinds of enhancement have been analyzed, too. There is a clear necessity to reconcile abstraction with performance, as it is stated by S. Gorlatch in [5].

BMF formalism [2, 5] is based on recursion too, and there the notion of homeomorphism is essential. The distributions have been introduced as simple functions that transform a list into a list of lists. But, since few of the key distributions, such as block decomposition, can be defined in this calculus, so various hybrid forms, often called *skeletons* [3] have been introduced to bridge the gap.

Shape theory [6] is a more general approach. Knowledge of the shapes of the data structures is used by many cost models [7]. Static shape analysis can be applied to those programs for which the shape of the result is determined by that of inputs, i.e. the *shapely programs*. PowerList, *ParList*, and *PList* theories allow us to define shapely programs, but in a very elegant and simple way. They have proved to be highly successful in expressing computations that are independent of the specific data values.

2. Power, Par, and P Theories

PowerList. A *PowerList* is a linear data structure whose elements are all of the same type. The length of a *PowerList* data structure is a power of two. The type constructor for *PowerList* is:

$$PowerList : Type \times \mathbb{N} \to Type \tag{1}$$

and so, a *PowerList* with 2^n elements of type X is specified by $PowerList.X.n$ ($n = loglen.l$). 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.

There is no way to directly address a particular element. $PowerList$ algebra is defined by these operators and by axioms that assure the existence of unique decomposition of a $PowerList$, using one of *tie* or *zip* operator; and the fact that *tie* and *zip* operators commute [9]:

$$(p \mid q) \natural (u \mid v) = (p \natural u) \mid (q \natural v)$$

ParList. The $ParList$ data structure is analogue with $PowerList$, with the difference that the number of the elements is not a power of two. The type constructor for $ParList$ is:

$$ParList : Type \times \mathbb{N}^* \to Type \tag{2}$$

and a $ParList$ with n elements of type X is specified by $ParList.X.n$.

It is necessary to use other two constructor operators: *cons*($\triangleleft$) and *snoc*($\triangleright$); they allow adding an element to a $ParList$ at the beginning or at the end of the $ParList$.

Axioms of $ParList$ algebra express like those from $PowerList$ algebra, the existence of a unique decomposition of a $ParList$, using constructor operators, the commutativity of *tie* and *zip* operators, and some axioms that make connection among operators [9].

PList. $PList$ data structure is a generalization of PowerList, constructed with n-way $\mid$ and $\natural$ operators; e.g. for positive n the n-way $\mid$ operator takes n similar $PList$s and returns their concatenation. While the $PowerList$ notation is tied to radix 2, the $PList$ notation enables us to state properties and algorithms in the radix that is more suited for the problem. The $PList$ notation is even more general; it allows the use of mixed radices in specifications and facilitates algebraic reasoning about such specifications [10].

A $PList$ is a non-empty linear data structure, whose elements are all of the same type, either scalars from the same base type, or recursively $PList$s that enjoy the same property. Square brackets are used to denote ordered quantification in $PList$

algebra. The expression $[|i : i \in \overline{n} : p.i]$ is a closed form for the application of the n-way operator $|$, applied to the $PList$ elements $p.i$. The notation $i \in \overline{n}$ means that the terms of the expression are written from 0 trough $n-1$ in the numerical order. The $PList$ axioms, define also the existence of unique decomposition of $PList$ using constructor operators [9].

PowerArray, ParArray, PArray. The type function for $PowerArray$ data structure

$$PowerArray : Type \times (\mathbb{N})^n \to Type \tag{3}$$

has one argument a type (X) and n arguments that are positive natural numbers. This constructor returns the type of all data structures with elements of type X and with 2^{n_i} elements on the $i-$th dimension. For example, if we consider that the first dimension is for columns and the second for rows, $PowerArray.\mathbb{N}.2.3$ is the type of all matrices with 2^2 columns and 2^3 rows, with natural numbers as elements.

If p, q are two similar $PowerArray$s we can use two kinds of construction operators *tie* and *zip*:

$$u = p \mid_i q$$
$$v = p \natural_i q, \text{ where } i = 0, 1.$$

For example, if $p, q \in PowerArray.X.m.n$ and $u = p \mid_0 q$, $v = p \natural_0 q$ then $u, v \in PowerArray.X.(m+1).n$. Now, u is the matrix formed by the columns of p followed by the columns of q and v is the matrix whose columns are alternatively taken from p and q.

The types of the $PowerArray$ constructors are as follows:

$[.] : X \to PowerArray.X.0$
$.|_0. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.(m+1).n$
$.|_1. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.m.(n+1)$
$.\natural_0. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.(m+1).n$
$.\natural_1. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.m.(n+1)$

In a similar way the $ParList$ and $PList$ theories are extended to $ParArray$ and $PArray$ theories.

3. Abstractness, Software development methodology, Architecture independence

The definition of these notations and data structures implies a very high degree of abstractness.

The correctness of the parallel programs specified with these notations can be proved based on the algebras and structural induction principles defined on the corresponding data structures.

On $PowerList$ data structures the corresponding induction principle, that allows function definitions, and the proving of $PowerList$ properties, is defined as follows:

If $\Pi : PowerList.X.n \to Bool$ is a predicate, the induction principle is:

$$\begin{array}{l}((\forall x : x \in X : \Pi.[x]) \\ \quad \wedge((\forall p, q, n : p, q \in PowerList.X.n \wedge n \in \mathbb{N} \ : \ \Pi.p \wedge \Pi.q \Rightarrow \Pi.(p \mid q)) \\ \qquad \vee \quad (\forall p, q, n : p, q \in PowerList.X.n \wedge n \in \mathbb{N} \ : \ \Pi.p \wedge \Pi.q \Rightarrow \Pi.(p \natural q))) \\ \Rightarrow \\ \qquad (\forall p, n : p \in PowerList.X.n \wedge n \in \mathbb{N} : \Pi.p)\end{array}$$

Functions are defined using pattern matching, and their properties are proved based on the structural induction principle. For example, the high order function *map*, which applies a scalar function to each element of a *PowerList* is defined as follows:

$$\begin{array}{l} map : (X \to Z) \times PowerList.X.n \to PowerList.Z.n \\ map.f.[a] = [f.a] \\ map.f.(p \mid q) = map.f.p \,|\, map.f.q \end{array} \tag{4}$$

An induction principle is also defined for *ParList*s. But, in this case, a proving has three stages: the base case, the odd inductive step and the even inductive step. The rule of structural decomposition for *ParList* is as follows: when the number of the elements is even, then *tie* or *zip* operators are used, and when this number is odd, then *cons* or *snoc* are used. This way, the decomposition is unique.

The *ParList* function definition must contain definitions corresponding to these three stages.

For example, the function *reduce* is defined as follows:

$$reduce : (\oplus) \times ParList.X.n \to Y$$

is defined by:

$$\begin{array}{lcl} reduce.\oplus.[a] & = & [a] \\ reduce.\oplus.(p \,|\, q) & = & reduce.\oplus.p \ \oplus \ reduce.\oplus.q \\ reduce.\oplus.(a \lhd q) & = & a \ \oplus \ reduce.\oplus.q. \end{array} \tag{5}$$

where the first argument is a binary associative operator on X type. The equivalent definition that uses $\natural$ instead of $|$ can be used, too.

As for the previous data structures an induction principle is defined for *PList*s, too [9]. Functions over *PList* are defined using two arguments. The first argument is a list of arities: *PosList*, and the second is the *PList* argument. (*PosList* is the type of general linear lists *List* that contain only positive numbers – $List.\mathbb{N}^*$. The empty linear list is denoted by $[]$, and we have the operators *cons* and *snoc* defined on these lists.) Functions over *PList* are only defined for certain pairs of these input values; to express the valid pairs the specification of the function has to define the following predicate:

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

to characterize where the function is defined.

We illustrate this, by defining the function *sum*. This function computes the sum of all elements of a $PList$ over a type where operator + is defined:

$$\begin{array}{lcl} defined.sum.l.p & \equiv & prod.l = length.p \\ sum.[].[a] & = & a \\ sum.(x \triangleleft l).[|i : i \in \overline{x} : p.i] & = & (+i : i \in \overline{x} : sum.l.(p.i)) \end{array}$$

where $prod.l$ computes the product of the elements of the list l, $length.p$ is the length of p.

For $PowerArray$s we have an induction principle, that includes a proof on each dimension. For example, any associative operator can be extended on $PowerArray$s; in the bidimensional case, the extended operator has the following type:

$$\odot : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.m.n$$

and is defined, based on the structural induction principle, by:

$$\begin{array}{lcl} [a] \odot [b] & = & [a \odot b] \\ (p|_0 q) \odot (u|_0 v) & = & (p \odot u) \mid_0 (q \odot v) \\ (p|_1 q) \odot (u|_1 v) & = & (p \odot u) \mid_1 (q \odot v). \end{array}$$

The $\odot$ operator can be also defined with $\natural$ constructor operator, or using $\natural$ on one dimension and $|$ on the other.

There is a strong similarity between a formal specification and a functional program, which can be considered as a composition of operators that modify the universe of the solving problem. Therefore, in order to create a PARES program for a certain specification we may start from an initial expression (derived from specification), and then to refine it based on equationally equivalent transformations. These transformations could be seen as a software development methodology.

Example 1 (Prefix sum). Given a list of scalars and an associative binary operator $\oplus$ on these scalars, the prefix sum returns a list of the same length where each element is the result of applying the operator on the elements up to and including the element in that position in the original list. We can assume that the operator $\oplus$ has an identity element 0.

For computing the prefix sum of a list of numbers, different programs can be defined.

The operator * on $PowerList$s shifts the elements of the list one position to the right and adds a zero in the leftmost position (the rightmost element is lost by this operation).

The prefix sum of a list l, $PS.l$ can be specified as the unique solution to the following equation (in z) [12]:

$$z : z = z^* \oplus l \tag{6}$$

A well known algorithm for computing the prefix sum is due to Ladner and Fischer [11]. In the $PowerList$ notation it can be written as:

$$LF.(p \natural q) = (LF.(p \oplus q))^* \oplus p \natural LF.(p \oplus q) \tag{7}$$

A proof by structural induction of the correctness of LF is presented by J. Misra in [12].

Example 2 (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. The new representation consists of a set of values, which are the polynomial values in the nth order unity roots $(w_j, 0 \leq j < n)$. The number of coefficients – n – leads to three cases.

A scalar function will be used in all the cases. The function $root : \mathbb{N} \to \mathbb{C}$ applied to n returns the principal nth order unity root :

$$root.n = e^{\frac{2\pi i}{n}}. \tag{8}$$

The case $n = 2^k$
PowerList data structures can be used, in this case, for the parallel program specification. A formula that computes the polynomial value in w_j is:

$$f.w_j = \sum_{l=0}^{2^{k-1}-1} a_{2l} * e^{\frac{2\pi ijl}{2^{k-1}}} + e^{\frac{2\pi ij}{2^k}} \sum_{l=0}^{2^{k-1}-1} a_{2l+1} * e^{\frac{2\pi ijl}{2^{k-1}}},\ 0 \leq j < n \tag{9}$$

We use an additional function $powers : Com \times PowerList.\mathbb{C}.n \to PowerList.\mathbb{C}.n$ that returns a $PowerList$ of the same length as p, containing the powers of x from 0 up to the length of p. It is defined by:

$$\begin{array}{lcl} powers.x.[a] & = & [x^0] \\ powers.x.(p \natural q) & = & powers.x^2.p \natural < x* > .(powers.x^2.q)) \end{array} \tag{10}$$

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

The definition of fft can be formally obtained starting from the expression:

$$fft.p = vp.p.(powers.z.p) \tag{11}$$

where the function vp computes the values of a polynomial (given by the list of its coefficients) in more points given by the second list argument:

$$\begin{array}{l} vp : PowerList.X.n \times PowerList.X.m \to PowerList.X.m \\ vp.[a].[w] = [a] \\ vp.p.(u|v) = vp.p.u|vp.p.v \\ vp.(p\#q).w = vp.p.w^2 \oplus (w \odot (vp.q.w^2)) \end{array} \tag{12}$$

The complete derivation for fft can be found in [12]. We present here just a shorter version in order to emphasize the software development methodology for this kind of programs.

The derivation starts from the idea that we have to find an equivalent expression of the function, which has to be more efficient – with a smaller time-complexity.

We denote $W.z.p = powers.z.p$, and we have two properties that are based on the properties of the nth order unity root z:

$$\begin{array}{l} W^2.z.(p \natural q) = W.z^2.p | W.z^2.q \\ W.z.(p \natural q) = W.z.p | - W.z.q \end{array} \tag{13}$$

Base Case:

$$\begin{array}{ll} & fft.[a] \\ = & \{\text{definition of } fft\} \\ & vp.[a].z \\ = & \{\text{definition of } vp\} \\ & [a] \end{array}$$

Inductive Step:

$$\begin{array}{ll} & fft.(p \natural q) \\ = & \{\text{definition of } fft\} \\ & vp.(p \natural q).(W.z.(p \natural q)) \\ = & \{\text{definition of } vp\} \\ & vp.p.(W.z.(p \natural q))^2 + (W.z.(p \natural q)) \cdot (vp.q.(W.z.(p \natural q))^2) \\ = & \{\text{property of } W\} \\ & vp.p.(W.z^2.p | W.z^2.q) + W.z.(p \natural q) \cdot vp.p.(W.z^2.p | W.z^2.q) \\ = & \{ \text{ definition of } vp\} \\ & vp.p.(W.z^2.p) | vp.q.(W.z^2.q) + (W.z.(p \natural q)) \cdot (vp.p.(W.z^2.p) | vp.q.(W.z^2.q)) \\ = & \{\text{definition of } fft\} \\ & fft.p | fft.q + (W.z.(p \natural q)) \cdot (fft.p | fft.q) \\ = & \{\text{property of } W\} \\ & fft.p | fft.q + (W.z.p | - W.z.q)) \cdot (fft.p | fft.q) \\ = & \{\text{definition of the operator } \cdot\} \\ & fft.p | fft.q + (W.z.p \cdot fft.p) | (-W.z.p) \cdot fft.q) \\ = & \{ \text{ operators definition } \} \\ & (fft.p + W.z.p \cdot fft.q) | (fft.p - W.z.p \cdot fft.q) \end{array}$$

So, the function $fft : PowerList.\mathbb{C}.n \to PowerList.\mathbb{C}.n$ can be defined as:

$$\begin{array}{lcl} fft.[a] & = & [a] \\ fft.(p \natural q) & = & (r + u * s) \mid (r - u * s) \end{array} \tag{14}$$

where

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

The case n prime

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

$$\begin{array}{l} fft : ParList.\mathbb{C}.n \to ParList.\mathbb{C}.n \\ fft.p = vp.p.(powers.z.p) \end{array} \tag{15}$$

The function $powers : Com \times ParList.\mathbb{C}.n \to ParList.\mathbb{C}.n$ is an extension of the one presented in the first case, defined on $PowerList$:

$$\begin{array}{lcl} powers.x.[a] & = & [x^0] \\ powers.x.(p \natural q) & = & powers.x^2.p \natural \ < x* > .(powers.x^2.q)) \\ powers.x.(a \lhd q) & = & [x^0] \lhd \ < x* > .(powers.x.q)). \end{array} \tag{16}$$

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:

$$f.w_j = \sum_{k=0}^{r_1-1} \left\{ \sum_{t=0}^{r_2-1} a_{tr_1+k} e^{\frac{2\pi ijt}{r_2}} \right\} e^{\frac{2\pi ijk}{n}}, \quad 0 \le j < n. \tag{17}$$

The inner sum represents the value at $w_{j \ mod \ r_2}$, of the polynomial with the degree equal to r_2–1 and the coefficients $\{a_k, a_{k+r_1}, \ldots, a_{k+r_1(r_2-1)}\}$. This value is computed by FFT for this polynomial. So, a recursive algorithm, that combines r_1 FFT, can be used. Recursively, this can be generalized for a product of type $n = r_1 * \cdots * r_k$.

Theorem 1. *The best factorization* $n = r_1 * r_2$ *for* FFT *(from the complexity point of view) is to choose* r_1 *from the prime factors of* n.

A proof of this theorem is given in [20].

Therefore, for the specification of the parallel algorithm, we consider the decomposition in prime factors $n = r_1 * \cdots * r_k$. The $PList$ data structures will be used.

$$fft : PosList \times PList.\mathbb{C}.n \to PList.\mathbb{C}.n \tag{18}$$

The $PosList$ is 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, and the proof can be found in [13]:

$$\begin{array}{l} defined.fft.l.p \equiv (prod.l = length.p) \\ fft.[x][\natural i : i \in \overline{x} : [a.i]] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : a.i * z^{(i*j)})], z = root.x \\ fft.(x \lhd l).[\natural i : i \in \overline{x} : p.i] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : r.i * u.i.j)], \text{ where} \end{array} \tag{19}$$

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

Remarks:

– For the base case it can be used the algorithm presented when n is prime,

$$fft.\,[x]\,[\natural i : i \in \bar{x} : [a.i]] = fft_{|ParList}.[\natural i : i \in \bar{x} : [a.i]] \tag{20}$$

which is more efficient.

– If the list of arities contains just values equal to 2, the algorithm becomes that specified in the first case.

– The algorithm for Fast Fourier Transformation can be done simultaneously with the decomposition of n in prime factors. If the prime factors become too large, then we can stop and apply the algorithm used when n is prime.

4. Variable scale of granularity: Distributions

The ideal method to implement parallel programs described with *PowerLists* is to consider that any application of the operators *tie* or *zip* as deconstructors, leads to two new processes running in parallel, or, at least, to assume that for each element of the list there is a corresponding process. This means that the number of processes grows linearly with the size of the data. In this ideal situation, the time-complexity is usually logarithmic (if the combination step complexity is a constant), depending on *loglen* of the input list.

A more practical approach is to consider a bounded number of processes n_p. In this case we have to transform de input list, such that no more than n_p processes are created. This transformation of the input list corresponds to a data distribution.

4.1. *PowerList* Distributions

We consider *PowerList* data structures with elements of a certain type X, and with length such that $loglen = n$. The number of processes is assumed to be limited to $n_p = 2^p$ $(p \leq n)$.

Two types of distributions – linear and cyclic, which are well-known distributions, may be considered. These correspond in our case to the operators *tie* and *zip*. Distributions are defined as *PowerList* functions, so definitions corresponding to the base case and to the inductive step have to be specified:

– linear

$$\begin{array}{l} distr^l.p.(u|v) = distr^l.(p{-}1).u \,|\, distr^l.(p{-}1).v, \text{ if } loglen.(u \,|\, v) \geq p \wedge p > 0 \\ distr^l.0.l = [l] \\ distr^l.p.x = [x], \text{ if } loglen.x < p. \end{array} \tag{21}$$

– cyclic

$$\begin{array}{l} distr^c.p.(u\natural v) = distr^c.(p{-}1).u \natural distr^c.(p{-}1).v, \text{ if } loglen.(u \natural v) \geq p \wedge p > 0 \\ distr^c.0.l = [l] \\ distr^c.p.x = [x], \text{ if } loglen.x < p. \end{array} \tag{22}$$

The base cases transform a list l into a singleton, which has the list l as its unique element.

Example 3. If we consider the list $l = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8]$, the lists obtained after the application of the distribution functions $distr^l.2.l$ and $distr^c.2.l$ are:

$$distr^l.2.l \ = distr^l.1.[1\ 2\ 3\ 4] \mid distr^l.1.[5\ 6\ 7\ 8] \ = \ [[1\ 2]\ [3\ 4]\ [5\ 6]\ [7\ 8]]$$
$$distr^c.2.l \ = distr^c.1.[1\ 3\ 5\ 7] \natural distr^c.1.[2\ 4\ 6\ 8] \ = \ [[1\ 5]\ [2\ 6]\ [3\ 7]\ [4\ 8]]$$

Function Transformation. We have proved in the paper [15] that the functions defined on *PowerLists*, could be transformed to accept distributions. The transformation is based on the following theorem.

Theorem 2. *Given*

- *a function f defined on $PowerList.X.n$ as*

$$f.(u|v) = \Phi^f_{|}(u, v), \tag{23}$$

  where Φ is an operator defined based on scalar functions and extended operators on PowerLists.
- *a corresponding distribution $distr^l.p, (p \leq n)$, and*
- *a function f^p* – the bounded parallel function *defined as*

$$\begin{array}{l} f^p.(u|v) = \Phi^{f^p}(u, v) \\ f^p.[l] = [f^s.l] \\ f^s.u = f.u \end{array} \tag{24}$$

then the following equality is true

$$f = flat_1 \circ f^p \circ dist^l.p \tag{25}$$

A similar theorem is proven for cyclic distributions, too.

4.2. *ParList* Distributions

For *ParList* functions we may also define distributions, but they are non-homogeneous. A distribution is considered to be w-balanced, if the difference between the maximum number of elements assigned to a process and the minimum number of elements assigned to a process is less or equal to w; a distribution is called homogenous, if $w = 1$, and perfect, if $w = 0$. In the *PowerList* case, the distributions are perfect [15]. One way of defining distributions over $ParList.X.n$ is to consider a distribution function with the first argument equal to p_t, which is the number of applications of the operators ($\triangleright$ or $\triangleleft$, and $|$ or $\natural$); the second argument is the list. To define these distributions, we need two auxiliary operators: $\hookrightarrow, \hookleftarrow$.

$$\begin{array}{l} \hookrightarrow: X \times \text{ParList.(ParList.}X).n \\ x \hookrightarrow l = (x \triangleleft first.l) \triangleleft rest.l \end{array} \tag{26}$$

where $first$ extracts the first element of a list, and the result of the function $rest$ is the list without the first element. In fact, $\hookrightarrow$ concatenates the first argument to the first sublist of a list of sublists, which is the second argument.
The operator $\hookleftarrow$ is similarly defined.

Distributions over $ParLists$ may be defined based on a pair of operators $(\triangleright, |)$, $(\triangleleft, |)$,$(\triangleright, \natural)$,or $(\triangleleft, \natural)$, depending on the function definition. For example, a linear distribution is defined as:

$$\begin{array}{l} distr^l : \mathbb{N}\times\text{ParList}.X.n \rightarrow \text{ParList}.(\text{ParList}.X).p \\ distr^l.p_t.(x \triangleleft l) = x \hookrightarrow distr^l.(p_t-1).l \\ distr^l.p_t.(u|v) = distr^l.(p_t-1).u|distr^l.(pt-1).v \\ distr^l.0.l = [l] \end{array} \tag{27}$$

The first argument p_t is computed as $p_t = p_{even} + p_{odd}$, where p_{even} is the number of applications of the operator *tie* or *zip*, and p_{odd} is the number of applications of the operator $\triangleleft$ or $\triangleright$ (the method of computing p_{even} and p_{odd} can be found in [17]).

Example.

If we consider the list $l = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$, and we have $4 = 2^2 = 2^p$ processors, we need to apply a distribution with $p_t = 3 = 2 + 1 = p + p_{odd}$. The number p_{odd} is obtained by counting the bits equal to 1, on positions less or equal than p in the binary representation of n, which is the length of the list l. The binary representation of n is for this example $n = (1001)_2$.

$distr^l.3.l \ = \ [1] \hookrightarrow distr^l.2.[2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$
$distr^l.2.[2\ 3\ 4\ 5\ 6\ 7\ 8\ 9] = distr^l.1.[2\ 3\ 4\ 5]|distr^l.1.[6\ 7\ 8\ 9] = [[2\ 3]\ [4\ 5]\ [6\ 7]\ [8\ 9]]$
$\Rightarrow$
$distr^l.3.l \ = \ [[1\ 2\ 3]\ [4\ 5]\ [6\ 7]\ [8\ 9]]$

Function Transformation. The transformation of a function on $ParList$ in order to accept distributions is similar to $PowerList$ function transformation. But, in this case, we have some restrictions for functions that could be transformed, due to the fact that the application of the operator $\triangleright$ (or $\triangleleft$) is postponed until the sequential computation starts. That means that the initial order of operators' application is not preserved. So, in order to transform a $ParList$ function for bounded parallelism a commutativity property of that function has to be proved.

Definition 1. We say that a $ParList$ function f satisfies the *commutativity property* iff

$$flat^* \circ f^p.([a] \triangleleft \overline{u}) = flat^* \circ f^p(a \hookrightarrow \overline{u}) \tag{28}$$

We call this property – commutativity – because it implies that the order in which the deconstruction operators *tie* and *cons* are applied, does not change the result.

Simple examples of functions that satisfy the commutativity property are: *map* and *reduce*.

Theorem 3. *For a ParList function f and a corresponding distribution $distr.p_t$ (defined based on the same operators), if the function satisfies the commutativity property, then*

$$flat^* \circ f^p.(distr.p_t.u) = f.u \tag{29}$$

The function f^p is the *bounded parallel function*, and $flat^*$ is defined based on the same operators as those used for the distribution[17].

There is a constraint imposed by the commutativity property, but still there is a large class of functions that satisfy this constraint. In BMF (Bird-Meertens Formalism) of lists [2, 3, 18, 5] the analysis is done only for homomorphisms, which are functions on lists that can be expressed as a composition of a *reduce* function (with a certain associative operator) and a *map* function. Bounded parallelism is discussed there too, but only for concatenation operator (so linear distribution or block).The functions *map* and *reduce* satisfy the *commutativity* property, so all combinations of them satisfy it as well.

4.3. *PList* Distributions

Distributions on $PList$ data structures can be defined in a similar way to the $PowerList$ case. The difference is that we have also the argument that contains a list with arities – l. The first argument of the distribution function – p – could have values in the range $[0..length.l]$. The distribution function splits the list l into two parts: the first part defines the shape of resulted list and the second part defines the shape of each sublist element of the resulted list. Because these lists of arities are needed in the computations of the resulted lists, we will define distributions using pairs formed each one by a list of arities and a $PList$. We consider these pairs as forming a new type $Pair : PosList \times \text{PList} \rightarrow Type$. For this type, we have the following two axioms that express the relation with operators $|$ and $\natural$:

$$
\begin{array}{l}
< l, [|i : i \in \overline{n} :< [a], u_i >] >=< l \triangleright a, [|i : i \in \overline{n} : u_i] > \\
< l, [\natural i : i \in \overline{n} :< [a], u_i >] >=< l \triangleright a, [\natural i : i \in \overline{n} : u_i] >
\end{array}
$$

Based on these, the definition of the distribution is:

$$
\begin{array}{l}
distr^l : \mathbb{N} \times Pair.PosList.\text{PList} \rightarrow Pair.PosList.\text{PList} \\
distr^l.p. < l_1, < l, u >>=< l_1, distr^l.p. < l, u >> \\
< l_1, distr^l.p. < (a \triangleleft l), [|i : i \in \overline{a} : u_i] >>= \\
\qquad < l_1, [|i : i \in \overline{a} :< [a], distr^l.p{-}1. < l, u_i >>] >, \text{ if } p > 0 \\
< l_1, distr^l.0.l.u >=< l_1, [< l, u >] >
\end{array}
\tag{30}
$$

Initially the list l_1 is equal to the empty list $[]$.
The cyclic distribution is defined similarly by replacing operator *tie* with operator *zip*.

The number of resulted sublist is equal to $n_p = prod.p.l$, $prod.p.(a \triangleleft l_1) = a * prod.p - 1.l_1; prod.0.l = 1$.

Example 5.

$$
\begin{array}{l}
distr.2. < [], < [2\ 3\ 2], [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12] >>= \\
< [], distr^c.2. < [2\ 3\ 2], [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12] >>= \\
< [], (< [2], distr^c.1. < [3\ 2], [1\ 3\ 5\ 7\ 9\ 11] >> \natural
\end{array}
$$

$< [2], distr^c.1. < [3\ 2], [2\ 4\ 6\ 8\ 10\ 12] >>) >=$
$< [], (< [2], (< [3], distr^c.0. < [2], [1\ 7] >> \natural < [3], distr^c.0. < [2], [3\ 9] >>$
$\natural < [3], distr^c.0. < [2], [5\ 11] >>) > \natural$
$(< [2], (< [3], distr^c.0. < [2], [2\ 8] >> \natural < [3], distr^c.0. < [2], [4\ 10] >>$
$\natural < [3], distr^c.0. < [2], [6\ 12] >>) >) >=$
$< [], (< [2\ 3], ([< [2], [1\ 7] >] \natural [< [2], [3\ 9] >] \natural [< [2], [5\ 11] >] >) \natural$
$(< [2\ 3], ([< [2], [2\ 8] >] \natural [< [2], [4\ 10] >] \natural [< [2], [6\ 12] >]) >) >=$
$< [2\ 3], [[< [2], [1\ 7] >] [< [2], [2\ 8] >] [< [2], [3\ 9] >]$
$[< [2], [4\ 10] >] [< [2], [5\ 11] >] [< [2], [6\ 12] >]] >$

Function Transformation. As in *PowerList* case the functions are transformed for bounded parallelism using distribution. The difference is that the number of division parts is different for different steps. In order to emphasize this kind of transformation, we will present an application – Numerical Integration with Rectangle Formula.

Example 6 (Numerical Integration with Rectangle Formula). For a function $f : [a, b] \to \mathbb{R}$, the integral $I = \int_a^b f.x dx$ can be approximated using the rectangle formula as:

$$Q_{D_k}.f = \frac{1}{3} Q_{D_{k-1}}.f + h \sum_{i=1}^{2m} f.x_i,$$

where $h = \dfrac{b-a}{3^k}$, $m = 3^{k-1}$, $k = 1, 2, \ldots$ and the x_i form a division on interval $[a, b]$ with $n = 3^k$ points:

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

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

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

$$rect : Real \times PosList \times PList.Real.n \to Real$$

defined by:

$$\begin{array}{lcl} defined.rect.l.p & \equiv & prod.l = length.p = n \\ rect.[].[x] & = & h_k * x \\ rect.h_k.(3 \lhd l).[\natural i : i \in \overline{3} : p.i] & = & \frac{1}{3} rect.(3h_k).l.(p.1) + h_k * sum.(2 \rhd l).(p.0 \natural p.2), \end{array}$$

where the arguments are:

– the first $h_k = \dfrac{b-a}{3^k}$ is the division step,

– the second is a list formed by k values equal to 3, and

– the third is the $PList$ that contains the function values.

The function sum was defined in Section 3.

For bounded parallelism the parallel function $rect^p$ is defined as:

$$\begin{array}{lcl} rect^p.h_k.(3 \triangleleft l).[\natural i : i \in \overline{3} : u.i] & = & \frac{1}{3} rect^p.(3h_k).l.u.1 + h_k * sum^p.(2 \triangleright l).(u.0 \natural u.2) \\ rect^p.h_k.[].[< l, [v] >] & = & [rect.h_k.l.v] \end{array}$$

The parallel function $rect^p$ is applied to $< [], distr^c.k.l.p >=< l_k, u >$, where p is the initial $PList$, and l is the initial arities list.

4.4. *PowerArray* Distributions

This kind of distributions corresponds to Cartesian distributions [14], and they are defined by Cartesian products of one-dimensional distributions. This means that rows and columns are distributed independently.

Starting from this idea, distribution functions are defined on bidimensional *Power Array*s. The operators *tie* or *zip* may be used, on each dimension. Depending on the possible combinations we arrive to 4 types of data-distributions: *linear-linear, linear-cyclic, cyclic-linear, cyclic-cyclic*.

We consider the case of *linear-cyclic* distribution.

$$\begin{array}{l} distr^{lc}.p_0.p_1.(u|_0 v) = distr^{lc}.(p_0 - 1).p_1.u \ |_0 \ distr^{lc}.(p_0 - 1).p_1.v, \\ \ \text{if } loglen_0.(u|_0 v) \geq p_0 \wedge p_0 > 0 \\ distr^{lc}.p_0.p_1.(u \natural_1 v) = distr^{lc}.p_0.(p_1 - 1).u \ \natural_1 \ distr^{lc}.p_0.(p_1 - 1).v, \\ \ \text{if } loglen_1.(u \natural_1 v) \geq p_1 \wedge p_1 > 0 \\ distr^{lc}.0.0.l = [l] \\ distr^{lc}.p_0.p_1.l = [l], \ \text{if } loglen_0.l < p_0 \wedge loglen_1.l < p_1 \end{array} \tag{31}$$

If we apply this distribution to a structure $PowerArray.X.n_0.n_1$ with $n_0 \geq p_0$, and $n_1 \geq p_1$, then the size of the data structures that are elements in the resulted structure is equal to $2^{(n_0 - p_0)} * 2^{(n_1 - p_1)}$. We call these distributions on $PowerArray$, Cartesian distributions.

Remark. For multidimensional structures we may use $PowerArray$s or equivalent $PowerList$s with depths larger than 1. Cartesian distributions could be defined on $PowerArray$s, but not on their equivalent $PowerList$s. This is another advantage of using $PowerArray$s functions for multidimensional structures.

Function Transformation.

Theorem 4. *For a $PowerArray.X.n_0.n_1$ function f, and a corresponding distribution $distr.p_0.p_1$, the function f^p, the bounded parallel function, has the following property:*

$$f = flat \circ f^p. \circ distr.p_0.p_1 \tag{32}$$

where, the function f^p is the bounded parallel functions, and $flat$ has to be defined based on the same operators as the function f.

The proof contains a demonstration for each dimension, which are similar to that of Theorem of *PowerList* functions transformations.

Remark. Applying a distribution generates a change in the order of operators' application. For any *PowerArray* function, in order to assure the correct definition of it, it is necessary to prove first that the result of the function is independent of the order of operators applications [12]. From this, we have also the fact that the function result is not affected (the value is not changed) by the distribution.

5. Cost Measures: Time-complexity

There must be a mechanism for determining the costs of a program early in the development cycle, and in a way that does not depend critically on the target architecture. Such cost measures are the only way to decide, during development, the merits of using one algorithm rather than another.

We will evaluate for our programs time-complexities base on given processor-complexities.

Execution model. In this model a function specifies a parallel program, and in the ideal case when the number of processors is at least equal to the number of applications of the function on singletons (usually equal to the number of the list elements) this program can be computed completely in parallel; the order of the time-complexity being equal to the height of the tree associated to the divide&conquer computation. When we introduce a distribution, we specify that parallel execution is combined with sequential execution, so the resulted time-complexity is obtain as a sum of the time-complexity associated to the parallel execution and time-complexity of the sequential part: each processor will sequentially execute some computation, and then they work together in a parallel program of divide&conquer type.

PowerList. Considering a function defined on *PowerList*, and a distribution $distr.p.\cdot$, the time-complexity of the resulted program is the sum of the parallel execution time and the sequential execution time:

$$T = \Theta + T(f^p) + T(f^s) \tag{33}$$

where Θ reflects the costs specific to parallel processing (communication or access to shared memory).

The evaluation considers that the processor-complexity is 2^p ($O(2^p)$ processors are used).

Example 7 (Constant-time combination step). If the time-complexity of the combination step is a constant $T_s(\Phi) = K_c, K_c \in \mathbb{R}$, and considering the time-complexity of computing the function on singletons is equal to K_s ($K_s \in \mathbb{R}$ also a constant), then we may evaluate the total complexity as being:

$$T = \theta + K_c p\alpha + K_c(2^{n-p} - 1) + K_s 2^{n-p} \tag{34}$$

If $p = n$ we achieve the cost of the ideal case (unbounded number of processors).

For example, for *reduction* $red(\oplus)$ the time-complexity of the combination step is a constant, and $K_s = 0$; so we have

$$T_{red} = \theta + K_{\oplus}(p\alpha + 2^{n-p} - 1) \tag{35}$$

For extended operators $\odot$ the combination constant is equal to 0, but we have the time needed for the operator execution on scalars reflected in the constant K_s. A similar situation is also for the high order function *map*. In these cases the time-complexity is equal to

$$T = \theta + K_s 2^{n-p} \tag{36}$$

ParList. Since, the $ParList$ functions are transformed into $PowerList$ functions the time-complexity of the parallel computation is computed as it was analyzed for $PowerList$ case. The time-complexity for sequential computation is based on $ParList$ functions and depends on the maximal length of the sublists. We have the same general formula of time-complexity computation, but the functions f^s are $ParList$ functions, which are going to be computed sequentially. The time-complexity of the sequential part of the computation is evaluated as being the maximum of the time-complexities of each computation of the function f^s corresponding to each sublist.

The list with maximum length in the distributed list is either the first (if $\triangleright$ is used) or the last (if $\triangleleft$ is used). So, for the first case, the time-complexity formula is

$$T = \Theta + T(f^p) + T(f^s.(first.distr.p.u)) \tag{37}$$

The transformation of the $ParList$ programs into $PowerList$ programs is important because in this way sequential execution required by the operators *cons* or *snoc* is not interleaved any more with parallel execution, being postponed until the final stage.

PList. In order to evaluate the time-complexity of a program specified based on $PList$ structures we use a similar approach to that used in the case of $PowerList$ programs.

Example 8 (Constant-time combination step). If we have $PList$ functions defined on lists of type PosList with identical elements equal to a and corresponding $PList$s, and the evaluation considers that the processor-complexity is a^p ($O(a^p)$ processors are used) then we have the following evaluation:

$$T = \theta + K_c p\alpha + K_c(a^{n-p} - 1)\frac{1}{a-1} + K_s a^{n-p} \tag{38}$$

Example 9 (FFT). The time complexity of the FFT algorithm defined based on $PList$, depends on the prime factors of n and by their number k. If we consider that all the prime factors are less than a number M then the time complexity is $O(k)$, with a constant that depends on M. If, for example, $n = 3^k$ then $T.n = O(\log_3 n)$. In the sequential case the time complexity is $T^s.n = n(a_1(p_1 - 1) + \cdots + a_k(p_k - 1))$ if $n = p_1^{a_1} \ldots p_k^{a_k}$ [20].

PowerArray. The time-complexity associated to a *PowerArray* function is computed, as in the *PowerList* case, from the sum of the time-complexity associated to parallel execution $T(f^p)$, and that for sequential execution $T(f^s)$. $T(f^p)$ could be computed as a sum composed by terms that represents time-complexities corresponding to computations on each dimension.

Remark. If the total number of processors is fixed and equal to 2^p, then the analysis of time-complexity may lead to the most appropriate decomposition $p = p_0 + p_1$.

Set-Distributions. By using set-distribution, a data element is distributed to more than one process [14].

- One possibility to introduce set-distributions on these special types of data structures is to increase the dimension of the data structure by replication, and then apply a distribution on the obtained list.
- Another possibility to introduce set-distributions is to apply first a distribution, and then use a replication for the distributed data structure.

For *PowerLists*, a simple 2^p times replication on the second dimension, $(p \geq 0)$, is defined by:

$$\begin{array}{l} rep_1.p.X = rep_1.(p-1).X \mid_1 rep_1.(p-1).X \\ rep_1.0.X = X \end{array} \tag{39}$$

Replication could be also combined with different permutation functions. For example, replication with right rotation is defined by the following function:

$$\begin{array}{l} repR_1 : \mathbb{N} \times PowerArray.T.n.0 \to PowerArray.T.n.p \\ repR_1.p.X = repR_1.(p-1).(X|_1 arr.2^{n-p}.X) \\ repR_1.0.X = X \end{array}$$

$$\begin{array}{l} arr : \mathbb{N} \times PowerArray.T.n_0.n_1 \to PowerArray.T.n_0.n_1 \\ arr.(2k).(X \natural_0 Y) = arr.k.X \natural_0 arr.k.Y \\ arr.(2k+1).(X \natural_0 Y) = arr.(k+1).Y \natural_0 arr.k.X \\ arr.k.(X|_1 Y) = arr.k.X|_1\ arr.k.Y \\ arr.k.[a] = [a] \end{array} \tag{40}$$

Distribution Costs. Data-distribution implies some specific costs which depends on *the maximum number of distinct elements (θ_d)* which are assigned to a processing element. We denote this cost by T_d, and its complexity order depends on θ_d.

In *PowerList* case, this cost is equal to the length of the sublists created after the execution of the distribution function *distr.p*. If the input list has 2^n elements and we apply the distribution function with the first argument equal to $p, p \leq n$, then the distribution cost has the complexity order equal to $O(2^{n-p})$.

In the set-distribution case, where data replication is used, this kind of costs depends on more parameters. If we use a simple replication on a new dimension, and then apply a Cartesian distribution, T_d is not influenced by the distribution on the

new dimension. But, if a more complicated replication is used, T_d depends on the particular replication function.

Example 10 (Lagrange Interpolation). The evaluation of the Lagrange polynomial in a given point x is an example of application where the set-distributions could be successfully used. The input lists are: $X = [x_0, \ldots, x_{m-1}]$ distinct points; $F = [f.x_0, \ldots, f.x_{m-1}]$ function values. We will not present the entire derivation of the program, but only the parts that emphasize the importance of choosing the right distribution. (More details could be found in [16].) We have:

$$lagrange.X.F.x = reduce(+).(L.X.x * F)$$
$$L.X.x = U1.X.x/U2.X$$
$$U1.X.x =< reduce(*).(< x\text{–}> .X) \ / \ > .(< x\text{–}> .X)$$

Function $U2$ computes the list of products formed by $(x_i - x_0)\ldots(x_i - x_{i-1})(x_i - x_{i+1})\ldots(x_i - x_{m-1})$, for any $i : 0 \leq i < m$. Since in these calculations an element appears in computations more than once, the use of data replication is appropriate.

First, we consider a replication that is combined with a rotation function $repR_1.n.X$ (Equation 40). In this way, each column contains different elements, which means that each column contains the elements of list X in a different order.

$$U2.X = UA.(repR_1.n.X)$$
$$UA.A = reduce_1(*).(dif.(first.A).A)$$
$$dif.(x|_0 y).(X|_0 Y) \ = \ dif.x.X \ |_0 \ dif.y.Y$$
$$dif.x.(X|_1 Y) \ = \ dif.x.X \ |_1 \ dif.x.Y$$
$$dif.[x].[y] = \begin{cases} x - y, & \text{if } x \neq y \\ 1, & \text{otherwise} \end{cases}$$

The function UA is the most costing one, and we will refer to it in more detail.
Distribution: We define and use a Cartesian distribution $distr^{ll}.p_0.p_1$, where $p_0, p_1 < n$ and $p_0 + p_1 = p$. This means that $2^{p_0} * 2^{p_1} = 2^p$ processors are available for working in parallel. The distribution $distr.p_0.p_1$ has to be *linear-linear*, since the function is defined based on $|_0$ and $|_1$ operators. In order to accept the distribution, the function UA is transformed:

$$U2.X = UA^p.(distr^{ll}.p_0.p_1.(repR_1.n.X))$$
$$UA^p.A = reduce_1^p(*).(dif^p.(first.A).A)$$

Time-complexity is $O((p_1 * 2^{n-p_0})\alpha + 2^{2n-p})$ but the ***distribution costs*** have to be analyzed, as well. Each processor contains a matrix with $2^{(n-p_0)+(n-p_1)}$ elements, but not all of them are distinct. In order to reduce the cost of data distribution, the number of distinct elements on each processor has to be as small as possible; the minimum is equal to $2^n/2^{p_0}$. We have $\theta_d(distr^{ll}.p_0.p_1.A) = min\{2^n, 2^{n-p_0} + 2^{m-p_1} - 1\}$. But, this cost is very high, so we have to change the replication method. We can start from a distribution (one-dimensional $distr.p_0.X$) of the list X, then to replicate this distributed list based on $repR_1.p_0.\cdot$, and then to use replication based on $repR_1.(n - p_0).\cdot$ for each resulted sublist. After we apply the function $flat$ we

obtain a matrix B that has the property that each column is a different permutation of the list X, and we may apply the same function UA to compute the products $u.x_i$.

If we apply a Cartesian distribution $distr^{ll}.p_1.p_0$, each resulted submatrix will have only 2^{n-p_0} distinct elements, if $p_1 \geq p_0$, and $2^{n-p_0} \times 2^{p_0-p_1}$ distinct elements if $p_1 < p_0$. The analysis of the time-complexity and the distribution cost leads to the conclusion that the best decomposition of p is $p = p_0 + p_1 = 2p_0, p_0 = p_1$, provided that p is even.

6. Efficient implementation

Mappings on hypercubes have been analyzed by J. Misra [12] and J. Kornerup [9] for the programs specified based on *PowerList* notations; they are based on Gray code. Each fundamental operator on *PowerList*s can be efficiently implemented on a hypercube. The technique encodes the *PowerList*s with a reflected Gray code, and this encoding can be viewed as a domain transformation. Algebraically, it is an isomorphism between the algebra of *PowerList*s and the algebra of Gray coded *PowerList*s.

The analysis assumes that the hypercube has a number of nodes at least equal to the number of lists' elements which are mapped onto, and this could be an unrealistic assumption. But, since we have proved that the functions over these structures can be easily transformed using distributions this problem is solved.

Also, K. Achatz and W. Schulte have presented in [1], a methodology for deriving explicit programs for massively data parallelism from specifications based on *PowerList*s. They present a set of semantic preserving transformation rules, which make the implicit data parallelism in a divide&conquer scheme over *PowerList*s explicit, by introducing topology independent communication operations on *PowerList*s.

Functions over *ParList*s are transformed using distributions into functions over *PowerList*s, and so, the mappings used for *PowerList*s can be used, too.

Example 11 (FFT). We can implement the algorithm of FFT defined in Section 3, based on *PList* data structures, using a *recursive* interconnection network [10], which have the same arity list of the nodes like the arity list used for the calculation of fft. The implementation has two stages: a descendent stage and an ascendant stage.

J. Kornerup showed in [10] that the recursive network is isomorphic with *butterfly* and *iterative* networks, and thus networks can also be used.

7. Parallel Programming Paradigms

Obviously, the presented model is very appropriate for shapely programs based on divide&conquer paradigm, but not only. The division partition is controlled in order to obtain a good work-balance, and this is done in formalized way (based on defined algebras). Division could be done in two equal parts (*PowerList*, *PowerArray*), different number of equal parts (*PList*, *PArray*), or almost equal parts (*ParList*,

ParArray). Still, other paradigms, such as direct parallelization ("embarrassingly parallel computations", SPMD, ParFor) could also be expressed in this model. *PList* data structure together with the algebra defined on it allows different kind of computations. For example, a computation of "parFor" type (n tasks executed in parallel) is defined using a *PList* function with a list of arities formed of only one element $[n]$.

In order to allow sequential computation to be also expressed in this model, we may combine classical functional programming with the model described before. This means that we have programs expressed as a composition of functions, and these functions could be defined on structures of PARES types based on structural induction principles, or they could be functions defined on simple lists (based only on the operators *cons* and *snoc*, and their correctness being proved based on simple induction principles).

The parallel programming paradigm SEQ-PAR defines programs as a composition of parallel functions. So, the sequentiality is given by the sequence of functions, and so it is possible in this way to impose an order between computations, too.

Data-Pipeline computation could also be described. A pipeline consists of a list of stages, where each stage applies a different function to the results obtained in the previous stage. We can use a PARES function: *distributed map*, with two arguments: a list of functions and a list of data-inputs, which applies a different function to each data-input. Together with some kinds of shifting functions we can obtain a pipeline type program:

$$
\begin{aligned}
&init\_pipe : List.X \times List.X \to List.X\\
&init\_pipe.(u \triangleright a).(l \triangleright b) =\\
&\qquad\qquad init\_pipe.u.(dmap.f.(a \triangleleft l)))\\
&init\_pipe.[].l = l
\end{aligned}
$$

$$
\begin{aligned}
&pipe : List.X \times List.X \times List.X \to List.X\\
&pipe.(p \triangleright a).(l \triangleright b).q = pipe.p.(dmap.f.(a \triangleleft l)).(b \triangleleft q)\\
&pipe.[].(l \triangleright b).q = pipe.[].(dmap.f.(0_f \triangleleft l)).(b \triangleleft q)\\
&pipe.[].[].q = q
\end{aligned}
$$

(p=input-data, q=output-data)
(0_f is a value that will be finally ignored)

The function *dmap* could be defined as a *PowerList* function, or more general as a *PList* function:

$$
\begin{aligned}
&dmap : PowerList.F.n \times PowerList.X.n \to\\
&\qquad\qquad PowerList.X.n\\
&dmap.(f \mid g).(p \mid q) = dmap.f.p \mid dmap.g.q\\
&dmap.f.[a] = [f.a]
\end{aligned}
$$

$$
\begin{aligned}
&dmap : PosList \times PList.F.n \times PList.X.n \to PList.X.n\\
&dmap.(n \triangleleft l).[i : i \in \overline{n} : f_i].[i : i \in \overline{n} : p_i] =\\
&\qquad\qquad [i : i \in \overline{n} : dmap.l.f_i.p_i]\\
&dmap.[].f.[a] = [f.a]
\end{aligned}
$$

where F is the type of all functions $f : X \to X$, and X is a general type.

A very simple example of direct parallelization is that used for solving a linear system using Jacobi iterative method.

Example 12 (Linear System Solving – Jacobi Relaxation). The method repeats iteratively a computation that has as a result an approximation of the solution, and if this kind of computation is repeated enough time (until the difference between two successive solutions is less than a pre-established error) we obtain a solution. Each value X_i can be computed independently (they depend only on the previous approximation) and so the parallelization is very easy. A step executed by this method is described by the JacobiStep algorithm.

Algorithm JacobiStep$< n, A[0..n-1][0..n-1], B[0..n-1], X[0..n-1] >$
for $i = 0, n-1$ **in parallel do**
  $summ[i] \leftarrow 0$;
  for $j \leftarrow 0, n-1$ **do**
    if $(i \neq j)$ **then**
      $summ[i] \leftarrow summ[i] + A[i][j] * X[j]$;
    end if
  end for
  $summ[i] \leftarrow (B[i] - summ[i])/A[i][i]$;
  $X[i] \leftarrow summ[i]$;
end for

The description of this computation in the presented model is as follows:

$JacobiStep : PosList \times \text{PList.(ParList.}\mathbb{R}.n).n \times \text{ParList.}\mathbb{R}.n \times \text{ParList.}\mathbb{R}.n$
$\rightarrow \text{ParList.}\mathbb{R}.n$
$defined.JacobiStep.l.A.B.X \equiv l = [n]$
$JacobiStep.[n].A.X.B = [i : i \in \overline{n} : (B - f.A[i].B.X.i) \,/A[i]]$

{ $-$ and $/$ are extended operators }
$f.V.B.X.i = reduce(+).prod.V.X.(naturals.n).i$

$$\begin{cases} prod.(V1|V2).(X1|X2).(N1|N2).i = prod.V1.X1.N1.i \ * \ prod.V2.X2.N2.i \\ prod.(v \lhd V').(x \lhd X').(m \lhd N').i = \begin{cases} v * x \ \lhd \ prod.V'.X'.N'.i, & \text{if } i \neq m \\ 0 \ \lhd \ prod.V'.X'.N'.i, & \text{if } i = m \end{cases} \\ prod.v.x.m.i = \begin{cases} v * x, & \text{if } i \neq m \\ 0, & \text{if } i = m \end{cases} \end{cases}$$

$$naturals.n = prefix.(rep.[1].n), \quad \begin{cases} rep.l.n = \begin{cases} rep.l.\frac{n}{2} \mid rep.l.\frac{n}{2}, & \text{if } n \text{ is even} \\ l \lhd rep.l.n-1, & \text{if } n \text{ is odd} \end{cases} \\ rep.l.1 = l \end{cases}$$

The function $naturals.n$ returns the first natural numbers less than n, and uses the prefix sum function (Example 1). The function $rep.l.n$ returns a list that contains a list with n elements equal to l. Since the computation is dependent on the indices values, and the PARES data structures do not accept directly referring a particular

element, then a list of indices were introduced as an argument. The solution is not very elegant, but emphasizes the fact that the value-dependent computations could be described, too.

8. Conclusions

We have considered the theories: $PowerList$, $ParList$, $PList$, $PowerArray$, $ParArray$, $PArray$, together, in order to prove that they can form a good model for parallel programming with a very high level of abstraction – PARES (Parallel Recursive Structures). Based on the fact that an effective model has to fulfill the following requirements – abstractness, software development methodology, architecture independence, cost measures, no preferred scale of granularity, efficiently implementable – we have analyzed these requirements for this model. The conclusion is positive, and this was also emphasized by the presented examples. The communication, synchronization and mapping of these programs are implicit, and so the abstractness is very high and, they are architecture independent. The correctness of the programs – very important in parallel programming setting – is proved based on the algebras defined on these structures, and also based on structural inductions principles. Data-distributions were introduces as functions, and they allow different scale of granularity for the parallel programs defined on these structures. Cost measures were defined, and they include analyses for different granularities, based on data-distributions. Implementations could be done efficiently on hypercubes or on recursively defined interconnection networks; implementation can be also based on a set of semantic preserving transformation rules, which make the implicit data parallelism in a divide&conquer scheme over $PowerList$s explicit.

The model could be framed as a recursive model for data-parallelism. Divide& Conquer paradigm is naturally described, and the most important advantage is that the division stage is formally controlled and leads from the beginning to a good work-balance; data-distributions preserve this advantage because they were introduced as functions over PARES structures, too. But also, as we have shown, other kinds of parallel computations can be efficiently described and developed with this model.

References

[1] Achatz K., Schulte W., *Massive parallelization of divide-and-conquer algorithms over powerlists*, Science of Computer Programming, 1996.

[2] Bird R., *Lectures on Constructive Functional Programming*, in M. Broy editor, *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences, Vol. **55**, pp. 151–216. Springer-Verlag, 1988.

[3] Cole M., *Parallel Programming with List Homomorphisms*, Parallel Processing Letters, **5**(2):191–204, 1994.

[4] Coman G., *Numerical Analysis*, Editura Libris, 1995 (in Romanian).

[5] Gorlatch S., *Abstraction and Performance in the Design of Parallel Programs*, CMPP'98 First International Workshop on Constructive Methods for Parallel Programming, 1998.

[6] Jay C.B., *A semantics for shape*, Science of Computer Programming, Vol. **25**, No. 2, December 1995, pp. 251–283(33).

[7] Jay C.B., *Costing Parallel Programs as a Function of Shapes*, Science of Computer Programming, Computer Programming, Vol. **37**, No. 1, May 2000, pp. 207–224.

[8] Kornerup J., *Mapping a functional notation for parallel programs onto hypercubes*, Information Processing Letters, Vol. **53**, 1995.

[9] Kornerup J., *Data Structures for Parallel Recursion*, PhD Thesis, Univ. of Texas, 1997.

[10] Kornerup J., *PLists: Taking PowerLists Beyond Base Two*, in *Proceedings of CMPP'98 First International Workshop on Constructive Methods for Parallel Programming*, 1998.

[11] Ladner R.E., Fischer M. J., *Parallel Prefix Computation*, Journal of the ACM (JACM) Vol. **27**, Issue 4, pp. 831–838, 1980.

[12] Misra J., *PowerList: A structure for parallel recursion*, ACM Transactions on Programming Languages and Systems, Vol. **16**, No. 6, pp. 1737–1767, 1994.

[13] Niculescu V., *Parallel Algorithms for Fast Fourier Transformation using PowerList, ParList and PList Theories*, Lecture Notes in Computer Science, *Proceedings of International Conference EuroPar'2002*, Paderborn, Germany, August 2002, Springer-Verlag, pp. 400–403.

[14] Niculescu V., *On Data Distributions in the Construction of Parallel Programs*, The Journal of Supercomputing, Kluwer Academic Publishers, **29**(1): 5–25, 2004.

[15] Niculescu V., *Data Distributions in PowerList Theory*. Lecture Notes of Computer Science, Vol. **3722**: Theoretical Aspects of Computing, *Proceedings of ICTAC 2007*, Springer-Verlag, 2007: 396–409.

[16] Niculescu V., Guran A., *Efficient Recursive Parallel Programs for Polynomial Interpolation*, *Post-proceedings of KEPT 2009*, International Conference, Babes-Bolyai University Press, pp. 265–274.

[17] Niculescu V., Guran A., *Bounded Parallelism in PowerList and ParList Theories*, SYNASC 2009, *Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timişoara, 2009, IEEE Society Press, pp. 237–244.

[18] Skillikorn D.B., *Structuring data parallelism using categorical data types*, in *Programming Models for Massively Parallel Computers*, pp. 110–115, 1993, Computer Society Press.

[19] Skillicorn D.B., Talia D., *Models and Languages for Parallel Computation*. ACM Computer surveys, **30**(2): 123–136, June 1998.

[20] Wilf H.S., *Algorithms and Complexity*, Mason & Prentice Hall, 1985.