A Parallelization Framework for Recursive Tree Programs

Paul Feautrier*

Laboratoire PRiSM,
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

Abstract. The automatic parallelization of "regular" programs has encountered a fair amount of success due to the use of the polytope model. However, since most programs are not regular, or are regular only in parts, there is a need for a parallelization theory for other kinds of programs. This paper explore the suggestion that some "irregular" programs are in fact regular on other data and control structures. An example of this phenomenon is the set of recursive tree programs, which have a well defined parallelization model and dependence test.

1 A Model for Recursive Tree Programs

The polytope model [7, 3] has been found a powerful tool for the parallelization of array programs. This model applies to program that use only `DO` loops and arrays with affine subscripts. The relevant entities of such programs (iteration space, data space, execution order, dependences) can be modeled as Z-polytopes, i.e. as sets of integral points belonging to bounded polyhedra. Finding parallelism depends on our ability to answer questions about the associated Z-polytopes, for which task one can use well known results from mathematics and operation research.

The aim of this paper is to investigate whether there exists other program models for which one can devise a similar automatic parallelization theory. The answer is yes, and I give as an example the *recursive tree programs*, which are defined in sections 1.4 and 1.5. The relevant parallelization framework is presented in section 2. In the conclusion, I point to unsolved problems for the recursive tree model, and suggest a search for other examples of parallelization frameworks.

1.1 An Assessment of the Polytope Model

The main lesson of the polytope model is that the suitable level of abstraction for discussing parallelization is the *operation*, i.e. an execution of a statement.

* e-mail : Paul.Feautrier@prism.uvsq.fr

In the case of DO loop programs, operations are created by loop iterations. Operations can be named by giving the values of the surrounding loop counters, arranged as an *iteration vector*. These values must be within the loop bounds. If these bounds are affine forms in the surrounding loop counters and constant *structure parameters*, then the iteration vector scans the integer points of a polytope, hence the name of the model.

To achieve parallelization, one has to find subsets of independant operations. Two operations are dependent if they access the same memory cell, one at least of the two accesses being a write. This definition is useful only if operations can be related to the memory cells they access. When the data structures are arrays, this is possible if subscripts are affine functions of iteration vectors. The dependence condition translates into a system of linear equations and inequations, whose unknowns are the surrounding loop counters. There is a dependence iff this system has a solution in integers.

These observations can be summarized as a set of requirements for a parallelization framework:

1. We must be able to describe, in finite terms, the set of operations of a program. This set will be called the *control domain* in what follows. The control domain must be ordered.
2. Similarly, we must be able to describe a data structure as a set of *locations*, and a fonction from locations to values.
3. We must be able to associate sets of locations to operations through the use of *address functions*.

The aim of this paper is to apply these prescriptions to the design of a parallelization framework for recursive tree programs.

1.2 Related Work

This section follows the discussion in [5]. The analysis of programs with dynamic data structures has been carried mainly in the context of pointer languages like C. The first step is the identification of the type of data structures in the program, i.e. the classification of the pointer graph. The main types are trees (including lists), DAG and general graphs. This can be done by static analysis at compile time [4], or by asking the programmer for the information. This paper uses the second solution, and the data structures are restricted to trees.

The next step is to collect information on the possible values of pointers. This is done in a static way in the following sense: the sets of possible pointer values are associated not to an operation but to a statement. These sets will be called *regions* here, by analogy to the array regions [9] in the polytope model. Regions are usually represented as *path expressions*, which are regular expressions on the names of structure attributes [8].

Now, a necessary (but not sufficient) condition for two statements to be in dependence is that two of their respective regions intersect. It is easy to see that this method incurs a loss of information which may forsake parallelisation in

important cases. The main contribution of this paper is to improve the precision of the analysis for a restricted category of recursive tree programs.

1.3 Basic Concepts and Notations

The main tool in this paper is the elementary theory of finite state automata and rational transductions. A more detailed treatment can be found in Berstel's book [1].

In this paper, the basic alphabet is $\mathbb{N}$ (the set of non negative integers). ϵ is the zero-length word and the point (.) denotes concatenation. In practical applications, the alphabet is always some finite subset of $\mathbb{N}$.

A finite state automaton (fsa) is defined in the usual way by states and labelled transitions. One obtains a word of the language generated (or accepted) by an automaton by concatenating the labels on a path from an initial state to a terminal state. A rational transduction is a relation on $\mathbb{N}^* \times \mathbb{N}^*$ which is defined by a *generalized sequential automaton* (gsa): an fsa whose edges are labelled by input and output words. Each time an edge is traversed, its input word is removed at the begining of the input, and its output word is added at the end of the ouput. Gsa are also known as Moore machines. The family of rational transductions is closed by inversion (simply reverse the elements of each digram), concatenation and composition [1].

A regular language can also be represented as a regular expression: an expression built from the letters and ϵ by the operations of concatenation (.), union (+) and Kleene star. This is also true for gsa, with letters replaced by digrams.

The domain of a rational transduction is a regular language whose fsa is obtained by deleting the second letter of each edge label. There is a similar construction for the range of a rational transduction.

From one fsa c, one may generate many others by changing the set of initial or terminal states. $c(s;)$ is deduced from c by using s as the unique initial state. $c(;t)$ has t as its unique terminal state. In $c(s;t)$, both the initial and terminal states have been changed. Since rational transductions are defined by automata, similar operations can be defined for them.

1.4 The Control Domain of Recursive Programs

The following example is written in a C-like language which will be explained presently:

```
BOOLEAN tree leaf;                          void main(void) {
int tree value;                             5 : sum([]);}
void sum(address I) {                       }
1 : if(! leaf[I]) {
2 :     sum(I.1);
3 :     sum(I.2);
4 :     value[I] = value[I.1] + value[I.2]
        }
}
```

value is a tree whose nodes hold integers, and **leaf** is a Boolean tree. The problem is to sum up all integers on the leaves, the final result being found at the root of the tree. Addresses will be discussed in Sect. 1.5.

Let us consider statement 4. Execution of this statement results from a succession of recursive calls to **sum**, the statement itself being executed as a part of the body of the last call. Naming these operations is achieved by recording where each call to **sum** comes from: either line 2 or 3. The required call strings can be generated by a *control automaton* whose states are the functions names and the basic statements of the program. The state associated to **main** is initial and the states associated to basic statements are terminal. There is a transition from state p to state q if p is a function and q is a statement occuring in p body. The transition is labelled by the label of q in the body of p.

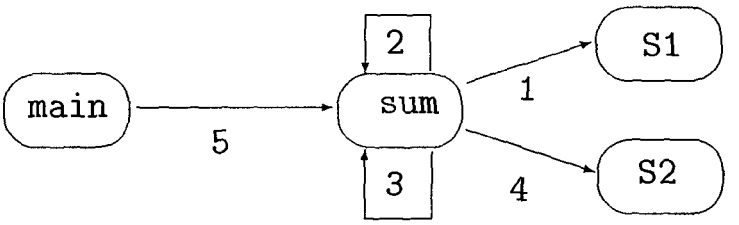

Fig. 1. The Control Automaton of **sum**

As the example shows, if the statements in each function body are labeled by ascending numbers, the execution order is exactly lexicographic ordering on the call strings.

1.5 Addressing in Trees

Remark first that most tree algorithms in the literature are expressed recursively. Observe also that in the polytope model, the same mathematical object is used as the control space and the set of locations of the data space. Hence, it seems quite natural to use trees as the prefered data structure for recursive programs.

In a tree, let us number the edges coming out of a node from left to right by consecutive integers. The name of node n is then simply the string of edge numbers which are encountered on the unique path from the root of the tree to n. The name of the root is the zero-length string, ϵ. This scheme dates back at least to Dewey Decimal Notation as used by librarians the world over.

The set of locations of a tree structure is thus $\mathbb{N}^*$, and a tree object is a partial function from $\mathbb{N}^*$ to some set of values, as for instance the integers, the floating point numbers or the characters.

Address functions map operations to locations, i.e. integer strings to integer strings. The natural choice for them is the family of *rational transductions* [1]. Consider again the above example. Notice the global declaration for the trees `value` and `leaf`. `address` is the type of integer strings. In line 4, such addresses are used to access `value`. The second address, for instance, is built by postfixing the integer 1 to the value of the address variable `I`. This variable is initialized to ϵ at line 5 of `main`. If the call at line 2 (resp. 3) of `sum` is executed, then a 1 (resp. 2) is postfixed to `I`.

The outcome of this discussion is that at entry into function `sum`, `I` comes either from lines 2 or 3 or 5, hence the regular equation:

$$I = \langle 5, \epsilon \rangle + I.\langle 2, 1 \rangle + I.\langle 3, 2 \rangle,$$

whose solution is the regular expression:

$$I = \langle 5, \epsilon \rangle.(\langle 2, 1 \rangle + \langle 3, 2 \rangle)^*.$$

Similarly, the second address in line 4 is given by the following rational transduction : $\langle 4, \epsilon \rangle.(\langle 2, 1 \rangle + \langle 3, 2 \rangle)^*.\langle 4, 1 \rangle$.

I conjecture that the reasoning that has been used to find the above address functions can be automated, but the details have not been worked out yet. It seems probable that this analysis will succeed only if the operations on adresses are suitably limited. Here, the only allowed operator is postfixing by a word of $\mathbb{N}^*$. This observation leads to the definition of a toy language, $\mathcal{T}$, which is similar to C, with the following restrictions:

- No pointers are allowed. They are replaced by addresses.
- The only data structures are scalars (integers, floats and so on) and trees thereof. Trees are always global variables. Addresses can only be used as local variables or functions parameters. No function may return an address.
- The only control structures are the conditionals and the function calls, possibly recursive. No loops or `goto` are allowed.

2 Dependence Analysis of $\mathcal{T}$

2.1 Parallelization Model

When parallelizing static control programs, one has first to decide the shape of the parallel version. One usually distinguishes between *control parallelism*, where operations executed in parallel are instances of different statements, and *data parallelism*, where parallelism is found among iterations of the same statement. In recursive programs, repetition of a statement is obtained by enclosing it in the body of a recursive function, as for example in the program of Fig. 2.

Suppose it is possible to decide that the operations associated to `S` and all operations generated by the call to `foo` are independent. The parallel version in Fig. 3 (where `{^ ... ^}` is used as the parallel version of `{ ... }` [6]) is equivalent to the sequential original. The degree of parallelism of this program

```
void foo(x) {
 S;
 if (p) foo(y);
}
```

Fig. 2. Sequential version

```
void foo(x) {
 {^
 S;
 if(p) foo(y);
 ^}
}
```

Fig. 3. Parallel version

is of the order of the number of recursive calls to `foo`, which probably depends on the data set size. This is data paralellism expressed as control parallelism. A possible formalization is the following.

Let us consider a function `foo` and the statements $\{S_1, \ldots, S_n\}$ of its body. The statements are numbered in textual order, and i is the label of S_i. Tests in conditional statements are to be considered as elementary, and must be numbered as they occur in the program text.

Let us construct a synthetic dependence graph (SDG) for `foo`. The vertices of the SDG are the statements of `foo`. There is a dependence edge from S_i to $S_j, i < j$ iff there exists three iteration words u, v, w such that:

- u is an iteration of `foo`.
- Both $u.i.v$ and $u.j.w$ are iterations of some terminal statements S_k and S_l.
- $u.i.v$ and $u.j.w$ are in dependence.

Observe that u is an iteration word for `foo`, hence belongs to the language generated by $c(;\texttt{foo})$. Similarly, v is an iteration of S_k relative to an iteration of S_i, hence belongs to $c(S_i; S_k)$, and $w \in c(S_j; S_l)$. As a consequence, the pair $\langle u.i.v, u.j.w \rangle$ belongs to the following rational transduction:

$$h = c(;\texttt{foo})^{=}.\langle i, j \rangle.c(S_i; S_k).c(S_j; S_l)^{-1},$$

in which if a is an automaton, then $a^{=}$ is the transduction obtained by setting each output word equal to the corresponding input word. This formula also uses an automaton as a transduction whose output words have zero length. Similarly, the inverse of an automaton is used as a transduction whose input words have zero length.

S_i and S_j may also access local scalar variables, for which the dependence calculation is trivial. Besides, one must remember to add control dependences, from the test of each conditional to all statements in its branches. Lastly, dependences between stqtements belonging to opposite arms of a conditional are to be ommited.

Once the SDG is computed, a parallel program can be constructed in several well known ways. Here, the program is put in serie/parallel form by topological sorting of the SDG. As above, I will use the C-EARTH version of the `fork ... join` construct, `{^ ... ^}`. The run time exploitation of this kind of parallelism is a well known problem [6].

2.2 The Dependence Test

Computing dependences for the body of function `foo` involves two distinct algorithms. The first, (or outermost) one enumerates all pairs of references which are to be checked for dependence. This is a purely combinatorial algorithm, of polynomial complexity, which can be easily reconstructed by the reader.

The inner algorithm has to decide whether there exists three strings x, y, w such that:

$$\langle x, y\rangle \in h, \quad \langle x, w\rangle \in f, \quad \langle y, w\rangle \in g.$$

where the first term expresses the fact that x and y are iterations of S_k and S_l which are generated by one and the same call to `foo`, the second and third ones expressing the fact that both x and y access location w. f ans g are the address transductions of S_k and S_l. The first step is to eliminate w, giving $\langle x, y\rangle \in k = g^{-1} \circ f$. k is a rational transduction by Elgot and Mezei theorem [2]. Hence, the pair $\langle x, y\rangle$ belongs to the intersection of the two transductions h and k Deciding whether $h \cap k$ is empty is clearly equivalent to deciding whether $\ell \cap =$ is empty where $=$ is the equality relation and $\ell = k^{-1} \circ h$.

Deciding whether the intersection of two transductions is empty is a well known undecidable problem [1]. It is possible however to solve it by a semi-algorithm, wich is best presented as a (one person) game. A position in the game is a triple $\langle u, v, p\rangle$ where u and v are words and p is a state of ℓ. The initial state is $\langle \epsilon, \epsilon, p_0\rangle$, where p_0 is the initial state of ℓ. A position is a win if $u = v = \epsilon$ and if p is terminal. A move in the game consists in selecting a transition from p to q in ℓ with label $\langle x, y\rangle$. The outcome is a new position $\langle u', v', q\rangle$ where u' and v' are obtained from $u.x$ and $v.y$ by deleting their common prefix. A position is a loss if u and v begin by distinct letters: in such a case, no amount of postfixing can complete u and v to equal strings. There remains positions in which either u or v or both are ϵ. Suppose $u = \epsilon$. Then, for success, v must be the prefix of a string in the domain of ℓ when starting from p. This can be tested easily, and, if the check fails, then the position again is a loss. The situation is symmetrical if $v = \epsilon$.

This game may have three outcomes: if a win can be reached, then by restoring the deleted common prefixes, one reconstructs a word u such that $\langle u, u\rangle \in \ell$, hence a solution to the dependence problem. If all possible moves have been explored without reaching a win, then the problem has no solution. Lastly, the game can continue for ever. One possibility is to put an upper bound to the number of moves. If this bound is reached, one decides that, in the absence of a proof to the contrary, a dependence exists.

The following algorithm explores the game tree in breadth-first fashion.

Algorithm D.

1. Set $D = \emptyset$ and $L = \{\langle \epsilon, \epsilon, p_0\rangle\}$ where p_0 is the initial node of ℓ.
2. If $L = \emptyset$, stop. There is no dependence.
3. Extract the leftmost element of L, $\langle u, v, p\rangle$.
4. If $\langle u, v, p\rangle \in D$, restart at step 2.

5. If $u = v = \epsilon$ and if p is terminal, stop. There is a dependence.
6. If both $u \neq \epsilon$ and $v \neq \epsilon$, the position is a loss. Restart at step 2.
7. If $u = \epsilon$ and if v is not a prefix of a word in the domain of $\ell(p;)$, restart at step 2.
8. If $v = \epsilon$ and if u is not a prefix of a word in the range of $\ell(p;)$, restart at step 2.
9. Add $\langle u, v, p\rangle$ to D. Construct all the positions which can be reached in one move from $\langle u, v, p\rangle$ and add them on the right of L. Restart at step 2.

Since the exploration is breadth-first, it is easy to prove that if there is a dependence, then the algorithm will find it.

This algorithm has been implemented as a stand alone program in Objective Caml. The user has to supply the results of the analysis of the input program, including the control automaton, the address transductions, and the list of statements with their accesses. The program then computes the SDG. All examples in this paper have been processed by this pilot implementation. As far as my experience goes, the case where algorithm D does not terminate has never been encountered.

2.3 `sum` Revisited

Consider the problem of parallelizing the body of **`sum`**. There are already control dependences from statement 1 to 2, 3, and 4. The crucial point is to prove that there are no dependences from 2 to 3. One has to test one output dependence from `value[I]` to itself, two flow dependences from `value[I]` to `value[I.1]` and `value[I.2]`, and two symmetrical anti-dependences.

Let us consider for instance the problem of the flow dependence from `value[I]` to `value[I.1]`. The ℓ transduction begins in the following way:

$$\ell = (\langle 2,2\rangle + \langle 3,3\rangle)^*.\langle 3,2\rangle \ldots.$$

Algorithm D finds that there is no way of crossing the $\langle 3,2\rangle$ edge without generating distinct strings. Hence, there is no dependence.

On the other hand, algorithm D readily finds a dependence from 2 to 4. All in all, the SDG of **`sum`** is given by Fig. 4, to which corresponds the parallel program in Fig. 5 — a typical case of parallel divide-and-conquer.

3 Conclusion and Future Work

I have presented here a new framework in which to analyze recursive tree programs. The main differences between the present method and the more usual pointer alias analysis are:

- Data structures are restricted to trees, while in alias analysis, one has to determine the *shape* of the structures. This is a weakness of the present approach.

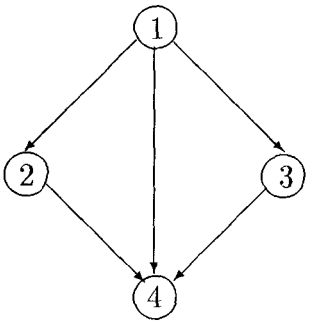

Fig. 4. The SDG of sum

```
void sum(address I)
{
1 : if(! leaf[I]) {
        {^
2 :       sum(I.1);
3 :       sum(I.2)
        ^};
4 :     value[I] =
              value[I.1] + value[I.2];
        }
}
```

Fig. 5. The parallel version of sum

- In $\mathcal{T}$, the operations on addresses are limited to postfixing, which, translated in the language of pointers, correspond to the usual pointer chasing.
- The analysis is operation oriented, meaning that addresses are associated to operations, not to statements. This allows to get more precise results when computing dependences.

Pointer alias analysis can be transcribed in the present formalism in the following way. Observe that, in the notations of Sect. 2.2, the iteration word x belongs to Domain(h), which is a regular language, hence w belongs to $f(\text{Domain}(h))$, which is also regular. This is the *region* associated to S_k. Similarly, w is in the region $g(\text{Range}(h))$. If the intersection of these two regions is empty, there is no dependence. When compared to the present approach, the advantage of alias analysis is that the regions can be computed or approximated without restrictions on the source program.

It is easy to prove that when $f(\text{Domain}(h)) \cap g(\text{Range}(h)) = \emptyset$ then ℓ is empty. It is a simple matter to test whether this is the case. The present implementation reports the number of cases which can be decided by testing for the emptiness of ℓ, and the number of cases where Algorithm D has to be used.

In a $\mathcal{T}$ implementation of the merge sort algorithm, there were 208 dependence tests. Of these, 24 were found to be actual dependences, 34 where solved by region intersection, and 150 required the use of algorithm D. While extrapolating from this example alone would be jumping at conclusions, it gives at least an indication of the relative power of the region intersection and of algorithm D. Incidentaly, the SDG of merge sort was found to be of the same shape as the SDG of sum, thus leading to another example of divide-and-conquer parallelism.

The $\mathcal{T}$ language as it stands clearly needs a sequential compiler, and a tool for the automatic construction of address relations. Some of the petty restrictions of Sect. 1.5 can probably be removed without endangering dependence analysis. For instance, having trees of structures or structure of trees poses no difficulty. Allowing trees and subtrees as arguments to functions would pose the usual

aliasing problems. A most useful extension would be to allow trees of arrays, as found for instance in some versions of the adaptive multigrid method.

How is $\mathcal{T}$ to be used? Is it to be another programming language, or is it better used as an intermediate representation when parallezing pointer programs as in C or ML or Java? The latter choice would raise the question of translating C (or a subset of C) to $\mathcal{T}$, i.e. translating pointer operations to address operations. Another problem is that $\mathcal{T}$ is static with respect to the underlying set of locations. It is not possible, for instance, to insert a cell in a list, or to graft a subtree to a tree. Is there a way of allowing that kind of operations?

Lastly, trees are only a subset of the data structures one encounter in practice. I envision two ways of dealing, e.g., with DAGs and cyclic graphs. Adding new address operators, for instance a prefix operator:

$$\pi(a_1 \ldots . a_n) = (a_1 \ldots . a_{n-1})$$

allows one to handle doubly linked lists and trees with an upward pointer. The other possibility is to use other mathematical structures as a substrate. Finitely presented monoids or groups come immediately to mind, but there might be others.

References

1. Jean Berstel. *Transductions and Context-Free Languages.* Teubner, Stuttgart, 1979.
2. C.C. Elgot and J.E. Mezei. On relations defined by generalized finite automata. *IBM J. of Research and Development*, 47–68, 1965.
3. Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data-Parallel Programming Model*, pages 79–103, Springer, 1996.
4. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag or a cyclic graph? In *PoPL'96*, pages 1–15, ACM, 1996.
5. Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general dependence test for dynamic, pointer based data structures. In *PLDI'94*, pages 218–229, ACM Sigplan, 1994.
6. Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Shereen Ghobrial, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. *Int. J. of Parallel Programming*, 25(4):305–338, August 1997.
7. Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93, LNCS 715*, pages 398–416, Springer-Verlag, 1993.
8. James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI'88*, pages 31–34, ACM Sigplan, 1988.
9. Rémi Triolet, François Irigoin, and Paul Feautrier. Automatic parallelization of FORTRAN programs in the presence of procedure calls. In Bernard Robinet and R. Wilhelm, editors, *ESOP 1986, LNCS 213*, Springer-Verlag, 1986.