2nd Year Progress Report:

Parallelisation of Data Structures via Static Dependency Analysis

Tim Lewis

April 13, 1999

1 Introduction

Producing programs that runs efficiently on parallel computers is a difficult task and research into the problem falls into two basic areas.

- Designing parallel languages that allow the programmer to express the parallelism in the program in a natural and flexible way.
- Designing automatic parallelising compilers/code analysers that take existing sequential codes and expose the parallelism in them, producing a parallel version.

My work primarily takes the second approach, but overlaps with the first, in that the programmer is expected to augment the sequential code with descriptions of the data structures that it uses.

Imperative languages, such as C, allow programmers to manipulate pointer-based structures. Thus a structure can contain a pointer to another structure of the same type, as one of its components. These data structures are often called recursive , since the definition depends on itself, and the structure is not limited in size. Automatic parallelisers can often not deal with such structures effectively since there is little restriction on where the pointers point to. Thus linkages between different data nodes are not known, and hence dependencies between different parts of a program are difficult to predict. Approaches to this specific problem fall into two camps, similar to the ones above.

- Structure language : A language for defining structural dependencies is defined.
- Fully automatic : The compiler infers dependency information only from the sequential information that defines it.

There are many methods in the literature that detail specification languages for describing the 'shape' of structures. Not all of them are designed with the goal of dependency analysis and hence parallelisation in mind.

My approach is to force the programmer to describe the linkages in the structure more closely by declaring where each pointer may point. If we want to describe a general graph structure (such as a network of roads) there may not be enough regularity in it to allow for a very full description. We may just have to say that any node can be linked to any other by any edge. At the other extreme we could have a binary tree where each node is joined by two pointers left and right to two separate sub-trees. Somewhere between these two extremes are structures with sufficient regularity that they can be described and useful parallelising information about programs that use them can be extracted.

The strength of this method is that it combines the following features.

- Specifications can be rigid, each pointer can only point to one possible node. This gives accurate dependencies for structures such as trees and lists.

- Flexible descriptions can be produced. A pointer can point to a number of possible nodes, the actual one only being decided for a particular run of the code. This allows a much larger range of possible structure to be handled (approximately) by these descriptions.

2 Background

We will first begin by looking at how dynamic data structures are handled in a language such as C, and then relate this to our approach.

Consider a tree data structure:

```
struct Tree {
    int value;
    Tree * l;
    Tree * r;
    Tree * n;
};
```

The items `value`, `l`, `r` and `n` are the *fields* of the structure, and may contain items of data (such as `value`) or pointers to other parts of the structure (such as `l`, `r` and `n`). Although this is not specified in the code, we assume that `l` and `r` point to two disjoint subtrees, and the `n` pointer links the leaf nodes of the tree in a linked list.

How is such a structure represented in our system? Firstly we have a fixed list of symbols (the *alphabet* A) that correspond to the fields in the structure. We now restrict slightly by having a subset $G \subseteq A$ of *generators*. These pointers form a tree skeleton for the structure, and every node in the structure is identified by a unique string of symbols (the *pathname*, a member of the set G^*), which is the path from the root of the tree to that particular node.

In addition our description of the structure contains a set of relations, $\rho_i \subseteq G \times G$, one for each non-generator direction i. This relation links nodes that are joined by a particular pointer. One node may be joined to more than one target node via a particular pointer. It is often useful to consider each relation as a function from pathnames to a set of pathnames: $F_i : G^* \to \mathcal{P}(G^*)$. A *word* is a string of directions: we append words to path names to produce new path names.

Currently each relation is represented as a *two-variable finite state automaton* (2FSA). A (deterministic) finite state automaton (FSA) is a simple model of computation, where an automaton reads a string of symbols one at a time and move from state to state. If it ends up in one of the accept states when the string is exhausted, then it is accepted. The automaton consists of a finite set of states S and a transition function $F : S \times A \to S$, which gives the next state for each of the possible input symbols in A. A *two-variable* FSA, differs in that the automaton is attempting to accept a pair of strings and inspects one symbol from each of them at each transition. It can be thought of as a one-variable FSA, but with the set of symbols extended to $A \times A$. There is one subtlety, in that we may want to accept strings of unequal length, so the shorter is padded with the additional '$-$' symbol. This makes the actual set of symbols $((A \cup \{-\}) \times (A \cup \{-\})) \backslash (-,-)$.

We also use 2FSAs to hold and manipulate the dependency information that we gather.

[ECH+92]

Dataflow information follows the exact flow of data in a program from *source*, where a value is defined (written) to the *sink*, where it is used (read). A *control word* is a label of a run-time instance of a program statement. For a nest of recursive functions with labelled statements, it is a string of labels from the statements that contain function calls.

3 Current Programming Model

I have been working with analysing a fairly restricted programming language that manipulates these structures. This approach seems common for research in this area, see [CCG96a] and [CC96], although some work with fairly unrestricted versions of languages such as C, for example in [HHN94]. To illustrate the description there is a fragment of typical code in Fig. 1

The code works with one global data structure. Data elements are accessed via pathnames, which are

```
main (Tree *v) {
  Init(v->l);
  Traverse(v->l);
}

Init(Tree *i) {
  if (i==NULL) {return;}
  i->l = i->l;
  i->r = i->r;
  Init(i->l);
  Init(i->r);
}

Traverse(Tree *w) {
  if (w == NULL) {return;}
  Update(w->l, w->r);
  w->data = w->p->data;
  Traverse(w->r);
}

Update(Tree *t, Tree *a) {
  if (t == NULL) {return;}
  t->data = a->p->l->n->data;
  if (t->l != NULL) {Update(t->l, a->r);Update(t->r, a->l);}
  return;
}
```

Figure 1: A fragment of code from my restricted language

used in the same manner as conventional pointers. The code consists of a number of possibly mutually recursive functions. These functions take any number pathname parameters, and return `void`. It is assumed that the first (`main`) function is called with the root path name. Each function may make possibly recursive calls to other functions using the syntax '`Func(w->g)`' where `g` is any generator.

It should be made clear that in order to perform our analysis, the code can not recurse in a direction that is not a generator; otherwise the construction of the map from control words to pathnames ceases to be representable by a regular language. This is a restriction of the method. We can still produce some information, however, by *unrolling* the recursion, and analysing within the function.

There is a related problem if a function calls another one with the parameter `w`, *e.g.* '`FuncName(w)`'. We can handle this provided it does not occur within a mutually recursive set of functions, *i.e.* the function can only be called a finite number of times.

The basic statements of the code are reads and writes to various parts of the structure. A typical read/write statement is '`w->a = w->b`' where `w` is a variable and `a` and `b` are words of directions, and denotes the copying of an item of data from `w->b` to `w->a`, within the structure.

One point should be made about these static control programs: if we do not allow some run-time dependencies in the code, we may as well run the program and trace its dataflow directly. Therefore we had better be able to deal with variable sized structures. We allow conditional statements of the form '`if (pathname == NULL) then ...`' or '`if (pathname != NULL) then ...`'. This restricts the type of analysis that can be done. For example, any dataflow analysis has the potential to become inaccurate when we cannot decide statically if a statement is executed or not. We can still find sets of nodes accessed by a particular statement and deduce coarser-grained parallelism.

4 Extending the Model

There are a number of ways that we can extend the restricted programming language. Some of these increase the power of the language to express particular algorithms, others add to the expressibility of the language for the programmer.

'`if ... then ... else ...`' constructs Currently we ignore conditionals and produce approximate information. If we can predict at compile-time whether a statement will be executed, we want to be able to use that information. For example, we can guarantee that a statement with no enclosing `if` clause will be executed. Similarly, we can always predict that one of the `then` or `else` blocks will be executed. This relates to the the 'fuzzy' approach given in [CC96]. Here the authors deal with the problem of statements that may or may not be executed at run-time, in the context of attempting dataflow analysis. Although the exact source of a particular item of data may not be possible to find they can trim the set of potential sources. This is done by identifying potential sources that will be overwritten before the item of data is next read. We can apply this kind of approach, by deriving information indicating that the execution of a statement can imply that certain other statements must have been executed previously. Certain more coarse grained parallelisation techniques, (e.g. *function call* parallelism, see [GHZ98]) do not need full dataflow information, however, so some dependency analysis can still be done.

Control Structures Currently we only consider a set of possibly recursive functions, we may want to extend this to `for`/`while` loops. This requires us to add loop counters to the control words that uniquely determine the run-time instances of statements.

Pointers Our use of pathnames can be thought of as a more disciplined version of pointers, so allowing conventional pointers in the language seems unnecessary. We do not intend to allow pointer arithmetic.

Pointer Assignments So far we have only considered code that reads and writes into data values of a structure. Structures can change dynamically by assigning to particular pointers. If we want to allow this we have to consider the two possibilities.

- For a non-generator direction, check that the assignment does not invalidate the given description.
- For a generator direction, assume that all the child data values have been copied over by that assignment.

For the first case we would like to be able to take into account the new information about where the pointer points. In the second we may want to disallow such assignments since they contradict the no-sharing property of the generator directions.

Multiple structures Currently we only consider one global data structure. We can extend this to multiple structures by combining them into one structure that we treat in the current manner. There is the problem of efficiency with this method, as it increases the number of generator directions. The complexity of the FSA manipulations are sensitive to the size of the underlying alphabet. We need to find methods of handling this without increasing the alphabet size.

Function Types We will want to extend to allow functions that return values. For arguments/return values that are scalar data items (int, real, *etc* ...) this is straightforward, we can augment the global structure so that the data can be passed through it.

Arrays To include arrays in our model, we need to include much the work that has been done for dataflow analysis of arrays, see [CCG96b] for an analysis of recursive functions over arrays. This is worth doing, but would require a considerable amount of work.

Recursion in non-generator directions This awkward restriction reduces the ability to express algorithms. Relaxing this would require us to use a more general model for holding the dependency information. Candidates for this model would be languages given by context-free grammars or algebraic transducers. Alternatively, we could use the existing system, and take a regular set as an approximation for the true non-regular set. Whether such approximations are good enough to extract useful information may be a worthwhile area of research.

5 General Approach

The basic system for the parallelisation of programs is as follows. In addition to the code which conforms to the programming model in section 3, the programmer supplies a detailed description of the linkages within that structures that it uses. These descriptions may be supplied in a number of different specification languages. The description will be converted into an underlying Automatic Group (AG) description, and sections of the code can then be analysed, producing dependency or dataflow information. This can then be inspected to find independent sections of code.

5.1 Definitions

Static Control Programs Code where the exact control flow can be determined at compile time.

6 Related work

There are a number of existing description languages for complex data structures.

ASAP The description consists of a set of axioms that describe when nodes can be aliased or are disjoint. The axioms are based on regular expressions in the direction symbols. See section 8.2 for a fuller description of the axioms, and how they can be converted to AG descriptions.

Graph Types The structure is a tree with additional routing pointers, defined by expressions.

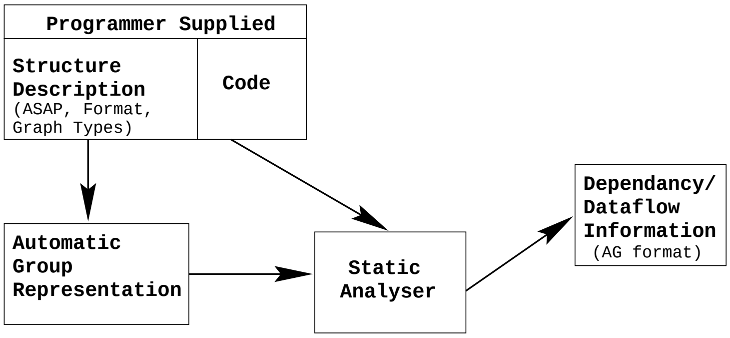


Figure 2: Approach

The routing expressions are regular expressions over the directions, with additional terms that allow the expressions to test the type of the current node.

Group Fields The group is described by a *presentation*, a list of equations that summarise all the possible cycles in its graph. This is a very general description and may include structures where we cannot determine whether two nodes are linked by a particular direction (a situation related to the *word problem*. Analysis of such structures would have to work on a restricted subset of these groups.

Shape Types The structure is represented by a list of triples, where a pointer name precedes the pair of nodes that it links. A grammar gives the structure of the list. Certain questions that we can answer cleanly for Automatic Groups may not be readily solved for these structures, since the context-free languages are more general than regular languages.

Analysing recursive functions within imperative programs has also been considered.

Recursive Tree Programs [Fea98] This work looks at dependencies in a class of programs using *rational transducers* as the framework. These are similar, but more general than the two-variable FSAs we use here. The generality implies that certain manipulations produce problems that are undecidable, and a semi-algorithm is proposed as a partial solution. Also the only data structures considered are trees, although extension to doubly linked lists and trees with upward pointers is hinted at. However, even programs that operate over relatively simple structures like trees can have complex dependency patterns. My method can be used to produce the same information from these programs. This indicates a wider applicability of my approach from the one involving structures with complex linkages.

Algebraic Transductions [CC98] This paper looks at the problem of analysing recursive functions over arrays. They use the larger class of language, context-free languages as descriptions of the dependencies that are discovered. Even so, they conclude that they will require an even larger class of language (possibly an indexed grammar) to store all the information they wish. Handling multidimensional arrays will also require a more sophisticated language. Although arrays can be described using our method, our method does not handle them too well, mainly because we have to restrict the ways that the code is allowed to recurse through them. See the restrictions mentioned in section 3.

7 Detailed Model

Here the method is discussed, followed by a description of the simple programming model whose programs have been analysed.

To illustrate the descriptions we use the example of a binary tree, see Fig. 3. In this tree is binary, each node has a left and right subtree and the nodes in each level are joined by a doubly linked list. The 2FSA description of this tree is given in Fig. 4.

2FSAs are a good choice of representation, since they can be readily manipulated using existing regular language theory. In particular we can produce FSAs for words of directions from the individual FSAs for each direction. We can also perform basic operations such as AND, OR, NOT, $\forall$ and $\exists$ with the FSAs involved and hence with the sets of control words.

These descriptions can be more accurate than the ASAP axioms, since they define a regular set of pairs of nodes, rather than two regular sets of nodes. This gives more control over which nodes are linked, but still allows some flexibility, since one pointer can point to a number of different nodes.

8 Enhancements

This is a description of the ways in which we have enhanced the method to handle additional information.

8.1 Run-time leniency

Strictly speaking, the AG definition in [ECH$^+$92] would not allow a particular node to be linked to any one of a set of target nodes. This sort of property is useful if there is some sort of run-time flexibility about precisely what a pointer links to, but there is a some restriction on the set of possible target nodes. However, since the formalism does allow multiple path names for the same node, we can link one node to many others. We just choose to interpret this as a number of different nodes rather than many names for the same node.

8.2 Aliasing and disjointness, with ASAP

In the ASAP method, linkage information in a structure is expressed via a series of axioms. The axioms come in two basic types, either describing when nodes are equal (*aliased*) or unequal (*disjoint*). We augment the existing 2FSA structure description with two further 2FSAs, $\mathcal{F}_a$ to hold the aliasing information and $\mathcal{F}_d$ to hold the disjointness information. We can then combine the description with $\mathcal{F}_a$ and $\mathcal{F}_d$ at appropriate points in the analysis.

There are three kinds of axiom in the ASAP framework.

$$\begin{array}{ll} 1) & \forall p \quad p.RE1 \neq p.RE2 \\ 2) & \forall p \neq q \quad p.RE1 \neq q.RE2 \\ 3) & \forall p \quad p.RE1 = p.RE2 \end{array}$$

Where p, q are any path names and $RE1, RE2$ are regular expressions over A^* . Given a regular expression $RE1$ it is trivial to convert it to a two variable 'appended' FSA denoted Ap_{RE1}, that accepts $(p.RE1, p)$ for all p, . Additionally, using $\mathcal{F}_{\neq}$ the FSA that accepts a pair of paths if they are not equal, we can give expressions that convert axioms of each type to FSAs that accept the equivalent pairs of words. The required equations are as follows.

$$\begin{array}{ll} 1) & \mathrm{Ap}_{RE1}.(\mathrm{Ap}_{RE2}^{-1}) \\ 2) & \mathrm{Ap}_{RE1}.F_{\neq}.(\mathrm{Ap}_{RE2}^{-1}) \\ 3) & \mathrm{Ap}_{RE1}.(\mathrm{Ap}_{RE2}^{-1}) \end{array}$$

We then build $\mathcal{F}_d$ by ORing together all the FSAs produced from axioms 1 and 2, and $\mathcal{F}_a$ by ORing the ones from axiom 3. We can thus automatically generate $\mathcal{F}_a$ and $\mathcal{F}_d$ from ASAP axioms, and thus combine these descriptions with the bare AG ones.

It should be noted that the FSAs for some sets of axioms can get quite large, for example the axioms for a sparse matrix produce a $\mathcal{F}_d$ of approximately 150 states. This may be too large for much further analysis of the structure to be completed.

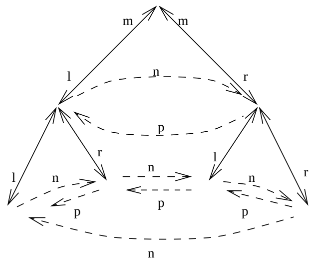


Figure 3: A binary linked tree structure.

8.3 Type information

As mentioned previously the AG formalism allows for a *word acceptor* automaton W that accepts only a regular subset of all possible path names as being legal. This can be used to restrict certain pointers from certain locations, for example we may not want the root node of a tree to have a parent pointer, although all the other nodes will need one. However, we can use this word acceptor automaton to allow different types of nodes. Rather than have a homogeneous structure, where each node must have the same number of outgoing pointers, we allow each node to have a type that determines the number and names of such pointers. Thus we can nest the structures, for example to produce a list of trees, built from descriptions of a tree and list.

9 Overview of Analysis

We take as input an 2FSA description of a structure and a fragment of code as described previously. We can then create a number of FSAs that describe certain properties of the program.

1. The *definition* FSA, that accepts the pair (control word, path-name) if the control word writes (or defines) that node of the structure.
2. The *use* FSA for describing nodes that are read by a particular statement.

We can combine these FSAs to produce the *conflict* FSA. This is a description that links each read statement to a set of potential sources. We can simplify the conflict FSA further by removing potential sources that occur after the sink. For more technical details see [LA98].

Since we cannot determine which statements will actually be executed we cannot in general refine this to produce more accurate dataflow information.

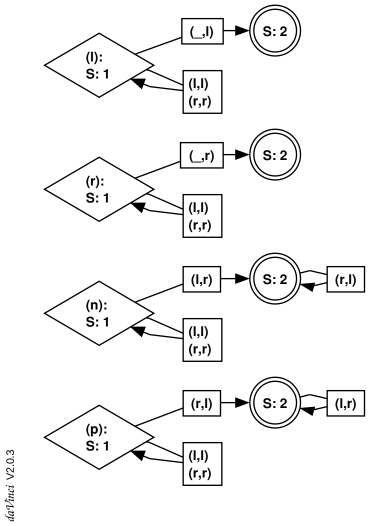
We can also use this pair to create FSAs for the different variants of dependency, read after write, write after write, *etc* ...

10 Parallelisations

Once we have gleaned information from these programs, we need to consider the different kinds of automatic parallelisation that may be possible.

Function Call Parallelism Two or more function calls are distributed to separate processors. This requires the analyser to spot when the data read and written by each call does not clash. We can produce FSA descriptions of the data accessed by each function call. Note that this information may be inaccurate, since there may be conditional data accesses that depend on run-time information. However, it will be *conservative*, in the sense that the actual set of data will be a subset of that computed. This is the type of parallelism that is being extracted in [Fea98], using the EARTH-C language.

Fine Grained Parallelism Individual statements are scattered across processors. This requires




```
Generator l, r;
Link n, p;
{l:  }
{r:  }
{n:
  States 2;    Start 1;    Accept 2;
  Transitions[
              [(l,l)->1;(r,r)->1;(l,r)->2;]       //From state 1
              [(r,l)->2;]                         //From state 2
  ]
}
{p: (n)^}
```

Figure 4: Description of a binary linked tree. The rhombus nodes are start nodes and are labelled with the direction name e.g. (l). Other states are circles and are labelled with state number e.g. $S : 2$. The boxes hold the pair of symbols for that transition. Double borders indicate an accept state. Below is the source code version of the description.

knowledge about the dataflow dependencies between different statements. We can produce this information for restricted programs with static control, any improvement will require the *fuzzy* approach given in [CC96].

The 'function call' approach has been explored using a parallel language called 'Cilk' [Lei]. Cilk is a multi-threaded language based on ANSI C. It adds the keywords `spawn` and `sync`, for spawning and synchronising parallel function calls (threads). The threads operate in shared memory, so spawned function calls must operate on discrete sets of data, or the outcome of the computation will be uncertain and probably undesirable.

Cilk comes with a tool called the 'Nondeterminator' which, given a cilk program and the data on which it operates, tests for possible data conflicts in spawned threads. The answers received are approximate, for the given data set.

Our aim has been to decide where, if anywhere, we can insert `spawn` and `sync` commands, given a program in our reduced language. This is solved when we can produce descriptions of all the data nodes accessed by a particular function call. This is an amalgam of the *def* and *use* FSAs described previously. Initial results for the code analysed so far show that there is frequently some dependency between function calls from the same function. A more feasible future approach is to unroll the functions to a particular depth and look for independent calls from more than one function.

11 Vortex method of fluid flow simulation

This section follows through the analysis of part of the algorithm from the 'Fast Multipole Method' for solving fluid flow problems, see [Pri94]. The data structure used is a tree, with l and r pointers at each node. We also have additional pointers n and p that link nodes at each level into a linked list, and a s *spiral* pointer that joins the nodes in a spiral from bottom left of the tree to the root.

The nodes carry four items of data *psi*, *phi*, *theta* and *gamma*, all of which are multipole expansions of the potential due to various sets of nodes. The code shown in Fig. 5 has been simplified in two important ways.

- The summations represent more complex manipulations of these expansions.
- The code is a 'one-dimensional' version of the algorithm. In the two dimensional version there are four subtrees at each node, and the nodes in each level are linked into a grid, rather than a list.

If we analyse the function `PhaseTwo`, we get the FSA in Fig. 6, for the potential sources of statements. This shows that the reads that occur in statement B are produced by the write in statement A. Also the read in statement A uses the value produced in the previous function call.

We need a different approach to produce information for the function 'PhaseOne' since it recurses in a non-generator direction. We have 'unrolled' three of the function calls into the body of the loop, and have looked for dependencies within one call, rather than between all calls of the function. The dependency FSA produced in given in Fig. 8. The main observation is that there are only dependencies between calls that access the first three levels of the tree. In all other levels the four statements are independent and can therefore be executed simultaneously.

12 Update from Thesis Proposal

We shall look at my current position in the light of the further work suggested in last years Thesis Proposal.

Use of Markov algorithms

This approach has not been pursued for a number of reasons.

- The descriptions were difficult to manipulate, Markov algorithms can be composed, but the re-

```
FindLeftmostLeaf(w) {

   if (w->l->l == NULL) PhaseTwo(w);
   FindLeftLeaf(w->l);
}

PhaseOne(w) {

   Struct(w->phi) = Struct(w->l->phi) + Struct(w->r->phi);
   if (w->m == NULL)
   PhaseOne(w->s);
}

PhaseTwo(w) {

    if (w == NULL) return;
  A: Struct(w->psi) = Struct(w->m->psi)
     + Struct(w->m->p->l->phi) + Struct(w->m->p->r->phi)
     + Struct(w->m->n->l->phi) + Struct(w->m->n->r->phi);

  B: if (w->l == NULL) Struct(w->theta) = Struct(w->psi)
                    + Struct(w->p->gamma) + Struct(w->n->gamma);
  C: PhaseTwo(w->l);
  D: PhaseTwo(w->r);
}
```

Figure 5: Algorithm for fast multipole method

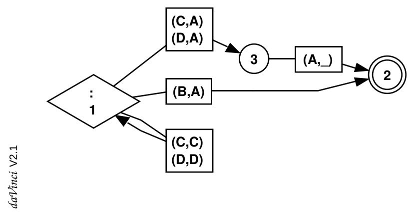


Figure 6: FSA for dependency of the second phase of fluid simulation algorithm

```
PhaseOne(w) {

   A: Struct(w->phi) = Struct(w->l->phi) + Struct(w->r->phi);
      if (w->m == NULL) return;
   B: Struct(w->s->phi) = Struct(w->s->l->phi) + Struct(w->s->r->phi);
      if (w->s->m == NULL) return;
   C: Struct(w->s->s->phi) = Struct(w->s->sl->phi) + Struct(w->s->s->r->phi);
      if (w->s->s->m == NULL) return;
   D: Struct(w->s->s->s->phi) = Struct(w->s->s->s->l->phi) + Struct(w->s->s->s->r->phi);
      if (w->s->s->s->m == NULL) return;
      PhaseOne(w->s->s->s->s);
}
```

Figure 7: PhaseOne of the algorithm, unrolled

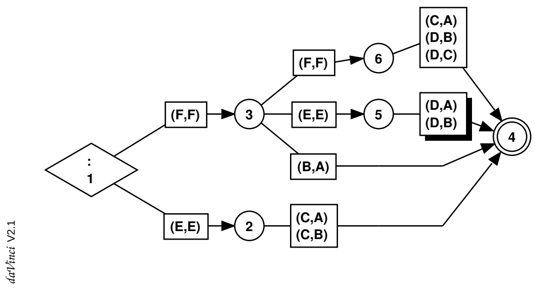


Figure 8: FSA for dependencies within one function call of the unrolled PhaseOne function

sulting algorithm is far more complex, and does not simplify as readily as FSAs.

- Converting existing specification languages into Markov algorithms is far from straightforward.

Use of Format descriptions

An earlier approach used a description based on a simplified regular expression to describe the linkages. These 'Format' expressions are not as powerful as the Automatic Group approach. However, the Format system is a more intuitive method for describing structures, so it would be worthwhile to be able to convert automatically from this description to an underlying AG description.

Generalise data structures

- *"Extend dataflow analysis to Group fields and Graph Types frameworks. These methods will need some modification, since the graph types system does allow structures with runtime dependencies, and the group approach is possibly too general to allow accurate analysis."*

The AG approach can handle structures with the kind of run-time dependencies that Graph Types structures can have. This works because these dependencies involve the different type of each node in the structure. Once the type of each node can be encoded in the path-name, we can express the appropriate properties, see section 8.3.

The Group Fields framework [Mic95] used a presentation description that would have been difficult to work with. It would allow structures with an undecidable word problem: we could not decide that two links led to the same node. It also uses a novel declarative method of describing algorithms, based on *webs* and *streams*, rather than a straightforward imperative language.

Using groups could require us to describe structures too strictly to be of any use, but the AG formalism can enable us to produce more vague descriptions, such as the target of a particular pointer can be *any one* of a set of nodes.

- *"Extend approach to a collection of properties,* e.g *commuting, inverse, parent,* etc ..., *allowing them to be combined to produce more general families of structures."*

This idea is feasible within the AG method. As already mentioned, describing structures directly using FSAs can be unwieldy, and this could be an approach that would make it more manageable. These particular properties mentioned above can all be converted automatically to FSA descriptions.

Extend program model by allowing general predicates on the recursive calls.

- *"If we do not want to restrict the form of such predicates, or allow them to be runtime dependent, follow the 'fuzzy' approach."*

This was discussed earlier in section 4.

- *"As with the Group Fields approach, predefine a subset of the potentially infinite structure for the program to operate on."*
- *"Use recursive* `constructs` *as in [CCG96a] to build up a the structure first. The functions then only operate over the values that have been defined."*

The current approach is closer to the way that recursive structures in C are handled normally. The description allows the structure to be thought of as infinite, but an actual structure is bounded by `NULL` pointers. This allows structures of irregular shape and dynamically growing and shrinking structures to be included.

13 Summary of work already completed

- Created a simple dataflow/dependency analyser for structures that are given in a 2FSA description. The code handled is a restricted version of C, described in section 3.

- Produced information from code that can be used to convert to a Cilk program. There is the problem that for the codes looked at, there does not seem to be many independent function calls.

- Considered compatibility with existing solutions, *e.g.* integration of ASAP information into the AG framework.

14 The proposed contents of the Thesis

1. Introduction: Explanation of the problem,
2. Literature Survey: Related Work
   (a) Dependency Analysis
   (b) Methods of specifying complex structures
   (c) Methods of analysing codes that use structures
3. AG method of describing structures. Dependency analysis of simple programs.
4. Integrating other specification languages: Using AG as an underlying representation, converting other specification languages into AG. Extending AG to accommodate these other systems.
5. Dependency Analysis
   (a) Extending program model to more realistic codes
6. Parallelising Code: Using extracted dependency information to produce parallel programs.
7. Practical Applications: Applying methods to real application codes. Vortex fluid flow simulations and others.

15 Work remaining

- Integrate, as far as is possible, Graph Types and Shape Types descriptions into the AG framework. Investigate using simpler description languages, and extracting information automatically from programs, as in [HS97].

- Extend program model to handle real codes. At the moment, actual codes have to be massaged into a form that the analyser can work with. Also the toy language is too restricted to cope with many of the program features available in, for example, ANSI C. Possible extensions were mentioned in section 4.

- Use information from analyser to parallelise codes. This will involve automatically generating code in some parallel language. Doing this for Cilk has already been investigated. A target language for a more fine-grained parallelism needs to be explored. A suitable language may be found amongst the 'Reactive Programming' languages, such as Esterel or Reactive C.

- Apply method to real world problems. N-body gravity simulations and vortex fluid flow are example of suitable real-world applications that use sufficiently complex data structures. In addition we can apply these methods to simpler structures such as tree or list algorithms.

References

[CC96] Albert Cohen and Jean-Francois Collard. Fuzzy array data-flow analysis – part II: Recursive programs. Technical Report 96-036, PRiSM, `http://www.prism.uvsq.fr`, December 1996.

[CC98] A. Cohen and J.-F. Collard. Applicability of algebraic transductions to data-flow analysis. Technical Report 9, PRiSM, U. of Versailles, January 1998.

[CCG96a] A. Cohen, J.-F. Collard, and M. Griebl. Data flow analysis of recursive structures. Technical Report 96/018, Laboratory PRiSM, University of Versailles, France, September 1996.

[CCG96b] Albert Cohen, Jean-Francois Collard, and Martin Griebl. Array data-flow analysis for imperative recursive programs. Technical Report 96/035, Laboratory PRiSM, University of Versailles, France, 1996.

[ECH+92] David B. A. Epstein, J. W. Cannon, D. E. Holt, S. V. F. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups*. Jones and Bartlett, 1992.

[Fea98] Paul Feautrier. A parallelization framework for recursive tree programs. Technical report, Laboratoire PRiSM, May 1998. Basically uses FSAs to analyse recursive tree programs. Only structures considered are trees.

[GHZ98] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in C programs with recursive data structures. In *Proceedings of the 1998 International Conference on Compiler Construction*, March 1998.

[HHN94] Joseph Hummel, Laurie J Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[HS97] Yuan-Shin Hwang and Joel Saltz. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In Z. Li et al, editor, *Languages and Compilers for Parallel Computing, 10th International Workshop LCPC'97*, number 1366 in Lecture Notes in Computer Science, pages 131–145, Minneapolis, Minnesota, USA, August 1997. Springer-Verlag.

[LA98] Tim Lewis and D. K. Arvind. Dependency analysis of recursive data structures using automatic groups. In Siddhartha Chatterjee et al, editor, *Languages and Compilers for Parallel Computing, 11th International Workshop LCPC'98*, Lecture Notes in Computer Science, Chapel Hill, North Carolina, USA, August 1998. Springer-Verlag.

[Lei] Charles E. Leiserson. The Cilk Project.

[Mic95] Olivier Michel. Design and implementation of 81/2, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, December 1995.

[Pri94] Gavin J. Pringle. *Numerical Study of Three-Dimensional Flow using Fast Parallel Particle Algorithms*. PhD thesis, Department of Mathematics, Napier University, Edinburgh, February 1994.