Posters: Extended Abstracts

Parallelising Programs with Algebraic Programming Tools

Anatoly E. Doroshenko and Alexander B. Godlevsky

Glushkov Institute of Cybernetics
National Academy of Sciences of Ukraine
Glushkov prosp., 40, Kiev 252650, Ukraine
E-mail: dor@d105.icyb.kiev.ua

Abstract. A rewriting based approach to dynamical parallelization of a general class of sequential imperative programs by means of the algebraic programming system APS is proposed. It gives advantages of rapid prototyping and evolutionary development of efficient parallelizers. The paper shows major features of a dynamical parallelizer implemented in the APS as well as techniques for designing efficient parallelizers.

1 Introduction

The dominant trend in automating programming for parallel computers is connected with compiler technology and remarkable achievements have been made in last decade in this area. (e.g. [3]). Nevertheless to develop a compiler for parallel computer system is still a difficult and expensive task that imposes a lot of restrictions and language simplifications for compiler to be practical and efficient. Therefore prototyping is an important tool to save efforts and time especially in its declarative form [1]. We follow an integrated compiler/interpreter approach to parallel software design having in mind that writing interpreter is much easier task than developing compiler but interpreters are commonly known to be inefficient. The emphasize is maden on program transformations and dynamical issues of parallelization that can be solved in compile-time and run-time respectively at low cost. Such approach that we call *dynamical parallelization* has been proved to be acceptable in large-grained computations for macroconveyer parallel multicomputers [6]. Recently likely approach was undertaken for run-time parallelization of functional languages [4].

In this paper we report on exploiting the approach of dynamical parallelization of programs on a new platform of the algebraic programming system APS [5] based on rewriting rules programming techniques. Early experience of application of the APS system for this purposes is described in [2]. It was not our goal to propose a new method of data dependence analysis and in this part we follow well known decisions adopted in parallelizing compilers. What is significant and new is to provide declarative treatment of programs analysis and transformations by means of rewriting techniques in the system of algebraic programming. It gives a great deal of flexibility of parallelizers and allows extracting parallelism independently on multiprocessor architecture and number of processors available in parallel system. Besides of rapid prototyping the advantages consist in controlability and verifiablity of the whole process of evolutionary program development.

2 The APS Main Features

The APS is an integrated rewriting rule based programming system. A methodology of the system application consists in flexible integration of four main paradigms of programming: procedural, functional, algebraic and logical that is achieved by adjusted use of corresponding computational mechanisms. The main objects in the system are terms of the algebra that is considered as absolutely free algebra of infinite (but finitely represented) trees. As a values of names these trees may have common parts and may be used to represent arbitrary labelled graphs. There are three types of system objects: *algebraic programs* (ap-modules), *algebraic modules* (a-modules) and *interpreters*.

Algebraic programs are texts in APLAN language syntax [5]. Each program contains the description of some signature of underlying algebra with syntax for constructing algebraic expressions (terms). It defines also the set of names and atoms. These objects together with numbers and strings constitutes the set of *primary objects*. The sets of names and atoms together with the signature of an ap-module define the *type* of this ap-module.

Algebraic modules contain internal representation of the data structures defined in ap-modules. They are being created by system commands that refer to ap-modules as a new object generators. The notion of a-module is dynamical one. It has a state which may changes in time. The change of the state of a-module takes place as a result of executing procedures located in it by means of interpreters. System interpreters are programs destined for the interpretation of the procedures written in APLAN. They are developing in C language on the base of libraries of functions and data structures to work with internal representation of system data structures. Each interpreter is connected with a distinct type which defines the restriction to algebraic modules which can be executed by the given interpreter. Each interpreter specifies the operational semantics of APLAN for the given class of a-modules and provides efficient implementation of the procedures, functions and strategies of rewriting for the systems located in the given module.

3 Rewriting Techniques for Dynamical Parallelization

To give a flavor of the APS and to demonstrate the rewriting style programming for parallelization we consider a short fragment of ap-modules that realises a piece of data dependency analysis implemented in our dynamical parallelizer — evaluation of the fact that two sets of array variables are disjoint.

If we designate the property of intersection nonemptyness of two sets of array variables V and W with predicate $Int(V, W)$ and a function evaluating the number of dimensions of array variable x with $art(x)$ then we can write following recurrence relations:

$$Int(V, W) \iff (\exists x, i, j)(x(i) \in V, x(j) \in W)$$
$$\& \ (n = art(x) \ \& \ Indx(i, j, n)$$
$$Indx(i, j, n) \iff (i = (i_1, \ldots, i_n),$$
$$j = (j_1, \ldots, j_n)) \ \&$$

$$(\forall l:\ 1 \le l \le n)(Noncomp(i_l, j_l) = 0)$$

$$Noncomp(k, m) \iff (k \neq m)\ \&\ (k, m\ are\ integers)$$

Informally, these relations mean that intersection is nonempty iff both sets V and W comprise two elements of the same array that in every component of index sets have expressions or coinsided integers.

The following fragment of APLAN code realises these relations using rewriting rules (abbreviated rs) in functional style to which standard interpreter is applied.

```
Int:=rs(x,x1,i,y,y1,j)(
   (nil,x) = 1, (x,nil) = 1,
   Indx(x(i) || x1,x(j)) ->
      ( (x(i) || x1,x(j) || y1) =
        Int(x(i) || x1 ,y1)),
      (x(i) || x1,x(j) || y1) = 0,
   compare(x,y) ->
      (( x(i) || x1,y(j) || y1)=
        Int(x1,y(j) || y1)),
      ( x(i) || x1,y(j) || y1 ) =
        Int(x(i) || x1,y1)
);
```

Its formal parameters x, i and j meaningly stand for just the same variables that in relation system of *Int*, others are additional. Fragment contains logical connection -> (implication), logical constants 0 and 1 and use ordered list representation of variable sets with *nil* standing for empty list and || for concatenation of list elements. These rewriting rules essentially consist of two parts. The second part prefixed with predicate $compare(x, y)$ is to seive two array variable sets and deleting from them all the variables whose names are different. The first part is to test index expressions of array variables with the same name for compatibility in the sense of *Int*.

4 Techniques for Parallelizers Development

To enhance dynamical parallelizers based on data dependency analysis some additional computational mechanisms aimed to breaking data dependences and transforming source programs to improve locality of computational activities for parallelization are developed. They are not new and are commonly used in compilers but we try to treat them as rewriting rules. Below are enumerated some of such techniques being intensively used in our parallelizer.

Concretization of variables. This rewriting technique consists in substituing values in algebraic expression instead of variables to reduce data dependences. Special but very important case of this technique is achieved when variables to be concretisized are indeed variables that body and/or condition of a loop are dependent on.

Localization of variables. This technique belongs to preliminary program transformations. The meaning of a localization constructs $loc(x)$ consists in generating a new copy of variable x whose scope is delimited syntactically by $loc(x)$ itself and

the nearest construct *endloc*. This gives a possibility to delete data dependence of constructs embraced on variable x with purely syntactic tools.

Coarse-grained computations. Defining some piece of computations as a *basic* operator we thereby represent it as a single operator (perhaps depending on parameters) in program dynamic parallelization. This technique of computations consolidation may be preferable due to at least two reasons. Firstly, it is tightly connected with coarse-grained parallelism in distributed memory multiprocessor systems and networks. Secondly, it is extremely agreed with dynamical mode of parallelization because it provides reducing parallelizer's workload, assists in transfering purely computational activity from parallelizer to processors of parallel system.

References

1. M. Chen, J. Cowie, Prototyping Fortran 90 Compilers for Massively Parallel Machines, *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation*, ACM Press, pp. 94-105, 1992.
2. A. B. Godlevsky, A. E. Doroshenko, Parallelizing Programs with APS, *ISSAC'93: Proc. ACM SIGSAM Int. Symp. on Symbolic and Algebraic Computation*, ACM Press, 1993, pp. 55-62.
3. S. Hiranandani, K. Kennedy, C.-W. Tseng, Compiling Fortran D for MIMD Distributed-Memory Machines, *Commun. ACM*, vol. 35(8), pp. 66-80, 1992.
4. L. Huelsbergen, J. Larus, Dynamic program parallelization, *Proc. 1992 ACM Conf. Lisp and Functional Programming*, ACM Press, pp. 311-323, 1992.
5. A.A.Letichevsky, J.V.Kapitonova, S.V.Konozenko, Computations in APS, *Theoretical Computer Science* **119**, 1993, pp.145-171.
6. V.S.Mikhalevich, Ju.V.Kapitonova, A.A.Letichevsky, On models of macroconveyer computations, in: *Information Processing 86* (IFIP, Amsterdam, 1986) 975-980.