Detection of Sequential Modules in Parallel Programs

M. Korsloot — Delft University of Technology
E. Tick — University of Oregon

CIS-TR-91-09
March 1991

Abstract

A compile-time technique is presented for determining if a set of procedures within a parallel program can be executed sequentially without causing deadlock. The analysis and methods are described for committed-choice parallel logic programming languages; however, the concepts are general enough for any concurrent languages with fine-grain communicating processes. We derive methods for ensuring that sequential evaluation of a program module cannot result in producer-consumer suspension within the module itself, thereby resulting in deadlock. The advantages of sequentializing fine-grain languages include the use of all "traditional" compiler optimizations, such as global register allocation, and continuation-stacking procedure invocation.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Contents

List of Figures

List of Tables

1 Introduction

"Theories can be classified in four categories: superb, useful, tentative and misguided."

R. Penrose [11]

Traditional parallel procedural languages evolved from sequential programming languages. The quest to uncover more parallelism, in more efficient ways, is paramount in the development of these languages. Alternatively, concurrent languages, such as committed-choice parallel logic programming languages, have a great deal of inherent parallelism. These languages and their implementations have been refined to exploit the parallelism more and more efficiently, with increasingly sophisticated interpreters and emulators, in both software, firmware, and hardware. Less research has been done concerning efficient compilation for the parallel execution of these concurrent languages. It is often the case that by sequentializing portions of a fine-grain parallel program, execution time decreases and processor utilization increases. The trick is to determine which sets of procedures should be serialized for overall benefit, and if they can be serialized safely, i.e., without chance of deadlock.

In this paper, we introduce a method to safely sequentialize pieces of concurrent programs, with the intention of increasing execution speed. Specifically, we describe our method with respect to the family of flat committed-choice parallel logic programming languages, such as FCP, FGHC, and Parlog [13]. To achieve serialization, we combine a general mode-analysis algorithm [18] with a goal-ordering algorithm described here. We fully describe the conditions under which the analysis can successfully sequentialize a program, and indicate practical uses of the technique, such as global register allocation and continuation-based goal management.

There are several reasons why sequential execution can be beneficial. One of the advantages of sequential code blocks is to increase granularity [5, 15]. An often-mentioned problem of committed-choice languages is their small average granularity, causing a high overhead for process management and an abundance of light-weight processes. Apart from this, sequentializing part of a program also makes it possible to use many well-known compiler optimization techniques (e.g., [1]), such as a global or interprocedural register allocation scheme.

Our ultimate goal is to exploit efficient memory management, optimal register allocation, and destructive variable assignment techniques within "sequential" program modules. For now, we envision the programmer defining the modules and their import/export procedures. Thus we can compile parts of committed-choice programs as if they were sequential, procedural languages, thereby increasing the average granularity. This is a form of bilingual programming, much like Strand [6] wherein the sub-language is in fact Strand itself. By judicious modularization of a program, overall efficiency of committed-choice programs will improve significantly with relatively simple analysis.

This paper is organized as follows. A brief review of committed-choice languages and argument modes is given in Section 2. Section 3 describes a first attempt at efficiently implementing committed-choice languages by exploiting directional programs, as first discussed by Gregory [8]. Section 4 describes a more general method of mode analysis, due to Ueda [18]. The results of this mode analysis can be used to ensure safe sequential execution, as described in Section 5. Section 6 summarizes how the preceding methods can serve as the basis for the efficient implementation of sequential modules.

2 Background and Terminology

In this section a brief review of committed-choice languages and argument modes is given. The terminology introduced here, for *input and output modes*, *modules*, *basic blocks*, and *internal and external suspension*, is necessary for the remainder of the paper. In the literature, the notion of modes is somewhat overloaded in its meaning, through sloppy usage and intuitive bias. We hope to clarify the intended meanings. It is important to stress that we limit ourselves to the family of flat committed-choice languages [13]. This limitation is not a severe handicap, as the expressive power of these languages is comparable[1] to those languages with deep guards [14]; however, their implementation is simpler and more streamlined.

In flat committed-choice languages, Horn clauses have the form

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n.$$

where m and n are zero or positive integers. H is the clause head, G_i is a guard goal, and B_i is a body goal. A *goal* can be considered a procedure invocation. A *conjunction* of goals is simply a set of goals, appearing within the same clause. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). For flat languages, the guard goals can only be built-in predicates, such as `integer(X)`, or `X>0`.

Both the clause head and internal goals can have arguments. The head arguments correspond to formal parameters in a procedure definition, whereas the guard and body goal arguments correspond to passed parameters in a procedure call. A *procedure* is defined as the set of clauses having the same name and number of head arguments.

[1]See Shapiro [12] for an in-depth comparison of this family of languages.

An important concept is the notion of *modes*. A procedure argument is used to communicate values between the caller and callee. Intuitively, when the caller passes a value into the callee, the corresponding argument is used in an input mode. Similarly, when the callee passes a value back to the caller, the argument is used in an output mode. However, due to the nature of the logical variable, a single argument, in a given invocation, can be used both to pass values in and return values back. One example of this is called "incomplete messages," e.g., `m(X)` is an input message containing an unbound variable `X`, meant to be bound with a return message by the callee.[2]

Mode declarations can be either explicit (as in Parlog [2]), or implicit. A mode can be either *input* or *output*. In Parlog, the user must declare the modes for each "top-level" argument of a predicate. A top-level argument is the outermost term passed to a procedure, distinct from any subterms that it may be composed of. The following definitions are due to Gregory [8] and others.

> *Definition*: An *input argument* of a goal, denoted here by mode '?', is an argument which is either instantiated when the goal is called, or it is a variable. However, during head matching or guard evaluation, when the input argument is a variable, it can never be instantiated by the head-matching process. Basically, only input matching is performed. □

> *Definition*: An *output argument*, denoted here by mode '^', is an argument which is always unbound in the caller, and whose value is bound by the callee. □

In the following, arguments of a predicate which have been declared as input or output will be named *input positions* or *output positions*, respectively. Furthermore, the goal(s) which instantiate a variable are called the *producer(s)* of this variable, while the goal(s) which use, but not instantiate, a variable are called the *consumer(s)* of that variable.

Parlog has a fixed mode per argument per procedure. Other committed-choice languages are more flexible, allowing the same argument position to have a *different* mode per clause (e.g., FGHC [16]). Others are still more flexible, allowing the same argument position to have a different mode per procedure invocation (e.g., FCP [13]). We do not think this flexibility is sufficiently utilized in practice to warrant its inclusion in the language definition. Therefore we consider only fixed modes in this paper.

There are some flaws, however, in the simple mode system. Primarily, the modes concern only top-level functors. For example, consider the following program:

```
mode f(?,^).
f([X|Xs], Z) :- true | g(X),h(Xs,Z).
```

[2]Foster likens this to enclosing a self-addressed stamped envelop within a letter [7].

The mode declaration of the first argument does not convey anything about the value of X when f/2 is called: it only specifies that the first argument of the caller must be a (non-empty) list. Furthermore, specifying an argument as input does not exclude the possibility that the argument is bound to a non-variable term in the body, e.g.,

```
mode f(?).
f(X) :- true | X=[].
```

The above is a legal Parlog program, and when called with the query "?- f(X)," X will be bound to [], which contradicts the intuitive notion of input argument.

The notions of input and output modes, as introduced by Ueda [18], are somewhat more consistent in this regard. Instead of simply specifying the top-level arguments, Ueda's method attempts to infer the modes of *all* variables and structures occurring in the clause. Ueda distinguishes terms by specifying the *path* which must be taken to get to a specific term. For example, the path to get to variable X in predicate f/2 above is specified as <f,1><.,1>,[3] i.e., the head of the first argument of f/2.

The *value* of a path is then defined as the first (principal) functor following this path when the term at the end of the path is instantiated. In our previous example, if X is instantiated to a(0,0), then the value of the path <f,1><.,1> is a/2. We now give definitions of input and output path modes due to Ueda [17].

> *Definition*: If a path is defined as *input*, the value (i.e., the principal functor) of this path *may* (but need not) be bound by the caller, and *will never* be bound by the callee. □

> *Definition*: If a path is defined as *output*, the value (i.e., the principal functor) of this path *will never* be bound by the caller, and *may* (but need not) be bound by the callee. □

We now define the notions of a *sequential module* and a *sequential basic block*. We use the term *module* to describe a set of self-contained predicates, which is entered (called) through a single entry-point (the *module entry-point*). In this context, "self-contained" means that no predicate inside the module calls any predicates outside the module, nor does any predicate outside the module call any predicate inside the module, apart from the visible module entry-point. For example, consider the following code segment:

```
a :- b,c,d.
b :- e,b.
c :- b.
d.
e.
```

[3]For lists, the functor ./2 is used, so <.,1> is the head of a list, while <.,2> is the tail of a list.

Predicates `b` and `e` comprise a module; this module is entered only through `b`. From within this module, there are no calls to predicates outside it. Extending this notion, a complete program can be viewed as a module with the top-level query calling the module entry-point (assuming that the query consists of a single goal only). If the goals inside a module are executed sequentially, the module is called a *sequential module*, for obvious reasons.

A *basic block* is defined simply as a group of body goals within the same clause. Note that this definition is more general than the standard definition for machine instructions [1], although the intention is similar. If these body goals are executed sequentially, the block is called a *sequential basic block*. Grouping together these body goals can be done in several ways. First, the goals can be folded together (either by the user or as a source-level transformation by the compiler) into another goal, which is specifically marked as "sequential." For example, in the previous code, if we group the goals `c` and `d`, the program could be transformed into:

```
:- sequential g.
a :- b,g.
b :- e,b.
c :- b.
d.
e.
g :- c,d.
```

A second possibility for grouping a set of body goals is using an operator such as the sequential conjunction operator '&' from Parlog:

```
a :- b, (c & d).
```

In the third option for marking a sequential basic block, no explicit transformations or annotations are made, but rather the compiler detects the presence of a sequential basic block, and implicitly transforms the block into an internal representation.

Extending these notions with respect to the call graph of a program, a module is defined as a subgraph of the entire graph. This subgraph is entered through only one node,[4] and has no edges to nodes outside the subgraph. In contrast, a basic block is a set of nodes which are all connected to the same parent. Recall that this implies that all goals in a clause can be executed concurrently. An example is shown in Figure 1a, which represents the aforementioned program. The left box represents a module, and the right box represents a basic block.

By sequentializing a basic block, the call graph changes as follows: if a basic block consists of n goals, for all but the first goal, the edges connecting them to their parent

[4]There may be multiple edges entering a module, but they can be connected only to the module entry-point.

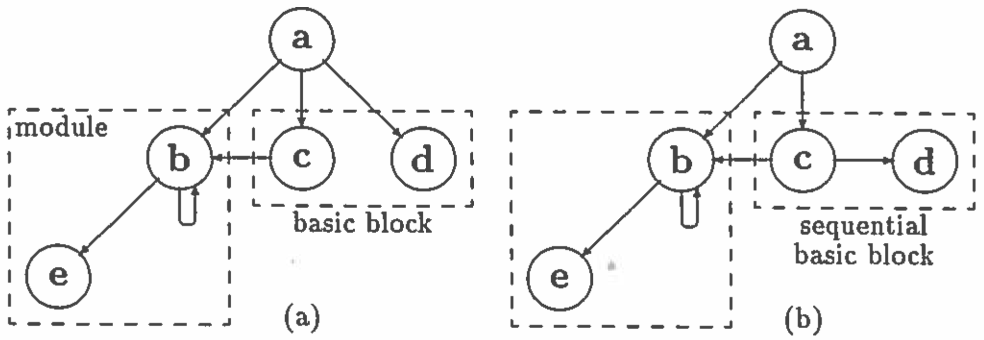

Figure 1: (a) Modules and Basic Blocks in Call Graphs. (b) A Call Graph with a Sequentialized Basic Block.

are removed from the call graph, and the n goals are connected such that goal i is connected to $i + 1$ (for $i < n$). This is shown in Figure 1b.

Algorithms for *selecting* sequential modules and basic blocks (i.e., their composite procedures and appropriate size) are an ongoing research topic of great importance, related to granularity analysis (e.g., [5, 15]). However, this is beyond the scope of our paper.

Another important notion to be defined in this context is that of *suspension*. The execution of a committed-choice goal will *suspend* whenever an input variable is not sufficiently instantiated for the goal to commit. This situation changes when another process further instantiates the variable, allowing the suspended process to resume. If the variable is never instantiated, *deadlock* will arise, which is defined as a situation where one or more suspended processes exist, but no runnable processes exist. We define suspension with respect to both modules and basic blocks:

Definition: A process (or a set of processes) is suspended *externally*, when the cause of the suspension lies outside the suspended module (basic block). □

For example, consider the query "`?- c(X),p(X).`" where `c/1` (the consumer of `X`) and `p/1` (the producer of `X`) are different modules. If `c/1` is executed first, then external suspension of `c/1` occurs. Intuitively, external suspension means that a process, external to the module, which produces data for the module, has run of out of data, and the module must wait for this producer to produce more data. When a process is suspended on more than one variable, with at least one of the causes of suspension outside the module, this is also called external suspension.

Definition: A process (or a set of processes) is suspended *internally*, when the cause of the suspension lies inside the suspended module (basic block).

□

Intuitively, internal suspension means that some variable which is local to the module or basic block is not sufficiently instantiated to allow the process to continue. If a process is suspended on more than one variable, suspension is called internal if and only if all causes are inside the suspended module. With these definitions, the following theorem is easy to derive:

Theorem 1 *If the goals inside a module or basic block are executed sequentially, then internal suspension will always cause deadlock for the top-level call of the module entry-point.* □

The proof is simple: if suspension occurs, the module has to wait for some other process to produce more data. However, this process is located inside the module, and will never be executed because the goals inside the module are executed sequentially. Therefore the call to this module deadlocks. Although this theorem may be obvious, its importance is that it clearly shows that internal suspension must be prevented for sequential basic blocks and modules. In the next section we begin to develop conditions under which this is true.

3 Directional Programs

One of the first attempts towards a parallel logic language which could be implemented efficiently on a distributed, loosely-coupled architecture was the Relational Language [3], the direct precursor of Parlog [2]. Apart from the mode system, which is also present in Parlog, the Relational Language featured *strong* arguments.

Definition: A *strong argument* of a procedure invocation, if it is in an output position, is a term that is completely constructed by a single body goal in that procedure's definition, without any contribution from other goals in this conjunction. The constructing goal is often called the *producer*. If a strong argument is in an input position, it is completely constructed by external goal(s), and no bindings to it are made in this procedure invocation.

□

Thus there can never be an output substitution for any variable occurring in a strong input argument position. This does not preclude the construction of terms containing unbound variables, but these variables can *never* be instantiated within the conjunction by a goal other than the producer. As an example, consider the following code segment:

```
mode f(^).
f1(X) :- X = g(0).
f2(X) :- X = g(Y).
f3(X) :- X = g(Y), Y = 0.
```

The X arguments in both f1/1 and f2/1 are strong (their values are completely constructed by a single body goal), whereas the X argument in f3/1 is not strong. Note that Y is not instantiated by another body goal in f2/1. The implication of strong variables is far reaching: "back communication" (e.g., the use of incomplete messages) is impossible. In general, the full power of the logical variable is lost.

For completeness, we now discuss the status of variables that are not strong. To retain the full power of the logical variable, *weak arguments* are necessary. For example, argument X in f3/1 is *weak* because, its value is constructed by two body goals within the same clause. For output arguments, the distinction between weak and strong is not important (in both cases, output bindings are only made through variables occurring in output positions), therefore in the following we will distinguish only between strong and weak *input* arguments.

> *Definition*: A *weak input argument* of a goal is one in which variables might be instantiated by the evaluation of the goal. □

Weak input arguments were introduced in Parlog (and implicitly used in other committed-choice languages), because Parlog programs only specify the modes for top-level arguments, without considering subterms of these top-level arguments. Consider, for example, the following correct Parlog program:

```
mode f(?).
f(g(X)) :- X=[].
```

Although the weak argument of f/1 is (correctly) defined as input, because it is a structured term g/1, the variable X inside this argument will be instantiated when f/1 is called.

With the definition of strong arguments, a *directional program* can be defined [8]:

> *Definition*: A *directional logic program* is a program in which all arguments (of all clauses of all procedures) are strong. □

A directional module can be defined similarly. In a directional program, the mode declarations indicate which body goal constructs the value of a variable: the goal in which the variable appears in an output argument position. Gregory [8] uses directionality only to check for the compile-time safety of guards, something which we do not discuss here. However, the directionality mechanism is more powerful — we show, in Theorem 2, that it can be used to significantly lower the number of suspensions:

> **Theorem 2** *If a program is directional, and given a correct ordering of the body goals, no deadlock will occur, and the program can only suspend externally, i.e., on variables whose producers are the top-level query.* □

Again, this theorem can easily be extended to cover modules. For example, consider the body goals "`f(A), g(A,B), h(B).`" Suppose `f/1` is the producer for `A` and `h/1` is the producer for `B`. A correct ordering, to avoid deadlock during sequential execution, is "`f(A) & h(B) & g(A,B).`"

It may be the case that more than one ordering exists satisfying this theorem; however, it is also possible that no such ordering exists, thus leading to internal suspension of a directional program. For example, consider the body goals "`f(A,B), g(A,B)`" where `f/2` is producer for `A` and `g/2` is producer for `B`. In this case, the program is still directional, but there is no ordering of `f/2` and `g/2` satisfying Theorem 2.

When body goals are executed in the "correct order," the number of suspensions is reduced because a producer of a variable is always executed before its consumer(s). As Section 3.2 illustrates, the requirement for a program to be directional (i.e., for all arguments to be strong), however, is too restrictive, and in the remainder of this paper we will describe an approach to ease this requirement, while retaining the full use of Theorem 2.

3.1 Program Directionality Check

A program is directional if all arguments are strong, i.e., for each variable there is no more than one goal invocation that is able to instantiate this variable. For Parlog, some programs are directional, whereas others, which use programming techniques such as incomplete messages, are not directional. To check for directionality, the entire program must be checked, including the query.

In a query, variables in output argument positions must be *distinct* variables, i.e., each variable may occur in no more than one output argument of a goal; this goal will then be the producer of this variable. To test whether a given program is directional, all clauses must be tested for the validity of the following rules due to Gregory [8]:

R1. A variable occurring in a head input position must not occur in output positions of goals in the body.

R2. Variables not occurring in head input positions must not occur in the output positions of more than one body goal.[5]

[5]Note that such a variable may occur in one or more output positions in the head!

This can be summarized as: "*In each clause and the query,*[6] *a variable may occur in any number of input positions in the head* or *in one output argument of a body goal, but not both* [8]." As is clear from these rules, the number of occurrences in input positions in body goals is of no relevance for program directionality.

To avoid internal suspension, we extend these rules with:

R3. For a variable occurring more than once in the body, the first "executed" occurrence[7] must be in an output position, thus later occurrences can only be in input positions.

Note that the variable in rule R3 cannot be in a head input position, by virtue of rule R1. Now, by extending Theorem 2 to cover modules, and defining "correct ordering" to be that ordering of body goals which adheres to rule R3 for directionality, the following theorem follows:

> **Theorem 3** *If, at compile-time, a sequential module can be proved to be directional, then only external suspension can occur with respect to that module.* □

In the rest of the program, other forms of suspension can occur, but the important point is that there can never be internal suspension in that module. If directionality can be forced or proved, the module can be compiled efficiently, as will be discussed in Section 6. Otherwise, compilation of the sequential module defaults to standard committed-choice compilation techniques (e.g., [4, 9, 13, 14]).

3.2 Example of Directionality Check

In this section we gives two examples of the directionality check to show why the requirement of strong arguments is too restrictive. Consider the sequential module `qsort`, containing the quicksort function, in Figure 2a. In this program we use a split difference list: the second and third arguments of `q/3` are the two parts of the difference list.[8] A typical query to this program is:

```
?- q([4,2,1,3],Sorted,[]).
     Sorted = [4,3,2,1]
```

[6]The query is treated as a clause body.

[7]Assuming a left-to-right execution order, this is also the textually first occurring body goal.

[8]A difference list uses a logical variable to represent the end of a list. In this way, lists can be composed and split without having to traverse and copy them. Usually difference lists (or D-lists for short) are written as `X-Y`, but sometimes they are used implicitly, with `X` and `Y` written as two separate arguments.

```
:- sequential module qsort.
:- export q/3.

mode q(?,^,?).
q([],R,R).
q([X|L],R0,R) :-
    s(L,X,L1,L2),
    q(L1,R0,[X|R1]),
    q(L2,R1,R).

mode s(?,?,^,^).
s([],_,[],[]).
s([X|Xs],A,S,[X|L1]) :- A>X | s(Xs,A,S,L1).
s([X|Xs],A,[X|S1],L) :- A=<X | s(Xs,A,S1,L).
```

(a)

```
:- export q/2.

mode q(?,?).
q([],R0-R) :- R0=R.
q([X|L],R0-R) :-
    s(L,X,L1,L2),
    q(L1,R0-[X|R1]),
    q(L2,R1-R).
```

(b)

Figure 2: Quicksort Program in Parlog (a) with split D-list (b) with explicit D-list.

With the modes[9] given in Figure 2a, and applying rules R1 and R2, it is easy to determine that the module is directional. As Gregory [8] points out, we can determine the directionality because the input arguments are strong. Consider, however, the version of the module with a difference list of the form `X-Y`, as shown in Figure 2b with the same definition of `s/4`. Here the query appears as:

```
?- q([4,2,1,3],Sorted-[]).
   Sorted = [4,3,2,1]
```

The first clause of `q/2` violates R1, so the directionality algorithm will qualify this program as non-directional, although in fact both programs function in the exact same manner. To overcome problems like this, making it possible to use the full power of sequential modules, another form of mode analysis is necessary.

4 Mode Analysis

It is clear that the requirement for full directionality with only strong arguments is too restrictive to be effective. Therefore another type of mode analysis is necessary. A viable alternative is the mode analysis described by Ueda [18] (other options, which we do not consider in this paper, are available, such as abstract interpretation [10]). Ueda's mode analysis automatically infers the modes of all arguments at the top-level

[9]The builtin =/2 operator performs general unification (as defined in Parlog and FGHC, for instance). Thus =/2 has no fixed modes, i.e., any combination of input and output modes is acceptable. In other words, one can assign any modes that satisfy rules R1, R2, and R3.

and those inside structured top-level arguments. A mode is either *input* or *output*[10] as defined in Section 2.

4.1 Simplified Rules

The mode analysis described in this section reviews Ueda's technique, although a different (hopefully more clear) exposition and less formal notation is used here. The mode analysis uses the notion of a *path* to denote a specific (textual) occurrence of a term. A path describes how the different layers of structured data elements must be "peeled off" to get to this specific occurrence. For example, given the clause head `f(X,g([X|_]))`, the path to the first occurrence of `X` is `<f,1>`, while the path to the second occurrence is `<f,2><g,1><.,1>`. This second path means "take the second argument of `f`," then "take the first argument of `g`," and then "take the first argument of the list."

The mode of a path p, which is denoted as $m(p)$, is defined as either input or output. The definitions of input and output in [18] are somewhat ambiguous, so we use two improved definitions [17]. Let p be a path leading to a variable. Then:

- "$m(p)$ = in" means that the value (i.e., the principal functor) of the variable at p *may* (but need not) be bound by the caller, and *will never* be bound by the callee.
- "$m(p)$ = out" means that the value (i.e., the principal functor) of the variable at p *may* (but need not) be bound by the callee, and *will never* be bound by the caller.

Intuitively, this implies that an input path will not be (further) instantiated, while an output path implies that a process can never suspend on the value of the path. Note that variable-variable unification does not count as "binding the value" of a variable. Thus, input paths leading to variables cannot cause suspension, even if the incoming argument is unbound.

If possible, the modes of all paths must be inferred to find a safe goal ordering, which avoids internal suspension. The mode of a path can be found by applying the following rules [17, 18]:

§1. For some path p in a clause, $m(p)$ = in, if either

(a) p leads to a non-variable in the head or body, or

(b) p leads to a variable which occurs more than once in the head, or

[10]Shortened to "in" and "out" in certain contexts.

(c) p leads to a variable which also occurs in the guard at path p_h and $m(p_h) = \text{in}$

§2. Two arguments of a unification body goal (`=/2`) have opposite modes, for all possible p, or more formally: $\{\forall p\ \mathtt{m}(<=,1>\mathtt{p}) \neq \mathtt{m}(<=,2>\mathtt{p})\}$.

To better understand the modes, we now give the intuition for these rules. If a path leads to a non-variable in the clause, then the value of the path is already known, and it will not be instantiated by the callee, thus its mode is 'in.' If a variable occurs more than once in the head, it can only be used for equality checking before commitment.[11] No bindings are allowed, thus again its mode is 'in'. Similarly, if a variable in the guard is used for checking (i.e., its mode is 'in'), then it is clear that value of the path to that variable is provided by the caller, so its mode is 'in.'

Because of the nature of unification, one of the arguments of `=/2` will function as producer for a specific path, while the other argument functions as consumer. Note, however, that §2 does *not* require that for all possible paths p, the modes of an argument are the same. For example, given the unification `[1,X]=[Y,2]`, `m(<=,1><.,1>) = in`, while `m(<=,1><.,2>) = out`.

When a variable occurs more than once, with at least one occurrence in the body, the situation gets more complicated. Because paths to variables occurring more than once in the head all have the same (input) mode (see §1b), it is correct (and simpler) to count only one occurrence of a variable in the head in the following rules.

§3. If there are exactly two occurrences, we have two possibilities:

(a) If both occurrences are in the body, the modes of their paths are inverted.

(b) If there is one occurrence in the head and one in the body, the modes of their paths are the same.

§4. If there are more than two occurrences of a shared variable (i.e., at least two occurrences in the body), the situation is even more complex:

(a) If the body contains more than two occurrences of the shared variable and the head has no occurrences, then one of the modes is 'out,' and the others are 'in.'[12]

[11]Note that these semantics are particular to FGHC. For example, the procedure "`f(X,X) :- X=3.`" can only succeed with the query "`?- f(3,3).`" If the intention was to output two copies of 3, then the proper code is: "`f(X,Y) :- X=3, Y=X.`" Parlog avoids this problem by allowing both arguments to be explicitly defined as output modes.

[12]This means that one of the occurrences is designated as the producer of this variable.

(b) If the head contains one occurrence of the shared variable (so the body has two or more occurrences), then the modes are as follows:

   i. The mode of the head occurrence is 'in' iff the modes of all body occurrences are 'in.'[13]

   ii. The mode of the head occurrence is 'out' iff *one* of the body occurrences is 'out,' and the other body occurrences are 'in.'

The inversion in §3 can be intuitively explained by looking at an input path in a clause head. For that clause, the variable acts as a consumer of data (therefore its mode is input). However, within the clause the variable in the head acts as a producer for the body of the clause, thus inverting its mode within the clause. The opposite of this holds for an output path in the head.

The complexity of §4 can be explained intuitively by looking at the inversion of the modes discussed previously, combined with the fact that only one occurrence of a variable can be its actual producer. The problem with this last rule is that it causes non-binary constraints to occur. To clarify this, we will give two examples of how the modes of a shared variable can look when multiple (three in this case) occurrences are present.

f(X)	:−	b1(X),	b2(X).
in		in	in
out		out	in
out		in	out

f	:−	b1(X),	b2(X),	b3(X)
		out	in	in
		in	out	in
		in	in	out

4.2 A Small Example of Mode Analysis

To explain this concept of modes and paths, and to understand the rules given above, it is best to start with a small example [18]. Figure 3 shows the proofs for `test/2`, a simple predicate with two clauses. In these proofs, let t_i`(p)` $\equiv$ `m(<test,i>p)`, while '$=_k$' represents the unification goal in clause k (in general, it denotes the k^{th} instance of `=/2`) and 'ϵ' represents an empty path. Furthermore, each step in the proof is annotated with the rule used. Each mode relationship proved is called an *axiom*, e.g., there are eight axioms comprising the full mode definition for `test/2`.

As an example of how such a proofs are constructed, consider the third axiom in Figure 3. We start with the path to the head of the list on the right-hand side of $=_2$`/2`. The value of this path is a non-variable (`push/1`), so according to §1a, its mode is 'in.' Using §2, we then derive that the mode of the corresponding path on the left-hand side

[13]Note that if a variable occurs more than once in the head, its mode is 'in' by §1b, implying that §4b-ii cannot be used.

test(M,S) :- M=:=0 | S $=_1$ [].
test(M,S) :- M=\=0 | S $=_2$ [push(M),pop(N)|S1], N1:=N-1, test(N1,S1).

1.		$t1(\epsilon)$ = in	§1c
2.		$t2(\epsilon)$ = out	
	a.	$m(<=_1,2>)$ = in	§1a
	b.	$m(<=_1,1>)$ = out	§2
	c.	$\forall p\ m(<=_1,1>p) = t2(p)$	§3b (on S)
	d.	$t2(\epsilon)$ = out	b+c: sub $p=\epsilon$
3.		$t2(<.,1>)$ = out	
	a.	$m(<=_2,2><.,1>)$ = in	§1a
	b.	$m(<=_2,1><.,1>)$ = out	§2
	c.	$\forall p\ m(<=_2,1>p) = t2(p)$	§3b (on S)
	d.	$t2(<.,1>)$ = out	b+c: sub $p=<.,1>$
4.		$t2(<.,2>)$ = out	see "3"
5.		$t2(<.,2><.,1>)$ = out	see "3"
6.		$\forall p\ (t2(<.,2><.,2>p) = t2(p))$	
	a.	$\forall p\ t2(p) = m(<=_2,1>p)$	§3b (on S)
	b.	$\forall p\ m(<=_2,2><.,2><.,2>p) \neq t2(p)$	§3a (on S1)
	c.	$\forall p\ m(<=_2,1><.,2><.,2>p) = t2(p)$	§2
	d.	$\forall p\ t2(<.,2><.,2>p) = m(<=_2,1><.,2><.,2>p)$	a: sub $p=<.,2><.,2>$
	e.	$\forall p\ t2(<.,2><.,2>p) = t2(p)$	c+d
7.		$t2(<.,1><push,1>)$ = out	
	a.	$t1(\epsilon)$ = in	see "1"
	b.	$\forall p\ t1(p) = m(<=_2,2><.,1><push,1>p)$	§3b (on M)
	c.	$m(<=_2,2><.,1><push,1>)$ = in	a+b: sub $p=\epsilon$
	d.	$m(<=_2,1><.,1><push,1>)$ = out	§2
	e.	$\forall p\ t2(p) = m(<=_2,1>p)$	§3b (on S)
	f.	$t2(<.,1><push,1>)$ = out	d+e
8.		$t2(<.,2><.,1><pop,1>)$ = in	
	a.	$m(<:=,2><-,1>)$ = in	intuition[a] (on N)
	b.	$\forall p\ m(<=_2,2><.,2><.,1><pop,1>p) \neq m(<:=,2><-,1>p)$	§3a
	c.	$m(<=_2,2><.,2><.,1><pop,1>)$ = out	a+b: sub $p=\epsilon$
	d.	$m(<=_2,1><.,2><.,1><pop,1>)$ = in	§2
	e.	$\forall p\ t2(p) = m(<=_2,1>p)$	§3b (on S)
	f.	$t2(<.,2><.,1><pop,1>)$ = in	d+e

[a]By definition, the mode $m(<:=,2>p)$ is 'in.'

Figure 3: Mode Analysis Proof for test/2.

of $=_2/2$ is 'out.' As there are two occurrences of the variable S, one in the head and one in the body, §3b applies. If we now substitute p=<.,1> in the last axiom derived, and combine it with the previous one, the final outcome is t2<.,1>) = out. The other axioms are proved similarly. Together these results give an idea how data flows within this procedure, i.e., which arguments (and which variables within structures) are input, and which are output. This information can be used by a compiler to make optimal use of its resources, such as memory and registers.

To show that mode analysis can fail as well, consider the following example:

```
f :- r(X,Y), g(X), g(Y).
r(X,Y) :- X=Y.
```

Applying §3a to the first clause gives: $\forall p\ \mathtt{m}(<\mathtt{r},1>\mathtt{p}) \neq \mathtt{m}(<\mathtt{g},1>\mathtt{p})$ and $\forall p\ \mathtt{m}(<\mathtt{r},2>\mathtt{p}) \neq \mathtt{m}(<\mathtt{g},1>\mathtt{p})$. In other words: $\forall p\ \mathtt{m}(<\mathtt{r},1>\mathtt{p}) = \mathtt{m}(<\mathtt{r},2>\mathtt{p})$ However, applying §2 to the second clause gives: $\forall p\ \mathtt{m}(<\mathtt{r},1>\mathtt{p}) \neq \mathtt{m}(<\mathtt{r},2>\mathtt{p})$, which clearly contradicts the previous conclusion, so the mode analysis fails and no consistent modes can be found. This implies that this code cannot safely be considered as a sequential module, and standard compilation techniques must be used. The reason why the mode analysis fails can be seen more clearly from the unfolded program:

```
f :- g(X), g(X).
```

If both occurrences of X had mode 'in,' then there would be no producer for X. On the other hand, if both occurrences of X are 'out,' then they are competing to produce a value for X, but it is not certain which of the processes will generate the value. Therefore we cannot say which occurrence must be executed first to obtain a safe, i.e., deadlock-free, sequential module.

4.3 A Larger Example of Mode Analysis

Figures 4 and 5 give the mode analysis for quicksort, shown in Figure 2. In these proofs, let $\mathtt{q}_i\mathtt{(p)} \equiv \mathtt{m(<q,i>p)}$ and $\mathtt{s}_i\mathtt{(p)} \equiv \mathtt{m(<s,i>p)}$. The most interesting point about this analysis is the sixth axiom in Figure 4. The analysis shows that three distinct modes of execution are possible for q/3, hinging on the use of shared variable X. Since X appears twice in the body of the second clause, the proof uses rule §4b. The three distinct modes depend on whether X is input by the clause head (first choice) or bound by a body goal (latter two choices). Given the mode analysis of s/4 in Figure 5, the second choice (b) is contradicted, since $\mathtt{s2}(\epsilon)$ = in. Therefore only two choices remain.

The problem is that these two choices are both valid: essentially Ueda's analysis indicates that without further information (e.g., modes of the query), X may be generated by the third argument of q/3 and output through the first argument (c), or vice versa

```
q([], R0,R) :- R0 =₁ R.
q([X|L],R0,R) :- s(L,X,L1,L2),q(L1,R0,[X|R1]),q(L2,R1,R).
```

```
1.  q1(ε) = in                                        §1a

2.  q3(ε) = in                                        §1a, 2nd body goal

3.  q2(ε) = out
    a.  ∀p q2(p) = m(<=₁,1>p)                         §3b
    b.  ∀p q3(p) = m(<=₁,2>p)                         §3b
    c.  ∀p q2(p) ≠ q3(p)                              §2
    d.  q2(ε) = out                                   "2"+c:  sub p=ε

4.  ∀p q1(<.,2>p) = s1(p)                             §3b (on L)

5.  ∀p q3(p) = q3(<.,2>p)
    a.  ∀p q3(<.,2>p) ≠ q2(p)                         §3a (on R1)
    b.  ∀p q3(p) = q3(<.,2>p)                         "3.c" + a

6.  Three possibilities:                              §4b
    a.  q1(<.,1>p) = in, s2(p) = in, q3(<.,1>p) = in
    b.  q1(<.,1>p) = out, s2(p) = out, q3(<.,1>p) = in
    c.  q1(<.,1>p) = out, s2(p) = in, q3(<.,1>p) = out

7.  s3(ε) = out
    a.  ∀p q1(p) ≠ s3(p)                              §3a (on L1)
    b.  s3(ε) = out                                   "1" + a

8.  s4(ε) = out                                       see "7"
```

Figure 4: Mode Analysis Proof for Quicksort: q/3.

(a). To the programmer, whose intent may have been the standard use of quicksort (a), this result may seem odd. However, choice (c) is valid — consider the query:

```
?- q([X],[3,2,1],[2,1]).
   X = 3
```

Although this is an artificially-created query, which is unlikely to appear in most programs, this query is valid, and should execute successfully!

This example shows that to safely execute this module sequentially, more information is needed about the mode of the first argument. One way of finding this mode is by looking at the the place(s) where the module is called. For example, given the definition and the modes of gen/2 in Figure 6, with g_i(p) ≡ m(<gen,i>p), look at the following call to the module:

```
...,gen(Max,List),q(List,SortedList,[]),...
```

Because only one occurrence of a variable in the body (or query) can have mode out

s([], _,S,L) :- S $=_1$ [], L $=_2$ [].
s([X|Xs],A,S,L) :- A>X | L $=_3$ [X|L1], s(Xs,A,S,L1).
s([X|Xs],A,S,L) :- A=<X | S $=_4$ [X|S1], s(Xs,A,S1,L).

1.		s1(ϵ) = in	§1a
2.		s1(<.,1>) = in	§1c
3.		s2(ϵ) = in	§1c
4.		s3(ϵ) = out	
	a.	m(<$=_1$,2>) = in	§1a
	b.	m(<$=_1$,1>) = out	§2
	c.	$\forall p$ m($=_1$,1>p) = s3(p)	§3b
	d.	s3(ϵ) = out	c: sub p=ϵ
5.		s4(ϵ) = out	see "4"
6.		$\forall p$ s1(p) = s1(<.,2>p)	§3b
7.		$\forall p$ s4(p) = s4(<.,2>p)	
	a.	$\forall p$ s4(p) $\neq$ m(<$=_3$,2><.,2>p)	§3a (on L1)
	b.	$\forall p$ s4(p) = m(<$=_3$,1>p)	§3b (on L)
	c.	$\forall p$ s4(<.,2>p) = m(<$=_3$,1><.,2>p)	b: sub p=<.,2>
	d.	m(<$=_3$,1>p) $\neq$ m(<$=_3$,2>p)	§2
	e.	$\forall p$ s4(p) = s4(<.,2>p)	a+c+d
8.		$\forall p$ s3(p) = s3(<.,2>p)	see "7"
9.		s4(<.,1>) = out	
	a.	s1(<.,1>) = in	see "2"
	b.	$\forall p$ s1(<.,1>p) = m(<$=_3$,2><.,1>p)	§3b
	c.	m(<$=_3$,2><.,1>) = in	b: sub p=ϵ
	d.	m(<$=_3$,1><.,1>) = out	§2
	e.	s4(<.,1>) = out	see "7.b"
10.		s3(<.,1>) = out	see "9"

Figure 5: Mode Analysis Proof for Quicksort: s/4.

```
gen(0,X) :- G =1 □.
gen(I,X) :- I1:=I-1, X =2 [I|Xs], gen(I1,Xs).
```

1.	g1(ϵ) = in	§1a
2.	g2(ϵ) = out	§1a, §2, §3b
3.	g2(<.,1>) = out	"1", §4b-i, §2
4.	$\forall p$ g2(p) = g2(<.,2>p)	§3a, §3b, §2

Figure 6: Mode Analysis for gen/2.

and Figure 6 shows that g_2(<.,1>) = out, we can conclude that q_1(<.,1>) = in, which leaves us with only the normal use of quicksort (a).

This example shows that to safely execute a module sequentially, sometimes more information is needed than can be derived from the module definition itself. One way of finding this information is by using the context information, i.e., looking where the module is called. A second method is having the user explicitly specify modes to disambiguate such cases. For example, to indicate that an argument will be input throughout, i.e., for all subterms, the user can specify a *strong input argument*, perhaps with the mnemonic '??'. Note that this is not the same as fully ground!

One can even envision having the compiler query the programmer to provide this disambiguating information. The advantage of compiler queries is that only a small percentage of modes will likely need to be disambiguated. Thus the programmer would be saved from having to map out all the modes *a priori*, and need only specify ones that the analysis proves are ambiguous. We might add that this tends to be a good programming practice anyway! Good logic programmers specify the top-level argument modes of each procedure (possibly as a comment), but as shown, this is not sufficient documentation in all cases. For the example discussed above, the determination of safe sequential execution hinges on whether or not the head of the first argument is bound, so help is needed for this one mode only.

5 Putting it All Together: Goal Ordering

In this section we discuss how the mode information is used to reorder body goals within a clause to ensure safe sequential execution. Recall that in Section 3.1 we gave a directionality rule, R3, which ensures that internal suspension is impossible in a sequential module. The rule states that for a variable occurring more than once

q1(ϵ)	in	q1(ϵ)	in
q1(<.,1>p)	in	q1(<.,1>p)	out
q1(<.,2>)	in	q1(<.,2>)	in
q2(ϵ)	out	q2(ϵ)	out
q3(ϵ)	in	q3(ϵ)	in
q3(<.,1>p)	in	q3(<.,1>p)	out
q3(<.,2>)	in	q3(<.,2>)	in
s1(ϵ)	in	s1(ϵ)	in
s1(<.,1>)	in	s1(<.,1>)	in
s1(<.,2>)	in	s1(<.,2>)	in
s2(ϵ)	in	s2(ϵ)	in
s3(ϵ)	out	s3(ϵ)	out
s3(<.,1>)	in	s3(<.,1>)	out
s3(<.,2>)	out	s3(<.,2>)	out
s4(ϵ)	out	s4(ϵ)	out
s4(<.,1>)	out	s4(<.,1>)	out
s4(<.,2>)	out	s4(<.,2>)	out

Table 1: Two Possible Sets of Ground Modes for `q/3`.

in the body, the first executed occurrence must be in an output position, thus later occurrences can be in input positions only. The basic idea is to find an ordering that does not contradict this rule, using the axioms derived by the mode analysis. We now present an algorithm for doing this.

The first point to consider is that not all the mode axioms that fully describe a procedure are needed here, nor do we need the axioms in their full generality. Certain axioms, which we call *recursive axioms*, have a general form similar to axiom 5 of `q/3`: $\forall p$ `q3(p) = q3(<.,2>p)`. The meaning is that the mode of any third argument path is the same as the mode of the tail of that path. Intuitively, the procedure is recursing on the third argument, which is a list.

For the purposes of goal ordering, only the modes of variables appearing (syntactically) in the procedure definition are needed. Therefore all general paths p in the derived axioms can be instantiated to ϵ. For example, in the previous axiom, `q3(`ϵ`) = q3(<.,2>)`. Combining this with axiom 2 in Figure 4 allows us to derive that `q3(<.,2>) = in`. We call this process of instantiating general mode axioms into less general axioms, *recursive grounding*. As another example, combining axiom 4 in Figure 4 with axiom 1 in Figure 5 gives `q1(<.,2>) = in`.

Table 1 shows the two possible sets of modes for procedure `q/3`, after recursive grounding. The next step is to attempt to order the body goals in the second clause[14] so that the constraints implied by both sets of axioms (with respect to rule R3) are satisfied. The two sets of modes can be illustrated as follows:

```
q([X|L],R0,R) :- s(L,X,L1,L2), q(L1,R0,[X|R1]), q(L2,R1,R).
q([?|?], ^,?) :- s(?,?, ^, ^), q( ?, ^,[?| ?]), q( ?, ^,?).
q([^|?], ^,?) :- s(?,?, ^, ^), q( ?, ^,[^| ?]), q( ?, ^,?).
```

For each set of modes, the ordering algorithm iterates through each variable occurring more than once in the body: X, L1, L2, and R1. For each, constraint(s) are created relating the goals containing that variable. For example, for the first set of modes, X induces no constraints (since both the first and second goals use X for input). Examination of L1, however, induces the constraint: $G1 < G2$. Each new constraint is checked for consistency with previously generated constraints. A contradiction is fatal: the clause cannot be ordered for sequential execution.

Continuing with the previous example, L2 induces $G1 < G3$ and R1 induces $G3 < G2$. So far, all these constraints are consistent. However, analyzing the second set of modes, we find that X induces $G2 < G1$. This alone causes the analysis to fail (and subsequent examination of L1 causes additional contradictions within the second set of modes itself).

If we can derive that the mode of the head of the first argument is 'in' (for example using the context information, as shown in Section 4.3), then there is only one set of modes, and the ordering algorithm terminates successfully with the following order:

```
q([X|L],R0,R) :- s(L,X,L1,L2), q(L2,R1,R), q(L1,R0,[X|R1]).
q([?|?], ^,?) :- s(?,?, ^, ^), q( ?, ^,?), q( ?, ^,[?| ?]).
```

6 Conclusions and Future Research

This paper presents a compile-time technique for determining if a set of procedures within a parallel program can be executed sequentially without causing deadlock. More specifically, the analysis and methods are described in the context of committed-choice parallel logic programming languages, such as FCP, FGHC, and Parlog. These concurrent languages have inherent fine-grain parallelism, so that the task at hand is to throttle high-overhead parallelism, rather than uncovering more parallelism. We present a framework of sequential program modules and basic blocks that can be derived and guaranteed to be deadlock-free at compile time. This paper outlines a source-to-source code optimization that ensures that sequential execution can proceed smoothly. Thus "traditional" procedural language optimizations that have previously been discarded by those implementing committed-choice languages (e.g., [4, 9, 13, 14]) can now be

[14]The first clause has only one goal, so it does not require ordering.

considered, such as continuation-based goal management and interprocedural register allocation.

Future research in this area includes two major targets. First, algorithms must be designed for the selection of sequential modules and basic blocks. Selection walks a fine line between making modules too large, thereby throttling too much parallelism, and making modules too small, thereby not increasing granularity enough to be effective. The second target of future research is to gain experience with the techniques by automating the mode-analysis algorithm, and characterizing some benchmark programs. Automation of Ueda's method is an open research area, as is the characterization of what percentage of code within real programs can be successfully sequentialized.

Acknowledgements

M. Korsloot was supported by a grant from the Delft University of Technology. E. Tick was supported by an NSF Presidential Young Investigator Award. We are indebted to Kazunori Ueda for helping us understand the mode analysis.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1985.

[2] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 84–139. MIT Press, Cambridge MA, 1987.

[3] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 171–178. ACM, Portsmouth NH, October 1981.

[4] J. A. Crammond. *Implementation of Committed-Choice Logic Languages on Shared-Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Endinburgh, May 1988.

[5] S. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 174–188, June 1990.

[6] I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conference on Logic Programming*, pages 497–512. Cleveland,

MIT Press, October 1989.

[7] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[8] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.

[9] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, August 1987.

[10] C. S. Mellish. Abstract Interpretation of PROLOG Programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Ltd, Chichester, 1987.

[11] R. Penrose. *The Emperor's New Mind.* Vintage, 1989.

[12] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[13] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1,2. MIT Press, Cambridge MA, 1987.

[14] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[15] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. *New Generation Computing*, 7(2):325–337, January 1990.

[16] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

[17] K. Ueda. personal communication, November 1990.

[18] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.