KNOWLEDGE ENGINEERING: PRINCIPLES AND TECHNIQUES
Proceedings of the International Conference on Knowledge Engineering,
Principles and Techniques, KEPT2007
Cluj-Napoca (Romania), June 6–8, 2007, pp. 261–268

INTRODUCING DATA-DISTRIBUTIONS INTO *POWERLIST* THEORY

VIRGINIA NICULESCU[1]

Abstract. *PowerList* theory is well suited to express recursive, data-parallel algorithms. Its abstractness is very high and assures simple and correct design of parallel programs. We try to reconcile this high level of abstraction with performance by introducing data-distributions into this theory. One advantage of formally introducing distributions is that it allows us to evaluate costs, depending on the number of available processors, which is considered as a parameter. Also, the analysis of the possible distributions for a certain function may lead to an improvement in the design decisions. Another important advantage is that after the introduction of data-distributions, mappings on real parallel architectures with limited number of procesing elements could be analyzed.

1. Introduction

*PowerList*s are data structures introduced by Misra [3], which can be successfully used in a simple and provable correct, 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. To assure methods that verify the correctness of the parallel programs, algebras and structural induction principles are defined on these data structures. Based on the structural induction principle, functions and operators, which represent the parallel programs, are defined. Generally, unbounded parallelism (the number of processes is not limited) is analyzed using these structures. Still, the most practical approach of bounded parallelism may be introduced, and so, the distributions, too.

Mappings on hypercubes have been analyzed for the programs specified based on these notations [3, 2]; they are based on Gray code. The analysis assumes that the hypercube has more nodes that the lists which are mapped onto, and this is an unrealistic assumption.

2000 *Mathematics Subject Classification.* 65Y05, 68Q85.

Key words and phrases. parallel computation, abstraction, design, distribution, data-structures.

The PowerList notation has been proved to be a very elegant way to specify parallel algorithms and prove their correctness. The main advantage of this model is that offers a simple, formal and elegant way to prove correctness. By formally introducing data distribution in this model, we enhance it with the possibility of formally evaluating costs for the case of bounded parallelism. In this way the high level of abstraction of these theories is reconcile with the performance.

2. *PowerList* Theory

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* l 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 is denoted by $[a]$. If two *PowerList* structures have the same length and elements of the same type, they are called *similar*s.

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

- using the operator *tie* $p \mid q$; the result contains elements from p followed by elements from q
- using the operator *zip* $p \natural q$; the result contains elements from p and q, alternatively taken.

Therefore, the constructor operators for *PowerList* are:

$$\begin{array}{lcl} [.] & : & X \to PowerList.X.0 \\ .|. & : & PowerList.X.n \times PowerList.X.n \to PowerList.X.(n+1) \\ .\natural. & : & PowerList.X.n \times PowerList.X.n \to PowerList.X.(n+1). \end{array} \tag{2}$$

Functions are defined 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:

$$\begin{array}{l} map.f.[a] = [f.a] \\ map.f.(p \natural q) = map.f.p \natural map.f.q \end{array} \tag{3}$$

Another example is the function $flat$ that is applied to a PowerList with elements which are in turn PowerLists, and it returns a simple PowerList:

$$\begin{array}{l} flat.[l] = l \\ flat.(p \natural q) = flat.p \natural flat.q \quad \text{or } flat.(p \mid q) = flat.p \mid flat.q \end{array} \tag{4}$$

The reduction of the list to an element by using an associative operator $\oplus$ is defined by:

$$\begin{array}{l} red.\oplus.[a] = a \\ red.\oplus.(p|q) = red.\oplus.p \oplus red.\oplus.q \end{array} \tag{5}$$

Binary associative operators on scalar types can be extended to *PowerList*.

3. 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. That 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.

Definition 1. *$D = (\delta, A, B)$ is called a (one-dimensional) distribution if A and B are finite sets, and δ is a mapping from A to B; set A specifies the set of data objects (an array with n elements that represent the indices of data objects), and the set B specifies the set of processes, which is usually $\overline{p}$. The function δ assigns each index $i(0 \leq i < n)$, and its corresponding element, to a process number* [4].

One advantage of *PowerList* theory is that it does not need to use indices, and this simplifies very much reasoning and correctness proving. So, we will introduce distributions not as in the definition above, but as functions on these special data structures.

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.

3.1. *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

$$\begin{aligned} & distr^l.p.(u|v) = distr^l.(p-1).u \mid distr^l.(p-1).v \\ & distr^l.0.l = [l] \\ & distr^l.p.x = [x], \text{ if } loglen.x < p. \end{aligned} \tag{6}$$

- cyclic

$$\begin{aligned} & distr^c.p.(u\natural v) = distr^c.(p-1).u \natural distr^c.(p-1).v \\ & distr^c.0.l = [l] \\ & distr^c.p.x = [x], \text{ if } loglen.x < p. \end{aligned} \tag{7}$$

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

3.1.1. *Properties.* If we consider $u \in PowerList.X.n$, then $distr.n.u = \overline{u}$, where $\overline{u}$ is obtained from the list u by transforming each of its elements into a singleton list.

Also, we have the trivial property $distr.0.u = [u]$.

The result of the application of a distribution $distr.p$ to a list $l \in PowerList.X.n$, $n \geq p$ is a list that has 2^p elements each of these being a list with 2^{n-p} elements of type X.

The properties are true for both linear and cyclic distributions.

3.1.2. *Function Transformation.* We consider a function f defined on $PowerList.X.n$ based on operator *tie* with the property that

$$f.(u|v) = \Phi(f.x_0, f.x_1, \ldots, f.x_m, u, v), \tag{8}$$

where $x_i \in PowerLists.X.k, k = loglen\ n - 1$, and $x_i = e_i.u.v, \forall i : 0 \leq i \leq m$, and e_i and Φ are expressions that may use scalar functions and extended operators on PowerLists. If the function definition Φ is more complex and uses other functions on $PowerList$s, then these functions have to be transformed first, and the considered function after that.

A scalar function f has zero or more scalars as arguments, and its value is a scalar. The function f is easily extended to a PowerList by applying f "pointwise" to the elements of the PowerList. A scalar function that operates on two arguments could be seen as an infix operator, and it also could be extended to PowerLists.

The extensions of the scalar functions on PowerLists could be defined either using the operator tie or zip. Some properties of these functions could be find in [3]. For the sake of the clarity, we will introduce the notation f^1 that specifies the corresponding extended function on PowerLists of the scalar function f defined on the scalar type X. For the case of one argument the definition is:

$$\begin{array}{l} f^1 : PowerList.X.n \to PowerList.X.n \\ f^1.[a] = [f.a] \\ f^1.(p|q) = f^1.p|f^1.q \ \text{ or } f^1.(p\natural q) = f^1.p\natural f^1.q \end{array} \tag{9}$$

Further, f^2 (which is the notation for the extension of f on PowerLists with elements which are in turn PowerLists) could be defined:

$$\begin{array}{l} f^2 : PowerList.(PowerList.X.m).n \to PowerList.(PowerList.X.m).n \\ f^2.[a] = [f^1.a] \\ f^2.(p|q) = f^2.p|f^2.q \ \text{ or } f^2.(p\natural q) = f^2.p\natural f^2.q \end{array} \tag{10}$$

We intend to show that

$$f.u = flat \circ f^p.(dist^l.p.u),$$

where

$$
\begin{array}{ll}
 & f^p.(u|v) = \Phi^2(f^p.x_0, f^p.x_1, \ldots, f^p.x_m, u, v) \\
(11) & f^p.[l] = [f^s.l] \\
 & f^s.u = f.u
\end{array}
$$

Function f^p corresponds to parallel execution, and function f^s corresponds to sequential execution.

Lemma 1. *Given a scalar function $f : X \to X$, and a distribution function $distr.p$, defined on $PowerList.X.n$, then the following equality is true*

$$dist.p \circ f^1 = f^2 \circ dist.p \tag{12}$$

Proof. To prove this lemma we use induction on p.

We prove the case of the linear distribution, but the case of the cyclic distribution $dist^c.p$ is similar.

Base case$(p = 0)$

$$
\begin{array}{ll}
 & f^2.(dist^l.0.u) \\
= & \{p = 0 \Rightarrow dist^l.p.u = [u]\} \\
 & f^2.[u] \\
= & \{f^2 \text{ definition}\} \\
 & [f^1.u] \\
= & \{distr^l.0 \text{ definition}\} \\
 & distr^l.0.(f^1.u)
\end{array}
$$

Inductive step

$$
\begin{array}{ll}
 & f^2.(dist^l.p.(u|v)) \\
= & \{ \text{ definition of } distr^l\} \\
 & f^2.(dist^l.(p-1).u|dist^l.(p-1).v) \\
= & \{f^2 \text{ definition}\} \\
 & f^2.(dist^l.(p-1).u)|f^2.(dist^l.(p-1).v)) \\
= & \{\text{induction assumption}\} \\
 & distr^l.(p-1).(f^1.u)|distr^l.(p-1).(f^1.u) \\
= & \{distr^l \text{ definition}\} \\
 & distr^l.p.(f^1.u|f^1.v) \\
= & \{f^1 \text{ definition}\} \\
 & distr^l.p.(f^1.(u|v))
\end{array}
$$

□

The previous result is naturally extended to scalar functions with more arguments, such as infix operators.

Scalar binary associative operators ($\oplus$), could be also extended on PowerLists as reduction operators – $red(\oplus)$. They transform a PowerList into a scalar. For

them, similar extensions as for scalar functions may be done.

$$\begin{array}{l} red^1(\oplus) : PowerList.X.m \to X \\ red^1(\oplus).[a] = a \\ red^1(\oplus).(p|q) = red^1(\oplus).p \oplus red^1(\oplus).q \end{array} \tag{13}$$

$$\begin{array}{l} red^2(\oplus) : PowerList.(PowerList.X.m).n \to PowerList.X.0 \\ red^2(\oplus).[l] = [red^1(\oplus).l] \\ red^2(\oplus).(p|q) = [red^1(\oplus).\big(red^2(\oplus).p|red^2(\oplus).q\big)] \end{array} \tag{14}$$

Also, a similar property in relation to distributions is obtained:

$$distr.p \circ red^1(\oplus) = red^2(\oplus) \circ distr.p \tag{15}$$

Theorem 1. *Given a function f defined on $PowerList.X.n$ as in Eq. 8, a corresponding distribution $distr^l.p, (p \leq n)$, and a function f^p defined as in Eq. 11, then the following equality is true*

$$f = flat \circ (f^p \circ dist^l.p) \tag{16}$$

Proof. We will proof the following equation

$$\begin{array}{c} (distr^l.p. \circ f).u = (f^p \circ dist^l.p).u \\ \text{for any } u \in PowerList.X.n \end{array} \tag{17}$$

which implies the equation 16. To prove this, we use again induction on p.

Base case$(p = 0)$

$$\begin{array}{ll} & f^p.(dist^l.0.u) \\ = & \{p = 0 \Rightarrow dist^l.p.u = [u]\} \\ & f^p.[u] \\ = & \{f^p \text{ definition}\} \\ & [f^s.u] \\ = & \{f^s \text{ definition}\} \\ & distr^l.p.(f.u) \end{array}$$

Inductive step

$$
\begin{array}{ll}
 & f^p.(dist^l.p.(u|v)) \\
= & \{ \text{ definition of } distr^l\} \\
 & f^p.(dist^l.(p-1).u|dist^l.(p-1).v) \\
= & \{f^p \text{ definition, scalar functions properties}\} \\
 & \Phi^2(f^p.(e_0^2.(dist^l.(p-1).u).(dist^l.(p-1).v)),\ldots, \\
 & \qquad f^p.(e_m^2.(dist^l.(p-1).u).(dist^l.(p-1).v)), dist^l.(p-1).u, dist^l.(p-1).v) \\
= & \{e_i \text{ are simple expressions – use scalar functions}\} \\
 & \Phi^2(f^p \circ\ distr^l.(p-1).(e_0.u.v),\ldots, \\
 & \qquad f^p \circ\ distr^l.(p-1).(e_m.u.v), dist^l.(p-1).u, dist^l.(p-1).v) \\
= & \{\text{induction assumption, and scalar functions properties}\} \\
 & (distr^l.p \circ \Phi)(f.(e_0.u.v),\ldots,f.(e_m.u.v),u,v) \\
= & \{f \text{ definition}\} \\
 & distr^l.p.(f.(u|v))
\end{array}
$$

□

For cyclic distribution the proof is similar; the operator *tie* is replaced with the operator *zip*.

3.2. **Time Complexity.** 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:

$$T = \alpha T(f^p) + T(f^s)$$

where α reflects the costs specific to parallel steps (communication or access to shared memory). The evaluation considers that the processor-complexity is 2^p ($O(2^p)$ processors are used).

Example.(**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:

$$T = K_c p\alpha + K_c(2^{n-p} - 1) + K_s 2^{n-p} \tag{18}$$

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

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

$$T_{red} = K_\oplus(p\alpha + 2^{n-p} - 1) \tag{19}$$

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

$$T = K_s 2^{n-p} \tag{20}$$

4. Conclusions

The *PowerList* theory forms an abstract model for parallel computation. It is very efficient for developing divide&conquer parallel programs. The abstractness is very high, but we may reconcile this abstractness with performance by introducing bounded parallelism, and so distributions. The necessity of this kind of reconciliation for parallel computation models is argued by Gorlatch in [1], and also by Skillicorn and Talia in [7].

We have proved that the already defined functions on *PowerLists* could be easily transformed to accept bounded parallelism, by introducing distributions. The functions defined based on operator *tie* have to use linear distributions, and the functions defined based on operator *zip* have to use cyclic distributions.

It can be argued that introducing distributions in this theory 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, one advantage of formally introducing the distributions is that it allows us to evaluate costs, depending on the number of available processors - as a parameter. Also, the analysis of the possible distributions for a certain function may lead to an improvement in the design decisions. 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 'deep-first' manner, which could be disadvantageous.

Also, after the introduction of distribution functions, mapping on real architectures with limited number of processing elements (e.g. hypercubes) could be analyzed.

References

[1] Gorlatch, S.,: *Abstraction and Performance in the Design of Parallel Programs*, CMPP'98 First International Workshop on Constructive Methods for Parallel Programming, 1998.
[2] Kornerup, J.: *Data Structures for Parallel Recursion.* PhD Thesis, Univ. of Texas, 1997.
[3] Misra, J.: *PowerList: A structure for parallel recursion. ACM Transactions on Programming Languages and Systems*, Vol. 16 No.6 (1994) 1737-1767.
[4] Niculescu, V.: *On Data Distributions in the Construction of Parallel Programs*, The Journal of Supercomputing, Kluwer Academic Publishers, 29(1): 5-25, 2004.
[5] Niculescu, V.: *Designing a Divide&Conquer Parallel Algorithm for Lagrange Interpolation Using* Power, Par, *and* P *Theories*, Proceedings of the Symposium "Zilele Academice Clujene", 2004, pp. 39-46.
[6] Skillikorn, D.B.: *Structuring data parallelism using categorical data types.* In Programming Models for Massively Parallel Computers, pp. 110-115, 1993, Computer Society Press.
[7] Skillicorn, D.B. and Talia, D.: *Models and Languages for Parallel Computation.* ACM Computer surveys, 30(2): 123-136, June 1998.

[1] Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: vniculescu@cs.ubbcluj.ro