Bounded Parallelism in PowerList and ParList Theories

Virginia Niculescu and Adriana Guran
Faculty of Mathematics and Computer Science
Babes-Bolyai University
Cluj-Napoca, Romania
Email: \{vniculescu, adriana\}@cs.ubbcluj.ro

Abstract—A very efficient model for recursive, data-parallel programs can be one based on PowerList, PowerArray, and ParList theories. It assures simple and correct design of this kind of programs, allowing work at a high level of abstraction. This high level of abstraction could be reconciled with performance by introducing data-distributions into these theories. In this paper, we generalize the data distributions defined on PowerLists by introducing data distributions for parallel programs defined using ParList structures. Using these distributions we also define a possibility to transform ParList parallel programs into PowerList parallel programs, which are more efficient. This is an important advantage since PowerList programs could be efficiently mapped on real architecture (e.g. hypercubes).

Keywords—parallel computation; abstraction; model; distribution; data-structures;

I. INTRODUCTION

PowerList, PowerArray and ParList are data structures introduced by Misra [7] and Kornerup [6], which can be successfully used in a simple functional description of parallel programs, that are divide and conquer in nature. They allow working at a high level of abstraction, especially because the index notations are not used. PowerArray structures are extensions of PowerLists on multiple dimensions, and ParLists are lists data structures with lengths not limited to powers of two. To assure methods that verify the correctness of 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 on these structures. The parallelism is implicit: each application of a deconstruction operator (zip or tie) means that we achieve two processes (programs) that could run in parallel. So, we obtain a tree decomposition, which is specific to divide&conquer programs. By introducing the distributions, we transform the input list into a list of sublists, and we may also consider that we specify a level where the computation done in parallel is completed. Down from that level the decompositions lead to sequential computations. By formally introduced distributions we assure that the correctness is preserved.

Also, after introducing the distributions, realistic cost measures can be more rigorously defined. The analysis of the time-complexity is done on the hypothesis of a PRAM model (shared memory), or a complete interconnection network (distributed memory).

We have analyzed the advantages of formally introduced distributions in PowerList theory [9], and in PowerArray theory [10]. They are mainly related to cost measures and mapping on real architectures.

In this paper, we extend that work by introducing data distributions for parallel programs defined using ParList structures. The advantage is that we are no more restricted to lists with lengths which are powers of two, so the degree of generalization is increased. Also, using these distributions we provide a possibility to transform ParList parallel programs into PowerList parallel programs. This is very important since PowerList programs could be efficiently mapped on real architectures (e.g. hypercubes).

With this extension we enlarge the base for a model of parallel computation with a very high level of abstraction. In order to be useful, a model of parallel computation 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 [13]. The first three requirements are evidently fulfilled by PowerList, PowerArray, and ParList theories. In this kind of models the parallelism is implicit, and hence the decomposition, communication, synchronization and mapping are implicit (if the model classification made by Skillicorn and Talia in [13] is considered.). So, without distributions, unbounded parallelism (the number of processes is not limited) was analyzed using these structures. Still, the most practical approach of bounded parallelism can be introduced, too.

Mappings on hypercubes have been analyzed for
the programs specified based on PowerList notations [7], [6]; they are based on Gray code. So, we may agree that the requirement of efficient implementation is fulfilled, too.

In other models, these kinds of enhancement have been analyzed, too. In BMF formalism [1], [11], [3] 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 [2] have been introduced to bridge the gap. Shape theory [4] is a more general approach. Knowledge of the shapes of the data structures is used by many cost models [5]. PowerList, ParList and PowerArray theories allow us to define shapely programs, but in a very elegant and simple way.

II. POWERLIST AND PARLIST THEORIES [7]

**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:

\[ \text{PowerList} : \text{Type} \times \mathbb{N} \rightarrow \text{Type} \]

and so, a PowerList \( t \) with \( 2^n \) elements of type \( X \) is specified by \( \text{PowerList}.X.n \) (where \( n = \log_{\text{len}}.t \), and the real length of \( t \) is \( 2^n \)). A PowerList with a single element \( a \) is called a singleton, and is denoted by \( [a] \). If two PowerList structures have the same length and elements of the same type, they are called similar.

Two similar PowerLists can be combined into a PowerList data structure with double length, in two different ways:

- using the operator \( \text{tie} \ p \ | \ q \); the result contains elements from \( p \) followed by elements from \( q \);
- using the operator \( \text{zip} \ p \ | \ q \); the result contains elements from \( p \) and \( q \), alternatively taken.

Therefore, the constructor operators for PowerList are:

\[
[\cdot] : X \rightarrow \text{PowerList}.X.0 \\
[. \cdot] : \text{PowerList}.X.n \times \text{PowerList}.X.n \rightarrow \text{PowerList}.X.(n + 1) \\
[\cdot \cdot] : \text{PowerList}.X.n \times \text{PowerList}.X.n \rightarrow \text{PowerList}.X.(n + 1) \\
\]

PowerList algebra is defined by these operators and by axioms that assure the existence of unique decomposition of a PowerList, using one of \( \text{tie} \) or \( \text{zip} \) operator; and the fact that \( \text{tie} \) and \( \text{zip} \) operators commute.

An induction principle is defined on PowerList data structures, which allows function definitions, and the proving of PowerList properties.

For example, the high order function \( \text{map} \), which applies a scalar function to each element of a PowerList is defined as follows:

\[
\text{map} : (X \rightarrow Z) \times \text{PowerList}.X.n \rightarrow \text{PowerList}.Z.n \\
\text{map}.f.[a] = [f.a] \\
\text{map}.f.(p \ | \ q) = \text{map}.f.p \ | \ \text{map}.f.q
\]

Other examples are the functions \( \text{flat}^* \) that are applied to PowerLists with elements which are in turn PowerLists, and return simple PowerLists. There are two functions of this type depending on the used operator: \( \text{tie} \) or \( \text{zip} \).

\[
\text{flat}^1 : \text{PowerList}.(\text{PowerList}.X.n).m \rightarrow \text{PowerList}.X.(n + m) \\
\text{flat}^1.[l] = l \\
\text{flat}^1.(p \ | \ q) = \text{flat}^1.p \ | \ \text{flat}^1.q \\
\text{flat}^1 : \text{PowerList}.(\text{PowerList}.X.n).m \rightarrow \text{PowerList}.X.(n + m) \\
\text{flat}^1.[l] = l \\
\text{flat}^1.(p \ | \ q) = \text{flat}^1.p \ | \ \text{flat}^1.q
\]

We denote by \( \pi \) the list obtained from the list \( a \) where each element was transformed into a singleton list.

Associative operators on scalar types can be extended to PowerList, too.

**ParList**

The ParList data structure is similar to PowerList, with the difference that the number of the elements is not a power of two.

The type constructor for ParList is:

\[ \text{ParList} : \text{Type} \times \mathbb{N}^* \rightarrow \text{Type} \]

and a ParList with \( n \) elements of type \( X \) is specified by \( \text{ParList}.X.n \).

It is necessary to use two other operators: \( \text{cons}(\cdot) \) and \( \text{snoc}(\cdot) \); they allow to add an element to a ParList at the beginning or at the end of the ParList.

The constructor operators are:

\[
[\cdot] : X \rightarrow \text{ParList}.X.1 \\
(. \cdot) : X \times \text{ParList}.X.n \rightarrow \text{ParList}.X.(n + 1) \\
. \cdot : \text{ParList}.X.n \times X \rightarrow \text{ParList}.X.(n + 1) \\
[\cdot \cdot] : \text{ParList}.X.n \times \text{ParList}.X.n \rightarrow \text{ParList}.X.(2n) \\
[\cdot \cdot] : \text{ParList}.X.n \times \text{ParList}.X.n \rightarrow \text{ParList}.X.(2n)
\]

Axioms of ParList algebra express, like those from PowerList algebra, the existence of a unique decomposition of ParList, using constructor operators, the commutativity of \( \text{tie} \) and \( \text{zip} \) operators, and some axioms that make connection among operators [6].
An induction principle is also defined for ParList. But, in this case, a proof has three stages: the base case, the odd inductive step and the even inductive step. The rule of structural decomposition for a ParList data structure is as follows: when the number of its elements is even the decomposition uses either tie or zip, and when this number is odd the decomposition uses either cons or snoc. This way, the decomposition is unique.

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

For example, the map function

$$\text{map} : (X \to Y) \times \text{ParList.X.n} \to \text{ParList.Y.n}$$

is defined by:

- $$\text{map.f.[a]} = [f.a]$$
- $$\text{map.f.(p.q)} = \text{map.f.p} \mid \text{map.f.q}$$
- $$\text{map.f.(a \triangleright q)} = f.a \triangleright \text{map.f.q}$$

where the first argument is a scalar function on X type.

Operators on type X can be extended over ParList.X in the following way. Let $$\otimes$$ be a binary associative operator on type X, $$\otimes : X \times X \to X$$.

The extended operator:

$$\otimes : \text{ParList.X.n} \times \text{ParList.X.n} \to \text{ParList.X.n}$$

is defined by:

- $$[a] \otimes [b] = [a \otimes b]$$
- $$(p \mid q) \otimes (u \mid v) = (p \otimes u) \mid (q \otimes v)$$
- $$(a \triangleright p) \otimes (b \triangleright q) = (a \otimes b) \triangleright (p \otimes q).$$

### III. DISTRIBUTIONS

The ideal method to implement parallel programs described with PowerLists is to consider that any application of the operators tie or zip as destructors, 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.

The distribution will transform the list into a list with $$n_p$$ elements, which are in turn sublists; each sublist is considered to be assigned to a process.

#### A. PowerList Distributions

The PowerList distributions were introduced and analyzed in [9]. In order to compare with the ParList case, we present a short review of the main results.

1) **Definition:** 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 $$n_p = 2^p \cdot (p \leq n)$$.

Two types of distributions – linear and cyclic, which are well-known distributions, may be considered. These two types 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**
  
  $$\text{distr}^l.p.[a|v] = \text{distr}^l.(p-1).u|\text{distr}^l.(p-1).v,$$
  
  if loglen.($u|v) \geq p \land p > 0$
  
  $$\text{distr}^l.0.l = [l]$$
  
  $$\text{distr}^l.p.x = [x], \text{if loglen}.x < p.$$

- **cyclic**
  
  $$\text{distr}^c.p.(u|v)^c = \text{distr}^c.(p-1).u^c|\text{distr}^c.(p-1).v^c,$$
  
  if loglen.(u|v) \geq p \land p > 0
  
  $$\text{distr}^c.0.l = [l]$$
  
  $$\text{distr}^c.p.x = [x], \text{if loglen}.x < p.$$

The base cases transform a list l into a singleton [l], having the list l as its unique element.

**Example 1:** 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:

- $$\text{distr}^l.2.l = \text{distr}^l.1.[1 2 3 4] \mid \text{distr}^l.1.[5 6 7 8] = [1 2 \mid 3 4 \mid 5 6 \mid 7 8]$$
- $$\text{distr}^c.2.l = \text{distr}^c.1.[1 3 4 7] \rightimes \text{distr}^c.1.[2 4 6 8] = [1 3 \mid 2 6 \mid 3 7 \mid 4 8]$$

2) **Properties:**

- If $$u \in \text{PowerList.X.n}, \text{then distr}^c.n.u = \pi.$$
- The result of the application of a distribution distr^c.p on a list $$l \in \text{PowerList.X.n}, \text{if p is}$$ a list that has $$2^n$$ elements each of these being a list with $$2^{n-p}$$ elements of type X.
- $$\text{flat}^* \circ \text{distr}^c.p.l = l \ast \text{could be | or \circ}.$$  
- Also, we have the trivial property distr.0.u = [u].

The properties are true for both linear and cyclic distributions.

3) **Function Transformation:** We have proved in [9] that the functions defined on PowerLists, could be easily transformed to accept distributions. The transformation is based on the following theorem.
Theorem 1: Given

- a function \( f \) defined on \( \text{PowerList}.X.n \) as
  \[
  f.(u \ast v) = \Phi^t(u, v),
  \]
  where \( \Phi \) is an operator defined based on scalar functions and extended operators on PowerLists, and \( \ast \) is \( | \) or \( \triangleright \).
- a corresponding distribution \( \text{distr}^+, p. (p \leq n) \), and
- a function \( f^p \) defined as
  \[
  f^p.(u \ast v) = \Phi^t(u, v) \\
  f^p.[i] = [f^p.i] \\
  f^p.u = f.u
  \]
then the following equality is true
  \[
  f = \text{flat}^+ \circ f^p \circ \text{distr}^+. p
  \]

4) Time Complexity: In order to evaluate the execution cost of PowerList program, we need an execution model for this type of programs. A very simple and ideal model assumes that for each computation of a function a new computational process is created. So, the time-complexity is proportional with the height of the processes’ tree (\text{loglen} of the initial input list).

If we consider a function defined on PowerLists, and we use a distribution \( \text{distr}^+. p. \), the time-complexity of the resulted program is the sum of the parallel execution time (the execution of the function \( f^p \)) and the sequential execution time (the execution of the functions \( f^p \)):

\[
T = \Theta + T(f^p) + T(f^p)
\]

where \( \Theta \) reflects the costs specific to parallel processing (process creation, communication or access to shared memory).

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

Example 2: (Constant-time combination step) If the time-complexity of the combination step is a constant \( T_c(\Phi) = K_\Phi, K_\Phi \in \mathbb{R} \), 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:

\[
T = \theta + K_p \alpha a + K_s(2^{n-p} - 1) + K_s 2^{n-p}
\]

(The constant \( \alpha \) reflects the communication or accessing memory costs.)

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

For example, for \text{reduction} \( \text{red}(\triangleright) \) the time-complexity of the combination step is a constant, and \( K_s = 0 \); so we have

\[
T_{\text{red}} = \theta + K_\triangleright (p \alpha + 2^{n-p} - 1)
\]

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 \( K_\odot \). A similar situation occurs for the high order function \( \text{map} \). In these cases the time-complexity is equal to

\[
T = \theta + K_s 2^{n-p}
\]

B. \text{ParList} Distributions

For \text{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 \( w = 0 \) [8]. In the \text{PowerList} case, the distributions are perfect.

1) Definition: One way of defining distributions over \text{ParList}.X.n is to consider a distribution function with two arguments: the first argument is equal to \( p_i \) – the number of applications of the operators \( \circ \) or \( \triangleright \), and the second argument is the list. To define these distributions, we need some auxiliary operators: \( \leftarrow, \leftarrow \).

\[
x \leftarrow l = (x \triangleright \text{first}.l) \triangleright \text{rest}.l
\]

where the function \( \text{first} \) extracts the first element of a list, and the result of the function \( \text{rest} \) is the list without the first element. In fact, \( \leftarrow \) concatenates the first argument to the first sublist of a list of sublists, which is the second argument.

The operator \( \leftarrow \) is similarly defined, but based on \( \circ \) and the function \( \text{last} \).

Distributions over \text{ParList}s may be defined based on a pair of operators \((\triangleright, \leftarrow)\), \((\triangleright, \circ)\), \((\circ, \triangleright)\), or \((\circ, \circ)\), depending on the function definition. For example, a linear distribution is defined as:

\[
\begin{align*}
\text{distr}^{\triangleright} : \mathbb{N} \times \text{ParList}.X.n & \rightarrow \text{ParList}.(\text{ParList}.X).p \\
\text{distr}^{\triangleright}.p_i.(x \triangleright l) & = x \leftarrow \text{distr}^{\triangleright}.(p_i-1).l \\
\text{distr}^{\triangleright}.p_i.(u|v) & = \text{distr}^{\triangleright}.(p_i-1).u|\text{distr}^{\triangleright}.(p_i-1).v \\
\text{distr}^{\triangleright}.0.l & = l
\end{align*}
\]

2) Method for \( p_i \) computation: We need to specify a method for computing \( p_i \), the number of needed applications of the operators \((\triangleright, \leftarrow), (\triangleright, \circ)\), or \((\circ, \triangleright)\) when we know the length of list, \( n \), and the number of available processors, \( n_p = 2^p \). Our intention is to obtain a list (the distributed list) which is a \text{PowerList} having the length equal to \( n_p = 2^p \), and its elements are lists which are of the \text{ParList} type.

The number \( p_i \) is computed as the sum \( p_i = p_\text{even} + p_\text{odd} \), where \( p_\text{even} \) is the number of applications
of the operator tie or zip, and \( p_{\text{odd}} \) is the number of applications of the operator \(<\) or \(>\). In the binary representation of the length of the list \( n \), an application of the operator \(|\) or \(\_\) corresponds to a last bit equal to 0, and after the application, this bit is removed; the application of the operator \(\oplus\) or \(\odot\) corresponds to a last bit equal to 1, and after the application, this bit is replaced by a 0 bit. Starting from \( p_{\text{even}} = p \), and \( n \), we can compute \( p_{\text{odd}} \) by counting the non-zero bits on positions less or equal than \( p_{\text{even}} \) in the binary representation of \( n \). Since we know \( p_{\text{even}} \) and we can compute \( p_{\text{odd}} \), the \( p_t \) can also be computed.

Example 3: 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 \( p_t = 3 = 2 + 1 = p + p_{\text{odd}} \). The number \( p_{\text{odd}} \) 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 \( n = (1001)_{2} \).

\[
\text{distr}^l.3.1 = [1] \leftarrow \text{distr}^l.2.[2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9] \\
\text{distr}^l.2.[2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9] = \\
\text{distr}^l.1.[2 \ 3 \ 4 \ 5] \ | \ \text{distr}^l.1.[6 \ 7 \ 8 \ 9] \\
\text{distr}^l.3.1 = [[1 \ 2 \ 3] [4 \ 5] [6 \ 7] [8 \ 9]]
\]

One way to balance the distributions for ParList is to alternatively use the operators \(<\) and \(\oplus\) for the odd cases.

3) Properties: The following properties are true for ParList distributions:

1) number of processes and the length of the resulted sublists:
   For a list \( u \) with \( \text{length.}u = n \) the result of distribution application \( \text{distr}^r.|.u \) where \( p_t = p_{\text{even}} + p_{\text{odd}} \) \( (p_{\text{even}} \leq k = \lfloor \log_2 n \rfloor) \) is a list with \( 2^{p_{\text{even}}} \) elements, and the maximal length of its elements (sublists) is equal to \( \text{max length} = 2^{k-p_{\text{even}}}+p_{\text{odd}} \). If the distribution is defined such that the \( \text{cons} \) and \( \text{snoe} \) operators are alternatively used, \( \text{max length} = 2^{k-p_{\text{even}}}+p_{\text{odd}}/2 \).

2) maximal distribution: for \( p_{\text{even}} = \lfloor \log_2 (\text{length.}u) \rfloor \) and \( m > \lfloor \log_2 (\text{length.}u) \rfloor + p_{\text{odd}} = \text{max.}u \), we have \( \text{distr}^r.m.u = \text{distr}^r.\text{max.}u.u \)

3) composition:
   Direct composition is not possible because the result of a distribution application is a PowerList of lists, but we have the following equality:
   \[
   (\text{flat}^* \circ \text{map}(\text{distr}^r.|.p)) \circ \text{distr}^r.p_u.l = \text{distr}^r.|.p_u+l
   \]
   (12)
   The function \( \text{flat}^* \) has to be defined based on the same operators as those used for the distribution.

4) relation with \( \leftarrow \) operator:
   \[
   a \leftarrow \text{distr}^r.m.u \ | \ \text{distr}^r.m.v = \\
   a \leftarrow \text{distr}^r.m+l.(u \ | \ v)
   \]
   (13)
   If \( u \) is an odd length list then \( \text{distr.1.u} = \text{distr.0.u} \), but since the value of \( p \) is calculated based on \( p = p_{\text{odd}} + p_{\text{even}} \) there is no case in which we apply \( \text{distr}^r.\).1.u to an odd length list.
   These properties are easily proved by induction.

4) Function Transformation: To transform a function on ParLists in order to accept distributions, we use the same approach as for PowerLists. But, in this case, we have some restrictions for functions that could be transformed, due to the fact that the application of the operator \( \oplus \) or \(\odot\) is postponed until the sequential computation starts. This means that the initial order of operators’ application is not preserved. So, in order to transform a ParList function for bounded parallelism it is necessary to prove a commutativity property of that function.

Definition 1: Consider a ParList function \( f \) with the following definition
   \[
   f.[a] = e \\
f.(p|q) = \Phi f.[p, q] \\
f.(a \oplus l) = \Phi f.[a, l]
   \]
   (14)
   where \( \Phi f.[ \) and \( \Phi f.\) are operators that may contain extended scalar functions and operators, and applications of function \( f \). We call the transformed function \( f^p \) which is defined as:
   \[
   f^p.(u|v) = \Phi f^p.(u, v) \\
f^p.(a \odot l) = \Phi f^p.(a, u) \\
f^p.(l) = [f.l]
   \]
   (15)
   bounded parallel function.
   Definition can be extended for other combination of operators: \(<\> and \(<\), or \(\oplus\) and \(\odot\), or \(\_\) and \(<\).

Definition 2: We say that a ParList function \( f \) satisfies the commutativity property iff
   \[
   \text{flat}^* \circ f^p.(|[a] \odot [\pi]) = \text{flat}^* \circ f^p.(a \rightarrow [\pi])
   \]
   (16)
   We call this property – commutativity – because it implies that the order in which the deconstruction operators tie and cons (or other combinations) are applied does not change the result.

Simple examples of functions that satisfy the commutativity property are: \text{map} and \text{reduce} (see Example 4).

Lemma 1: The commutativity property can be generalized as follows:
   \[
   \text{flat}^* \circ f^p.(a \rightarrow \text{distr}^r.m.u) = \\
   \text{flat}^* \circ f^p.(a \oplus \text{distr}^r.m.u), \forall 0 \leq m
   \]
   (17)
where the function flat\textsuperscript{*} is defined based on the same operators as the function distr.

**Proof:** The proof is based on complete induction on the maximum length \( n \) of the elements (sublists) of the distributed list distr\textsuperscript{*}.m.u.

**Base case** \( n = 1 \Rightarrow \) distr\textsuperscript{*}.m.u = \( \pi \). So, commutativity definition can be used.

**Inductive step** We assume that the relation 17 is true in all cases in which the lengths of the sublists of distr\textsuperscript{*}.m.u are less than \( n \), and we prove that it is also true for the case where \( n \) is at least one sublist with the length equal to \( n + 1 \).

We use the following notations:

\[
d = \text{distr}\textsuperscript{*}.m.u = [d_1, d_2, \ldots]
\]

\[
u = x \triangleright u
\]

We know that \( d = x \leftrightarrow \text{distr}\textsuperscript{*}.m-1.u.x \). Based on Property 4, we can prove that the first element of \( u \) is also the first element of \( d1 \), and so we have \( d1 = x \triangleright d.x \).

We start from the equality

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright [x] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x) =
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

which is obtained from the induction hypothesis. Then we transform each term of the equality:

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright [x] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x) =
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m-1.u.x)
\]

From these we obtain

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m.u) = \text{flat}\textsuperscript{*} \circ f^\prime.([a] \triangleright \text{distr}\textsuperscript{*}.m.u)
\]

**Lemma 2:** For an extended scalar function on ParLists \( f_s \), the corresponding bounded parallel function \( f_s^2 \) has the following property:

\[
f_s^2.(\text{distr}\textsuperscript{*}.p_i.u) = \text{distr}\textsuperscript{*}.p_i.(f_s.u)
\]

**Proof:** The proof is based on structural induction.

**Base case**

\[
u = [a] \Rightarrow \text{flat} \circ f_s^2.[a] = \text{flat} \circ [f_s.a] = f_s.a
\]

**Even inductive step**

\[
\nu = p/q \Rightarrow
\]

\[
f_s^2.(\text{distr}\textsuperscript{*}.p_i.(p/q)) = \text{distribution and scalar functions definition}
\]

\[
f_s^2.(\text{distr}\textsuperscript{*}.p_i-1.p) = \text{induction hypothesis}
\]

\[
\text{distr}\textsuperscript{*}.p_i-1.f_s^2.p = \text{distribution definition}
\]

\[
\text{distr}\textsuperscript{*}.p_i.(f_s^2.(p/q))
\]

**Odd inductive step**

\[
u = a \triangleright p \Rightarrow
\]

\[
f_s^2.(\text{distr}\textsuperscript{*}.p_i.(a \triangleright p)) = \text{distribution definition and Lemma 1}
\]

\[
f_s^2.(\text{distr}\textsuperscript{*}.p_i-1.p) = \text{scalar functions' properties}
\]

\[
f_s^2.(a) \triangleright \text{distr}\textsuperscript{*}.p_i-1.p = \text{induction hypothesis}
\]

\[
\text{distr}\textsuperscript{*}.p_i.(f_s.(a \triangleright p)) = \text{distribution and scalar functions definition}
\]

**Theorem 2:** For a ParList function \( f \) and a corresponding distribution distr\textsuperscript{*}.p (defined based on the same operators), if the function satisfies the commutativity property, then

\[
\text{flat}\textsuperscript{*} \circ f^\prime.(\text{distr}\textsuperscript{*}.p_i.u) = f.u
\]

**Proof:**

For ParList functions the proofs are based on the structural induction principle that treats the base case, and two inductive steps [6]. The even inductive step relies on the demonstration for transformation of the functions on PowerLists, so it is already proved. Also, the base case is trivial.

The proof of the odd case is based on Lemma 1 and Lemma 2:

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a \triangleright \text{distr}\textsuperscript{*}.p_i.p_i-1.p]) = \text{distribution definition}
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a \triangleright \text{distr}\textsuperscript{*}.p_i-1.p) = \text{Lemma 1}
\]

\[
\text{flat}\textsuperscript{*} \circ f^\prime.([a \triangleright \text{distr}\textsuperscript{*}.p_i-1.p)) = \text{distribution definition}
\]

\[
\Phi_s^2(a, p, f_s, \text{flat} \circ f^\prime.(\text{distr}\textsuperscript{*}.p_i-1.p)) = \text{induction hypothesis}
\]

\[
\Phi_s^2(a, p, f_s, f) = \text{f definition}
\]

\[
f.(a \triangleright p)
\]
Example 4 (Reduce and Map): The high order functions reduce and map satisfy the commutativity condition, and so they could easily be transformed for bounded parallelism. First, we consider the function reduce.(⊕) over ParList, where ⊕ is an associative operator:

\[
\text{reduce} : (X \times X \rightarrow X) \times \text{ParList}.X.n \rightarrow X
\]
\[
\text{reduce}(\oplus).(a \mapsto [a]) = a
\]
\[
\text{reduce}(\oplus).(p \circ q) = \text{reduce}(\oplus).p \oplus \text{reduce}(\oplus).q
\]
\[
\text{reduce}(\oplus).(a \triangleright u) = a \oplus \text{reduce}(\oplus).u
\]

To prove the commutativity property we use induction on the length of \(u\):

Base case \(u = [x]\)

\[
\text{flat}^b \circ \text{reduce}^p.(\oplus).(a \mapsto [x]) = [a \oplus x] = \text{flat}^b \circ \text{reduce}^p.(\oplus).u \]

The operator \(\triangleright\) is applied only for odd length lists, so \(u\) must have even length.

Inductive step

\[
\text{flat}^b \circ \text{reduce}^p.(\oplus).(a \mapsto \overline{p/q}) = \{\text{property of } \overline{p/q} \text{ and equation } 13\} \text{flat}^b \circ \text{reduce}^p.(\oplus).(a \mapsto \overline{p/q}) = \{\text{definition of the functions } \text{reduce} \text{ and } \text{flat}^b\} \text{flat}^b \circ \text{reduce}^p.(\oplus).(a \mapsto \overline{p/q}) = \{\text{induction hypothesis}\} \text{flat}^b \circ \text{reduce}^p.(\oplus).(a \mapsto \overline{p/q}) \oplus \text{flat}^b \circ \text{reduce}^p.(\oplus).\overline{q} = \{\text{definition of the function } \text{reduce}\} \text{flat}^b \circ \text{reduce}^p.(\oplus).(a \mapsto \overline{p/q}) = \{\text{induction hypothesis}\} \text{flat}^b \circ \text{reduce}^p.(\oplus).([a] \oplus p \text{ reduce}^p.(\oplus).\overline{q}) = \{\text{definition of the function } \text{reduce}\} \text{flat}^b \circ \text{reduce}^p.(\oplus).([a] \oplus p \text{ reduce}^p.(\oplus).\overline{q}) = \{\text{induction hypothesis}\} \text{flat}^b \circ \text{reduce}^p.(\oplus).([a] \oplus p \text{ reduce}^p.(\oplus).\overline{q})
\]

The proof of commutativity property for function map is similar.

For the functions that satisfy the commutativity condition, the use of distributions on ParLists is very advantageous, since the parallel computation is based on PowerLists.

Example 5 (Prefix Sum): This function yields, for any associative binary operator \(\odot\) and a list, the list of “prefix sums”. For example, on a list of three elements we have:

\[
\text{prefix}(\odot).[a \ b \ c] = [a \ a \odot b \ a \odot b \odot c]
\]

One way to compute \(\text{prefix}\) is given by the following definition:

\[
\text{prefix}(\odot).[a] = [a]
\]
\[
\text{prefix}(\odot).\circ p = \text{prefix}(\odot).p \circ \text{map}(\text{last}(\text{prefix}(\odot).p).\odot)(\text{prefix}.q)
\]
\[
\text{prefix}(\odot).p \circ a = \text{prefix}(\odot).p \circ \text{map}(\text{last}(\text{prefix}(\odot).p).\odot).[a]
\]

The function \(\text{prefix}\) can be seen as a composition of the functionals \(\text{reduce}.(\odot)\) and \(\text{map}.[\cdot]\), where the operator \(\odot\) is defined by \(u \odot v = u[\text{map}(\text{last}.u) \circ v]\); so, the function \(\text{prefix}\) satisfies the commutativity property, too.

5) Time-complexity: 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 \(f^r\) 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 \(f^r\) 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 \(\langle\) is used). So, for the first case, the time-complexity formula is

\[
T = \Theta + T(f^r) + T(f^r.(\text{first distr}.p.u))
\]

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

C. Discussion

It could be argued that the commutativity property imposes a constraint which is too restrictive. We have a constraint, but still there is a large class of functions that satisfy this constraint. In BMF (Bird-Meertens Formalism) of lists [1], [2], [12], [3] the analysis is done only for homomorphisms, which are functions on lists that can be expressed as a composition of a \text{reduce} function (with a certain associative operator) and a \text{map} function. Bounded parallelism is discussed there too, but only for concatenation operator (so linear distribution or block). We have proved that \text{map} and \text{reduce} satisfy the commutativity property, so all combinations of them satisfy it as well.
IV. Conclusions

The PowerList, PowerArray, and ParList notations have been proved to be a very elegant way to specify divide&conquer parallel algorithms and to prove their correctness. The main advantage of a model based on them is that it offers a simple, formal, and elegant way to prove correctness. Their abstraction is very high, but we may reconcile this abstractness with performance by introducing bounded parallelism. The necessity of this kind of reconciliation for parallel computation models was argued by Gorlatch in [3], and also by Skilllicorn and Talia in [13].

These theories are based not only on simple concatenation – by using operators tie, cons, and snoc, but also on interleaving – by using the operator zip. Another advantage of these theories lies in the fact that operators tie and zip force the decomposition in equal parts, so load-balance is implied.

We have proved that the already defined PowerList or ParList functions could be easily transformed to accept bounded parallelism, by introducing distributions. The distributions are defined in the same way as we define functions representing programs – based on pattern matching. Hence, distribution properties have been proved by induction too. Also, choosing a distribution strongly depends on initial function definition: it depends on the decomposition operator which is used. It can be argued that introducing distributions in these theories is not really necessary since we may informally specify that when the maximal number of created processes is achieved, the implementation transforms any parallel decomposition into a sequential one. Still, there are several advantages of formally introducing the distributions; the first and the most important is based on a practical reason: the required number of processors grows linearly with the size of the lists; the second is that it allows us to evaluate costs, depending on the number of available processors - as a parameter. Another advantage is that we may control the parallel decomposition until a certain level of tree decomposition is achieved; otherwise parallel decomposition could be done, for example, in a ‘depth-first’ manner, which could be disadvantageous. The analysis of the possible distributions for a certain function may lead to an improvement in the design decisions, too.

After the introduction of the distributions functions, mapping on real architectures with limited number of processing elements (e.g. hypercubes) can be achieved. Since, the functions over ParLists are transformed into PowerList function by using distributions, implementation on hypercubes are possible for them, too.

ACKNOWLEDGMENT

This work was supported by the research project ID-2286, Contract No. 477/2008, sponsored by the Romanian National University Research Council (CNCSIS).

REFERENCES