THE DESIGN AND IMPLEMENTATION OF A PARALLEL PROLOG OPCODE-INTERPRETER ON A MULTIPROCESSOR ARCHITECTURE

Carolyn Ann Hakansson
B.A., Reed College, 1984

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

August 3, 1987

The thesis "The Design and Implemenation of a Parallel Prolog Opcode-Interpreter on a Multiprocess Architecture" by Carolyn Ann Hakansson has been examined and approved by the following Examination Committee:

Peter Borgwardt, Thesis Research Advisor
Tektronix Inc.

David Maier
Associate Professor,
Department of Computer Science and Engineering
Oregon Graduate Center

Dan Hammerstrom
Associate Professor,
Department of Computer Science and Engineering
Oregon Graduate Center

Richard Hamlet
Professor,
Department of Computer Science and Engineering
Oregon Graduate Center

ACKNOWLEDGEMENTS

There are several people whose support and guidance has helped to make this thesis one of the most positive experiences in my graduate studies.

I would like to express my gratitude and appreciation to my advisor Peter Borgwardt, for providing me with this unique opportunity and experience in parallel logic programming. His time and energy were greatly appreciated.

I would also like to thank my thesis committee: David Maier, Dan Hammerstrom, and Dick Hamlet for their advice and comments that helped to shape this thesis; Sequent Computer Systems and Joe DiMartino for the use of the mkt3 Balance 21000 machine and the technical support and advice when it was needed; Bart Schaefer for his technical experience and knowledge of the Balance Series; and Casey Bahr for his encouragement and hours of proof-reading.

Finally, I would like to give a warm thank you to my family and friends whose understanding and reassurance helped make my graduate studies a valuable experience.

For my grandparents Nils and Anna Hakansson,

Martin and Selma Kates,

my parents Nils and Joyce Hakansson,

my brother Alexander Hakansson,

and my husband Casey.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

THE DESIGN AND IMPLEMENTATION
OF A
PARALLEL PROLOG OPCODE-INTERPRETER
ON A
MULTIPROCESSOR ARCHITECTURE

Carolyn Ann Hakansson
Oregon Graduate Center, 1987

Supervising Professor: Peter Borgwardt

Various algorithms for reducing overhead in parallel Prolog systems are studied, and an attempt to increase the speed of sequential Prolog execution is made by implementing a Parallel Prolog Opcode-Interpreter (PAPI). This opcode-interpreter exploits AND-parallelism and runs on a shared-memory multiprocessor architecture. Efforts are made to reduce the potential for high overhead in the areas of communication, backtracking, and variable-binding conflict detection.

The design and implementation of PAPI is discussed and followed by benchmark tests and analysis. The results from benchmark testing indicate that Prolog programs that backtrack do not show an improvement in performance over sequential execution, but deterministic Prolog programs executed in parallel illustrate a marked improvement over sequential execution. Prolog programs that contain large amounts of parallelism and create deep proof trees benefit most from this implementation of AND-parallelism. Possible explanations for these findings and suggestions for future research are also presented.

CHAPTER 1

Introduction

With the recent growth and decreased cost of multiprocessor computers comes the motivation and means for developing faster and more efficient programs. A single-processor Von Neumann architecture will not satisfy the demands for a low-cost, high-performance environment. The main drawback of this architecture is that a point of diminishing returns is reached where each additional increase in performance requires an excessive increase in cost. Recent research in areas such as VLSI and computer architecture has developed cheaper methods of producing chips and faster architectures incorporating parallelism. This improved technology has created supercomputers that exploit multiprocessor architecture and parallelism to make use of increased computational power effectively.

As hardware progresses, the opportunity for exploiting parallelism, hence increasing speed, in software is presented. Logic programs are a likely candidate for multiprocessor computers since (1) their sequential execution is inherently slower than that of imperative language programs and (2) they offer more opportunities for parallelism. There are many types of parallelism in logic programming to choose from, each with its own problems and attributes. One such problem is the potential for high overhead costs of time and memory. This overhead may, however, be reduced by choosing efficient algorithms for areas where overhead may present a problem.

The implementation of an efficient parallel Prolog opcode-interpreter on a shared-memory multiprocessor architecture is the focus of this thesis. The Prolog opcode-interpreter, unlike a Prolog interpreter, requires that a Prolog program be compiled into intermediate-code instructions and then assembled into opcode instructions before the program is executable. This method is used to gain greater efficiency over a pure Prolog interpreter.

The source of parallelism chosen for this project and implemented in the opcode-interpreter is AND-parallelism. Algorithms including the RAP scheme by DeGroot [DeG84] [DeG85] and semi-intelligent backtracking by Chang and Despain [ChD85] were incorporated in the parallel Prolog opcode-interpreter to minimize overhead.

1.1. Parallelism in logic programs

Parallelism in logic programming languages is often described as *natural*, due to the non-deterministic nature and declarative semantics of logic programs. Prolog is a logic programming language consisting of *clauses* (or rules) and *facts* [1]. Clauses are made of a *head* and a *body*. The head is a *literal* that matches goals and the body contains one or more goals. A goal is a *literal* with one or more *arguments*, in parentheses. These arguments are either *variables* (first character is uppercase), *values* (first character is lowercase), or *terms* (comprised of structures or functions). A fact is a clause without a body, that is, a single goal, and always has values for

[1] For a thorough discussion of Prolog, see [CIM84].

arguments. For example, the following are examples of clauses:

```
grandfather(X, alex):- male(X),
                       parent(X, Z),
                       parent(Z, alex).
```

In the first clause, `grandfather(X, alex)` is the head, `grandfather` is the predicate of the head, and `X` and `alex` are the head's arguments. The body, which is everything to the right of the `:-` symbol, consists of the goals `male(X)`, `parent(X, Z)` and `parent(Z, alex)`. Below are examples of facts.

```
male(nils).
male(hemming).
male(alex).
parent(nils, hemming).
parent(hemming, alex).
```

Prolog operates by matching (or unifying) a goal with a fact or a clause head. Matching (or unification) occurs if the predicates of the goal and clause head or fact are the same, and if the corresponding arguments match. If an argument is a variable, it is instantiated to the corresponding value or variable. In a Prolog program, often several facts and clause heads will match a goal. That is, there may be several possible paths through the program that lead to a solution. Prolog programs in which more than one fact or clause head matches a goal are *non-deterministic*, whereas Prolog programs in which only one fact or clause head matches a goal are *deterministic*.

The declarative semantics characteristic of Prolog arises from the separation of logic and control. That is, the programmer does not control or direct the

execution of the Prolog program (as one does, in say, "C"). Rather, the Prolog program contains the logic, true information, and rules for determining if a goal is `true` or not `true`, while the interpreter chooses the control, how and in what order to evaluate the execution paths. It is the separation of logic and control and the programmer's lack of control specifications that permit the evaluation of several execution paths in parallel, and hence, makes parallelism feasible in declarative languages.

Based on these natural characteristics in logic programming languages for parallelism, several types of parallelism have been defined and explored. Conery and Kibler [CoK85] [CoK81] discuss four types of parallelism for logic programs, two of which, AND-parallelism and OR-parallelism, are high level and exploit the non-deterministic character of logic programming languages (although AND-parallelism does not *depend* on non-determinism). The other two, STREAM-parallelism and SEARCH-parallelism, are of finer granularity and take advantage of the declarative semantics aspect of logic programming languages. A fifth type of parallelism, UNIFICATION-parallelism, focuses on the unification process in logic programs. A short description of each is given below.

AND-parallelism:

> AND-parallelism is the execution of several goals in a single clause body in parallel. The objective is to increase the speed of finding a single solution by examining the subparts of a clause simultaneously. If a possible solution fails at any point of execution, backtracking must be done to find another execution path.

OR-parallelism:

OR-parallelism occurs when a goal unifies with the head of more than one clause and each of the clauses is then executed in parallel. The objective is to increase the speed of execution by exploring multiple paths for solutions simultaneously. This search for all solutions simultaneously removes the need for backtracking.

SEARCH-parallelism:

SEARCH-parallelism applies to logic programs with a very large database of clauses. The database is broken into disjoint sets, and each set is searched, in parallel, for clauses whose head unifies with a given goal. This method is recommended for initializing OR-parallelism [CoK81].

STREAM-parallelism:

STREAM-parallelism is the examination of complex data structures by several processes in parallel with the process producing the structure. That is, one process may create a list structure while the other processes examine the finished members of the list structure as it is being constructed. This type of parallelism permits the testing for membership in a structure by several processes as the structure is created by another process and is often used in conjunction with AND-parallelism.

UNIFICATION-parallelism:

UNIFICATION-parallelism occurs while unifying a goal with the head of a clause. It may be possible to unify several pairs of corresponding arguments or terms in parallel. For example, if the goal is

```
stats(68, A, joyce)
```

and the clause head is

```
stats(HeightInches, WeightPounds, joyce)
```

then the unifications of `68` with `HeightInches`, `A` with `WeightPounds`, and `joyce` with `joyce` are independent and may be done in parallel.

The latter three types of parallelism described above are lower level and therefore pertain to specific functions of a logic program. For example, SEARCH-parallelism may be employed in logic programs that frequently search a large database of clauses. The program will have several processes searching the database in parallel, decreasing search time. But when the program is not searching, parallelism is not possible, and the program executes sequentially. Therefore, this type of parallelism is specific to logic programs that search databases extensively and only occurs during the search. The *specific application* characteristic and low granularity of the latter three types of parallelism is undesirable for this project, but the center of attention for others.

SEARCH-parallelism has been the topic of several researchers [EKM82] [WAD84] [TLJ84]. As searching is the most expensive part of sequential Prolog execution, it is not surprising that research has been in the direction of searching large databases of clauses in parallel. This is the case in D.H.D. Warren's research [WAD84] which presents an algorithm for which Prolog is used as a database query language. The major disadvantage with this form of parallelism is that the algorithms tend to be sophisticated and difficult to implement [TLJ84].

STREAM-parallelism has been researched in a pure form, as a part of a complex system [ClM79] [EmL81] [Kow74] [Sha83] [LiP84] [KTM86] [Bor84], and as the building block for several proposed parallel programming languages including: Relational Language, Concurrent Prolog, Parlog, Guarded Horn Clauses, and Oc [TaF86]. The fundamental scheme of this type of parallelism is the use of shared variables as a communication channel between two or more processes through unification. One

drawback, however, is that current implementations are restricted to deterministic Prolog programs [CoK85].

Finally, UNIFICATION-parallelism is examined by several researchers [TiW84] [MaU86] [DKM84] [MaM82]. Research by Mannila and Ukkonen [MaU86] illustrates that there is potential for increasing speed in the unification process as worst case sequential unifications require quadratic time. Their work outlines several methods for improving unification based on reducing the set-union problem [AHU 74] to the unification process in Prolog. The authors conclude, however, that an implementation of such a scheme would be complicated. Fundamental theoretical research by Dwork, Kanellakis, and Mitchell [DKM84] illustrates that although unification is a prime target for parallelism, it is inherently sequential, and thus, it is unlikely that any improvements in speed will result from parallel unification algorithms. The authors conclude that the special case of term matching, however, does have the potential for significant improvement in parallel execution.

The first two types of parallelism mentioned, AND-parallelism and OR-parallelism, are more attractive for their large granularity and hence applicability to entire logic programs. As a result, these are the most popular forms of parallelism [CoK85] [CoK81] [Her86] [Bor86] [CiH84].

AND-parallelism was chosen for this project over OR-parallelism for its ability to handle both deterministic and non-deterministic logic programs, its natural implementation on a multiprocessor shared-memory architecture, and the potential to keep the overhead costs of time and memory relatively low. OR-parallelism's improvement over sequential execution is limited to non-deterministic logic programs

and requires more overhead in the form of memory and copying time for structures. There are, however, algorithms that help control the overhead problems associated with OR-parallelism. The following paragraphs will compare the advantages and disadvantages of AND-parallelism and OR-parallelism and justify this choice.

1.2. AND vs. OR Parallelism

OR-parallelism is based on the principle that the clauses whose head unifies with the current goal are executed simultaneously. For example, if the goal of a Prolog program is:

```
uncle(roy, Y).
```

and the clauses whose heads match the goal are:

```
uncle(X, Y):- brother(X, Z), mother(Z, Y).
uncle(X, Y):- brother(X, Z), father(Z, Y).
```

then as the first clause is executed by the first process, another process executes the second clause. The point where a new process begins execution is called the *branching point.* In order to find all solutions to a query in OR-parallelism, both processes work independently to derive their own set of solutions, and the union of these sets is the set of solutions for the query. When only one or a few solutions are desired, the processes still work independently, and after the user is satisfied with the number of

answers found, all processes are terminated. No backtracking is required for pure OR-parallelism systems since each path of the program is executed and failing paths result in empty solution sets. OR-parallelism is attractive for its straightforward principle and was, until recently, the type of parallelism most often implemented for logic programming languages.

OR-parallelism is not, however, without its drawbacks. One major disadvantage is the high overhead of copy time and storage space required. Since each OR-process is independent, many OR-parallel systems rely on passing a copy of the complete state of work done prior to the branching point to a process in order for previous information to be accessible by that process. In addition, independent binding environments are kept by each process from the branching point forward. As the number of processes increases, more space is required to hold inherited information and more time is required for copying the growing environment to new processes. Soon, the overhead exceeds the benefits derived from OR-parallelism. This overhead, however, may be limited by specialized hardware and copy time may be reduced by only copying parts of the information and sharing others, as proposed by Warren and implemented by Overbeek, Gabriel, Lindholm, and Lusk [OGL85]. Another scheme for reducing memory costs in OR-parallelism is demonstrated in Ciepielewski and Haridi's OR-Parallel Token Machine [CiH84]. Their machine builds the proof tree in a depth first manner, to avoid the explosion of space required in a breadth first traversal of the proof tree, and removes the paths in the proof tree that are no longer necessary when a solution to a goal is found. That is, when a goal is satisfied, their OR-Token Machine removes all of the other paths in the tree that search for

the same solution as the goal just solved. This scheme frees memory as it is no longer needed and hence, reduces the memory and copying time overhead of new processes.

Another disadvantage to OR-parallelism is the potential for runaway processes. This problem rarely arises when only one solution to a query is desired, since the programmer terminates all processes when the first solution is given. But as more answers are required, it may become necessary for the programmer to decide if a running process is a runaway lost in useless work, or a slow process working on a difficult solution. In some cases, the Prolog program may be rewritten to avoid these tendencies, but many programs, such as those where the ordering of the clauses plays a crucial role, are not practicable for all-solutions OR-parallelism. AND-parallelism is able to avoid this problem more often than OR-parallelism since AND-parallelism executes the goals in the body of a clause in parallel, and thus the ordering of the clauses is maintained.

The main drawback of OR-parallelism is that its improvements are limited to highly non-deterministic programs. In order for a logic program to benefit from OR-parallelism, it must have many possible branching points. Deterministic programs, by my definition, do not have branching points. An example of a deterministic program is the one below that calculates the Fibonacci sequence. The user specifies a value for the variable X, representing the position in the sequence of Fibonacci numbers, and the program returns Y, the value of the number in position X. In this example, the fourth number in the Fibonacci sequence is requested.

```
query(Y):- fibonacci(4, Y).

fibonacci(0, 1).
fibonacci(1, 1).
fibonacci(X, Y):- X >= 2,
                  X1 is X - 1,
                  fibonacci(X1, Y1),
                  X2 is X - 2,
                  fibonacci(X2, Y2),
                  Y is Y1 + Y2.
```

OR-parallelism cannot improve upon sequential execution of this type of program.

AND-parallelism, on the other hand, provides beneficial results for deterministic programs in addition to non-deterministic programs. As mentioned earlier, AND-parallelism is the execution of several goals in a clause simultaneously to find a single solution. An example of AND-parallelism involves the goal

```
graduates(johndoe, masters).
```

and the clause

```
graduates(Person, Degree):-
                coursehours(Degree, NumOfHours),
                thesis_signed(Person, Advisor1),
                amount_owed_to_school(Person, 0).
```

The goal and clause head match, thus binding `Person` to `johndoe` and `Degree` to `masters` in the clause. At this point, the clause is:

```
graduates(johndoe, masters):-
                coursehours(masters, NumOfHours),
                thesis_signed(johndoe, Advisor1),
                amount_owed_to_school(johndoe, 0).
```

with three independent goals in its body. These goals must be independent when executed in parallel if variable binding conflicts are prevented.

AND-parallel execution begins when one process executes the goal `coursehours(masters,NumOfHours)`, another process executes the goal `thesis_signed(johndoe,Advisor1)`, and a third process executes the goal `amount_owed_to_school(johndoe,0)` simultaneously.

Now suppose the goal is:

```
graduates(Student, masters).
```

and the clause is the same as above. After matching the new goal and the clause head, the clause is:

```
graduates(Student, masters):-
                coursehours(masters, NumOfHours),
                thesis_signed(Student, Advisor1),
                amount_owed_to_school(Student, 0).
```

where `Person` is bound to `Student` and `Degree` is bound to `masters` throughout the clause. In this case, the goals in the clause body are not independent. If these dependent goals are executed in parallel, then the second and third processes will generate their own value for `Student`, but a unique value for

Student is required by the semantics of the clause. When dependent goals are executed in parallel, several processes may attempt to bind Student with different values. The first process to bind Student to a value sets the value of Student for all processes executing that clause. Processes that attempt to bind Student to a different value will fail. If the bound value is not semantically correct, then unnecessary work is done by all processes executing that clause in their attempt to match an incorrect value of Student. Thus, when dependent goals are executed in parallel and different values are found for a variable, a *variable-binding conflict* occurs.

Variable-binding conflicts and communication are the areas for expense in AND-parallelism. In order to prevent binding conflicts, variable dependency analysis must be done to ensure that the goals of a clause are independent. This analysis may be done at compile-time, runtime, or both. The problem with compile-time analysis is that not much information is available and the worst-case situation must often be chosen. For example, at compile time, the arguments in a specific clause may not be bound, but during runtime they are bound before reaching the clause. A compile-time algorithm would require sequential execution of the clause when it could have been executed in parallel. The runtime algorithm would allow the clause parallelism, but requires a large overhead for testing the clause. The RAP scheme, discussed in Chapter 3, reduces the large overhead of runtime costs and the inaccuracy of compile-time tests by conducting tests at both compile-time and runtime.

Communication is an area for potentially high overhead in AND-parallelism. Although processes must have some form of communication to inform other processes

when to backtrack and terminate, it need not be expensive. There are many possibilities for creating and sending messages among processes, but it is important to keep them small and infrequent. The overhead accrued through message passing is dependent on the architecture and the implementation, and should be kept to a minimum.

OR-parallelism and AND-parallelism have their relative advantages and disadvantages, but AND-parallelism was chosen as the focus of this project. AND-parallelism and OR-parallelism apply in different situations in Prolog programs; OR-parallelism searches for solutions in parallel while AND-parallelism works on parts of one solution in parallel. AND-parallelism requires backward execution, also called backtracking, when a failure occurs, whereas OR-parallelism does not engage in backward execution. One might conclude from this that AND-parallelism is slower since it backtracks and only produces one solution at a time, but this need not be the case. If the user desires only one solution to a query, then OR-parallelism becomes expensive. This expense in OR-parallelism is due to each OR-process generating large trees in search of many solutions, which in this situation is unnecessary, since only one solution is desired. AND-parallelism of independent goals does not produce this extent of unnecessary work. OR-parallelism also has a greater tendency for runaway processes and hence, potential difficulty for finding all solutions. In addition, OR-parallelism is slowed by the copying of information at each branching point. The final drawback of OR-parallelism is its inability to improve upon the sequential execution of deterministic programs.

1.3. Goals

The goal of this Master's thesis is to design and implement an efficient parallel Prolog opcode-interpreter that exploits AND-parallelism on a shared-memory multiprocessor architecture. The user is responsible for incorporating parallel expressions in the Prolog program, but is not burdened with excessive specifications and declarations for parallelism. Finally, this implementation should produce an increase in performance as the number of processes increase.

1.4. Outline

Chapter 2 presents an overview of the components of the Parallel Prolog machine: the compiler, the assembler, the opcode-interpreter, and Chapter 3 discusses variable-binding conflicts. The opcode-interpreter's forward sequential and parallel execution are analyzed in Chapter 4, and Chapter 5 focuses on the opcode-interpreter's backward sequential and parallel execution. Chapter 6 reports and analyzes results of test programs and Chapter 7 concludes this research and presents suggestions for further research.

CHAPTER 2

Introduction to the Parallel Prolog Machine

The first section of this Chapter presents an overview of the components of the Parallel Prolog Machine while the second section discusses variable-binding conflicts and methods of detecting these conflicts. The third section presents the variable-binding conflict detection method implemented in the parallel Prolog opcode-interpreter (PAPI), and finally, the fourth section presents several example programs.

2.1. The Parallel Prolog Machine

The Parallel Prolog Machine consists of three components; the compiler, the assembler, and the interpreter. A Prolog program is first compiled into intermediate-code instructions by a compiler written in C-Prolog. The intermediate-code instructions produced by the compiler include the instructions implemented by D.H.D. Warren in his Warren Abstract Machine in addition to several instructions specific to parallel execution. The set of intermediate-code instructions can be found in Appendix A of D.H.D. Warren's dissertation [War77] and Appendix A of this thesis.

After compiling the Prolog source code into intermediate-code instructions, the assembler, also written in C-Prolog, translates each intermediate-code instruc-

tion into an opcode instruction. An opcode instruction consists of a list of an uppercase letter, an integer between 1 and 100, and the intermediate-code instruction's arguments. The uppercase letter represents the type or characteristic of the instruction; for example, P specifies that the instruction is an entry to a procedure, M is for an instruction with three arguments, and C specifies a procedure call instruction. The integer value represents an intermediate-code instruction and the integer value is followed by the arguments of the intermediate-code instruction. As an example, intermediate-code instructions and opcode instructions for the `fibonacci` program presented in Chapter 1 are provided in Appendix B.

The first implementation of the opcode-interpreter was written in "C" by Doris Rea, Robert Herndon, and Peter Borgwardt at the University of Minnesota. Their opcode-interpreter executed stack-based sequential Prolog, except backtracking and the cut operation, which were not completed. Using this sequential stack-based opcode-interpreter for a backbone, I have added AND-parallelism and backtracking to create, PAPI, a parallel Prolog opcode-interpreter.

PAPI consists of about 7,500 lines of "C" code and runs on a Balance[1] 21000 made by Sequent Computer Systems. PAPI begins execution by initializing its environment for parallelism and loading the opcode file created by the assembler for the Prolog program into memory. The number of processes specified by the user are created, but all except the parent are put to sleep with the `sigpause(0)` system call. The opcodes are interpreted by the parent process and when parallelism is applicable, the child processes are awakened with the `kill(SIGALRM)` system call

[1] Balance and DYNIX are registered trademarks of Sequent Computer Systems.

and put to work, resulting in parallel execution of the Prolog program. Parallel execution ends after all solutions to the query have been found or the user does not desire more solutions. At this point, the processes are terminated.

Before continuing with a more detailed description of PAPI's execution, the problem of detecting variable-binding conflicts must be addressed and resolved. The following Chapter is dedicated to the variable-binding conflict issue.

CHAPTER 3

Variable-Binding Conflicts

As mentioned in Chapter 1, implementing AND-parallelism presents the potential for variable-binding conflicts. Although AND-parallelism does not require that these conflicts be avoided, encountering a variable-binding conflict can greatly reduce the benefits of parallelism. Since variable-binding conflicts are easily avoided by executing only independent goals in parallel, the cost involved with conflict detection is minimal compared to the cost when these conflicts arise. Therefore, a variable-binding conflict detection scheme is justified, and of course, the scheme with the lowest overhead is preferred.

3.1. Variable-Binding Conflict Detection

There are several approaches for detecting and preventing variable-binding conflicts. One approach requires that the programmer determines and specifies which goals in the Prolog program are guaranteed independent. This scheme, used in Delta Prolog and PARLOG, places the burden of variable-binding conflict detection in the hands of the programmer. Although this solution is acceptable for many applications, it violates the goal to keep the programmer free from excessive specifications. Therefore, other approaches that limit the programmer's involvement are preferred. These approaches involve algorithms which detect variable-binding

conflicts at runtime, compile-time, or both.

Conery employs a runtime variable-binding conflict detection scheme in his AND-process model [Con83]. By executing a series of runtime algorithms with some user specifications, he creates *dataflow* graphs. These graphs depict generator and consumer relationships[1], and determine goal ordering for the clause. The *dataflow* graphs also produce information determining if variables alias one another or are bound to values that share variables. Aliasing may occur in the following situation. If a Prolog program contains the clause:

```
a(X, Y):- b(X),c(Y).
```

it appears that `X` and `Y` are independent, hence permitting `b(X)` and `c(Y)` to be executed in parallel without variable-binding conflicts. Yet, `X` and `Y` may not be independent. For example, if the goal matching the clause head, `a(X, Y)`, is any one of the goals:

```
a(P, P).
a(P, g(P)).
a(g(P), h(2, P)).
```

then at runtime `X` and `Y` are aliases or share at least one variable between them. This type of variable-binding detection requires runtime testing and Conery's runtime scheme is efficient in detecting aliases. In addition, backward execution, or backtracking, traverses the dataflow graph to fail previous goals. Although Conery's

[1] When two or more goals have a variable in common, the goal whose variable is bound first is the generator of the variable while other goals containing the variable are consumers of the variable. The *dataflow* graphs determine which variables will be bound when the clause is reached, and hence, which goals in the body of the clause are independent.

algorithms extract all information relevant to parallelism and create the dataflow graph, there is an enormous amount of runtime support required for implementing and maintaining the graphs. This amount of required runtime support arises from the fact that the graphs must be recomputed when involved variables are bound and when work is redone during backward execution. Thus, this method is too expensive and does not limit overhead as desired.

A compile-time analysis scheme has been proposed and discussed by Chang, Despain, and DeGroot [CDD85]. This method, *Static Dependency Analysis* (STDA), involves the generation of *data dependency* graphs for each clause and each goal in the body of the clause by the *data dependency analyzer*. The analyzer also produces information for backward execution. The programmer, however, must supply the analyzer with information, such as, which queries are the entry points to the program and the state of the arguments when the query is called (i.e. *ground*, *independent*, or *dependent*). The graphs will then provide information as to which goals are independent and may be executed in parallel, in what order to execute dependent goals, and a scheme for backtracking semi-intelligently. Although this technique has the advantage over the previous method of avoiding high runtime overhead, it is based on a worst-case situation. Since very little information about variable-bindings is available at compile-time, there is a strong possibility that parallelism will be missed. In addition, the worst-case analysis and entry point declarations are not able to detect aliasing as accurately as Conery's runtime scheme does. STDA's inefficient alias detection may result in variable-binding conflicts despite work done by the detection algorithm. Intelligent backward execution also suffers as a result of

the worst-case analysis. This method has the advantage of reduced overhead, but forsakes parallelism and misses conflicts only detectable at runtime.

The final scheme is a combination of compile-time and runtime detection. DeGroot takes this approach in his Restricted AND-Parallelism (RAP) technique [DeG84] [DeG85]. At compile time, his *typing algorithm* assigns a *type* to each of the arguments in each of the clauses. There are three possible *types*:

type 1: a ground argument
such as `pop` in the goal `drink(Person, pop).`

type 2: a non-ground, non-variable argument
such as `[A, 1, 2, 3]` in the goal `list([A, 1, 2, 3]).`

type 3: a variable argument
such as `Person` in the goal `drink(Person, pop).`

As the STDA method makes worst-case type assignments among clauses, the RAP scheme makes worst-case type assignments within each clause due to the lack of binding information at compile-time. It is too expensive, as proved by Conery's method, to utilize only runtime algorithms for assigning variables. Therefore, DeGroot avoids the runtime overhead by assigning worst-case types to arguments at compile-time and then permitting arguments to *inherit* lower-valued types through normal runtime execution.

Type inheritance occurs at runtime during the matching (or unification) process. An example for the goal:

```
drink(tom, milk).
```

with argument type 1 for both `tom` and `milk` assigned during compile-time analysis, and the clause

```
drink(Person, milk):- likes(Person, milk),
                      thirsty(Person).
```

which has compiler assigned type 1 for `milk` and type 3 for `Person` (assigned at compile-time) is as follows: during forward execution, `drink(tom, milk)` and `drink(Person, milk)` are matched. The argument `Person` becomes bound to `tom` and the type of `tom`, type 1, is inherited by `Person`. Since the second argument, `milk`, is the same in both the goal and the clause head, no bindings are made and no type inheritance is done. After unification, the clause has arguments of type 1 only.

Another example matches the goal:

```
drink(Someone, milk).
```

with the clause above. In this case, the argument `Person` is bound to `Someone` and if the type number of `Someone` is less than that of `Person`, then `Person` inherits `Someone`'s type. Otherwise, no type inheritance is done since the types are the same. Again, `milk` has the same type in both goals so no binding or type inheritance occurs for the argument.

As in the conflict detection methods mentioned above, the RAP scheme creates an *execution graph* at compile-time. RAP differs from the previous schemes by utilizing *execution graph expressions* in the execution graph to express potential

parallelism of the clause. These expressions are evaluated at runtime and alleviate the need for more than one graph to be created. The six execution graph expressions employed by DeGroot are:

```
g
(seq E1 E2 ...)
(par E1 E2 ...)
(gpar(X1, X2, ...) E1 E2 ...)
(ipar(X1, X2, ...) E1 E2 ...)
(if E1 E2 E3)
```

where `g` represents a single goal to be executed, `seq` requires the goals `E1 E2 ...` to be executed sequentially, and `par` requires the goals `E1 E2 ...` to be executed in parallel. The `gpar` and `ipar` expressions indicate either sequential or parallel execution, depending on the types of arguments `X1,X2 ...` tested at run-time. If the `gpar` arguments are all type 1, then goals `E1 E2 ...` are executed in parallel, otherwise they are executed sequentially. If all of `ipar`'s arguments are independent of each other, then the goals following `ipar` are executed in parallel, otherwise they are executed sequentially. And finally, the `if` expression indicates that goal `E2` is executed if the Boolean goal `E1` evaluates to `true`, otherwise goal `E3` is executed.

At compile-time, DeGroot's method assigns types to each of the arguments based on worst-case conditions and creates an execution graph with execution

expressions. The `seq` and `par` expressions are employed when the dependency of the variables is known at compile-time, but if it is not known, instead of assuming the worst case of sequential execution, `ipar` and `gpar` are utilized and the applicable arguments are listed. At runtime, arguments may inherit better types through unification and when the execution expressions `ipar` and `gpar` are reached, their argument types are examined to determine how execution will proceed. This scheme is also able to detect aliasing of variables by testing the variable's *types* at runtime and determining the dependencies. However, it does not extract as much parallelism as Conery's scheme. For example, the compiler may fail to find parallelism, due to the approximations made by the typing algorithm, as explained by DeGroot [DeG85], and there may be a loss of parallelism as a result of the limited execution graph expressions. DeGroot's RAP scheme combines the positive aspects of Chang's and Conery's ideas, but the implementation has less compile-time overhead and overcomes some of the drawbacks encountered by Chang and Conery.

3.2. A Modified RAP Scheme

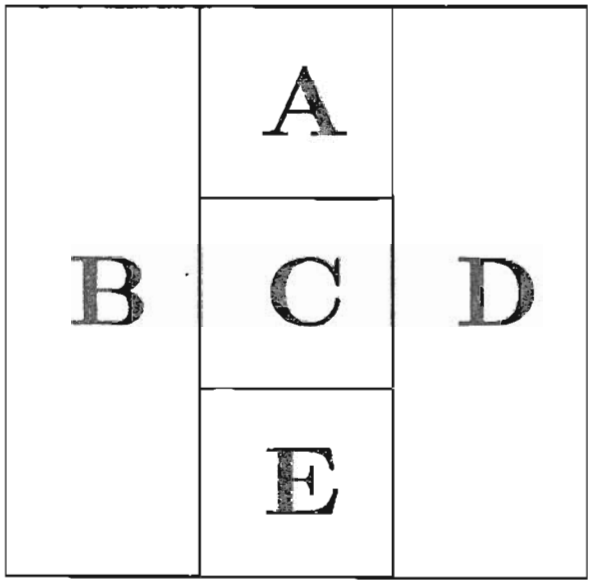
The variable-binding conflict detection method implemented in PAPI is a slight variation of DeGroot's RAP scheme. One small difference is that types are referred to as *ground*, *complex*, and *variable*, rather than type 1, type 2, and type 3. The major difference is that some of the compiler's responsibilities are shifted on to the programmer. Since the purpose of this project is to implement a parallel Prolog opcode-interpreter, the compiler was not modified to perform the variable-dependency analysis required by DeGroot's scheme. Rather, PAPI sets and modifies

the argument types and the programmer creates the execution graphs by inserting execution graph expressions into the Prolog program as it is written. If no execution graph expressions are present, then PAPI assumes sequential execution.

The optimal variable-binding conflict detection scheme, however, would be to permit the programmer to specify the execution graph expressions desired and implement the compiler such that it recognizes these specifications and adds more execution graph expressions where applicable. This scheme would provide the programmer with the option of specifying all, some, or none of the execution graph expressions while the compiler completes the task.

3.3. Example Programs

This modified RAP scheme is best illustrated through examples. The first example program, `mapcolor`, solves a map-coloring problem for a map with five regions:



and three colors:

```
red
blue
yellow
```

Using the colors above, the problem is to fill in each of the five regions of the map with a color, such that neighboring regions do not have the same color. That is, if region B is `red`, then regions C, A and E cannot be `red` since they are neighbors of region B.

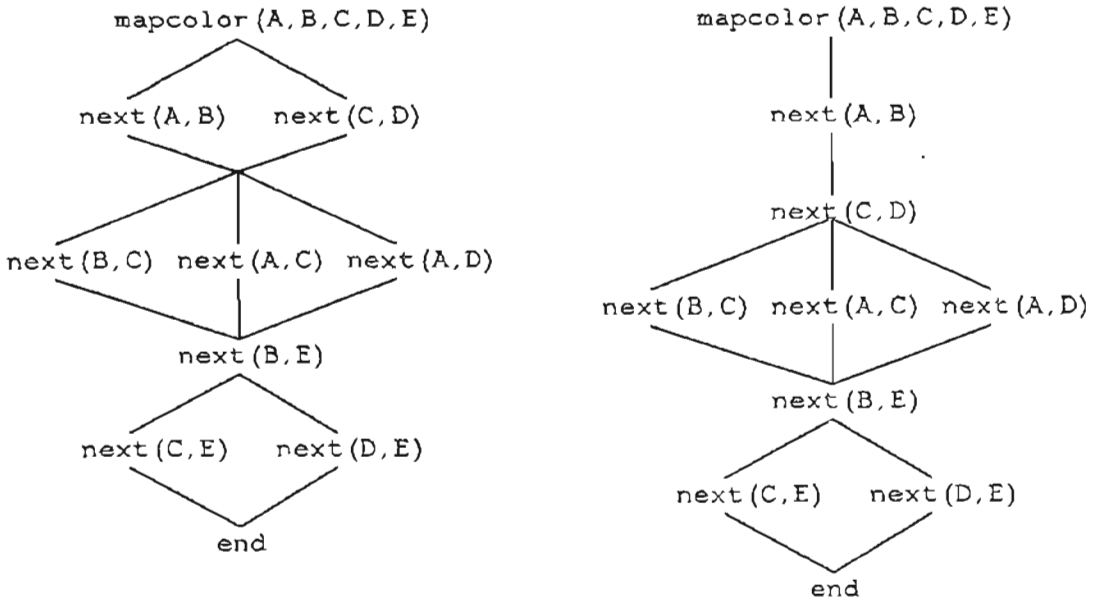
The sequential version of `mapcolor` is given below. The variable arguments in the query and `mapcolor` goal represent the five regions of the map, the colors are specified in the facts, and the `next()` goals specify the relationship of the regions.

```
query(V, W, X, Y, Z):- mapcolor(V, W, X, Y, Z).

mapcolor(A, B, C, D, E):- next(A, B),
                          next(C, D),
                          next(B, C),
                          next(A, C),
                          next(A, D),
                          next(B, E),
                          next(C, E),
                          next(D, E).

next(red, blue).
next(blue, red).
next(yellow, red).
next(red, yellow).
next(blue, yellow).
next(yellow, blue).
```

The sequential mapcolor program is modified to a parallel Prolog program by adding execution graph expressions as follows:

```
query(V, W, X, Y, Z):- mapcolor(V, W, X, Y, Z).

mapcolor(A, B, C, D, E):- gpar([A, B, C, D],
                               next(A, B),
                               next(C, D)
                              ),
                          par(
                               next(B, C),
                               next(A, C),
                               next(A, D),
                               seq(
                                   next(B, E),
                                   par(
                                       next(C, E),
                                       next(D, E)
                                      ),
                                  ),
                              ).

next(red, blue).
next(blue, red).
next(yellow, red).
next(red, yellow).
next(blue, yellow).
next(yellow, blue).
```

The two execution graphs for the parallel version of `mapcolor` are given below. The graph to the left occurs when `gpar` succeeds and turns into a `par`, while the graph to the right occurs when `gpar` fails and requires sequential execution.



Mapcolor's Execution Graphs
Figure 3.1

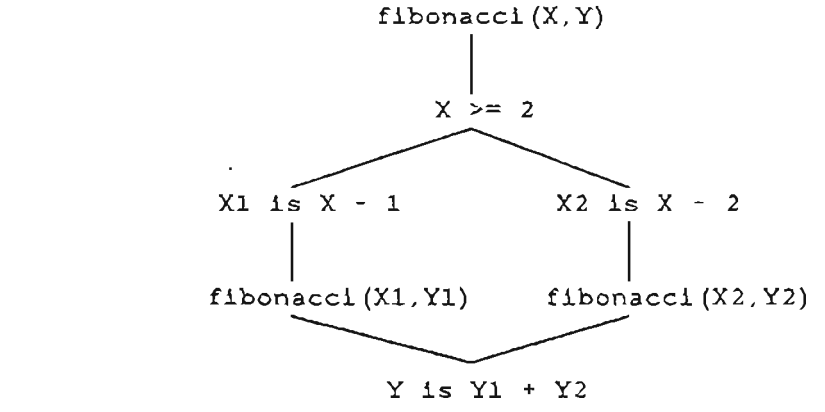
This use of gpar and par in mapcolor is beneficial, as the programmer knows that the goals after the par are independent (since the next(A, B) and next(C, D) goals prior to the par bind the arguments A, B, C, and D). Thus, it is not necessary to implement gpar in place of par and require that PAPI test the arguments of the goals when the programmer is sure that the arguments are bound. In some instances, however, it may be preferrable to use gpar rather than par. One such instance is the gpar before the first goal in the body of the clause. The arguments A, B, C, and D may be variables or values in the query. If all of

the arguments are values, then `par` is the best choice, but if at least one argument is a variable, then the goals must be executed sequentially. Since the programmer writing the `mapcolor` program is unable to predetermine the type of the arguments that will be specified in `mapcolor`'s query, `gpar` is the preferred execution graph expression for this situation.

The sequential `fibonacci` program, presented in Chapter 1, is rewritten for parallel execution with execution graph expressions as follows:

```
query(X, Y):- fibonacci(4, Y).

fibonacci(0, 1).
fibonacci(1, 1).
fibonacci(X, Y):- X >= 2,
                  par(
                      seq(
                          X1 is X - 1,
                          fibonacci(X1, Y1)
                         ),
                      seq(
                          X2 is X - 2,
                          fibonacci(X2, Y2)
                         ),
                     ),
                  Y is Y1 + Y2.
```

The execution graph for `fibonacci` is:



Fibonacci's Execution Graph
Figure 3.2

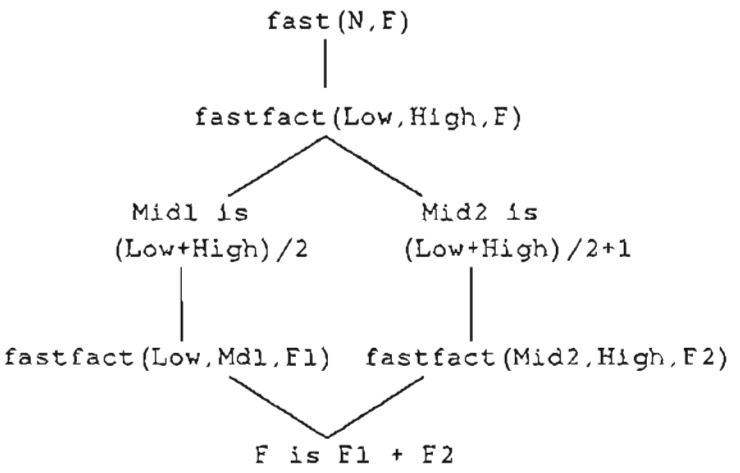
The final example is the program `fastfact`, which returns the factorial of an integer argument specified by the programmer. `Fastfact` is not the most efficient program for calculating the factorial of an integer, but its inefficiencies provide work for parallel execution. The sequential version of `fastfact` is:

```
query(F):- fact(4, F).

fact(N, F):- fastfact(1, N, F).

fastfact(N, N, N).
fastfact(Low, High, F):- Mid1 is (Low + High)/2,
                         Mid2 is (Low + High)/2+1,
                         fastfact(Low, Mid1, F1),
                         fastfact(Mid2, High, F2),
                         F is F1 * F2.
```

The second goal in the body of the `fastfact(Low, High, F)` clause may be moved to the position after goal `fastfact(Low, Mid1, F1)`, without changing the semantics of the `fastfact` program. Changing the order of the goals in the body of the clause permits parallelism in the program as seen below:

```
query(F):- fact(4, F).

fact(N, F):- fastfact(1, N, F).

fastfact(N, N, N).
fastfact(Low, High, F):-
                gpar([Low, High],
                     seq(
                         Mid1 is (Low + High)/2,
                         fastfact(Low, Mid1, F1)
                        ),
                     seq(
                         Mid2 is (Low + High)/2+1,
                         fastfact(Mid2, High, F2)
                        ),
                    ),
                F is F1 * F2.
```

The execution graph for `fastfact` is:



Fastfact's Execution Graph
Figure 3.3

CHAPTER 4

Forward Execution of PAPI

Forward execution of the stack-based opcode-interpreter is presented in two parts. First, single process memory management and sequential execution are discussed, and then multiple process memory management and parallel execution over distributed stacks are presented. Sequential execution of PAPI closely follows D.H.D. Warren's stack-based method for executing compiled Prolog [War77] [War83].

4.1. Single Process Memory Management

The data structures employed in PAPI's sequential execution include a *goalist* of goal structures, a *local* stack, a *global* stack, a *trail* stack, and a *goal* stack. A goal structure is allocated for a goal when the goal becomes the current goal to be matched with a clause head. Each goal structure holds information for sequential forward and backward execution of the goal for which it was created. Most of the information stored in a goal structure is in the form of pointers to other goal structures or pointers into the stacks listed above. The proof tree, for example, is represented through the forward sibling, backward sibling, parent, right child, and left child pointers maintained in each goal structure.

Each goal structure is allocated from the *goalist*, a large static array of goal structures. The *goalist*'s index is incremented as each goal structure is allocated, but is never decremented. Thus, memory for goal structures discarded through

backward execution is not reallocated for new goal structures. This method of allocation results from the costs involved with marking a discarded goal structure in the *goalist* "available" and then searching the large array of memory for an available goal structure each time PAPI allocates a goal structure. In addition, since goal structures contain pointers to other goal structures, once a goal structure is allocated, it cannot be moved within the *goalist*. As a result, PAPI may run out of space in the *goalist* during forward execution and be forced to terminate. This problem is discussed in more detail below.

As a clause head is unified with a goal, each argument in the clause is put in a record in the *local* stack and a pointer in the clause head's goal structure points to its arguments in the *local* stack. The *local* stack's records include fields for the argument type and the reference pointer for the argument. The argument type is either *ground*, *variable*, or *complex*, which is assigned to the argument by the modified RAP scheme. The reference field is a pointer to another record on the *local* stack if the type is *variable*, a pointer into the *global* stack if the type is *complex*, or a pointer to a value in a symbol table if the argument is *ground*.

The *global* stack contains the *complex* arguments, such as lists or functions. The fields in the records of the *global* stack are the same as those in the *local* stack. The argument type field is *ground*, *variable*, or *complex* (for structures within structures) and the reference field is a pointer to values or other records in the *global* or *local* stacks.

The *local* and *global* stacks share a single static array in memory. The *local* stack occupies the top portion of the array with the index value 0 as bottom of

stack, while the *global* stack resides on the lower portion of the static array with the index value ENDOFARRAY as its bottom of stack. A pointer is maintained for each stack by PAPI, *Ltop* and *Gtop*. As an argument is pushed onto the *local* stack during forward execution, Ltop is incremented, but if an argument is pushed onto the *global* stack, Gtop is decremented. Backward execution often pops the *local* and *global* stacks, thus maintaining the stack characteristics and preventing holes from occurring in either stack. It may occur, however, that the Ltop and Gtop meet at some point in the array structure. Thus, due to the declaration of the array, the stacks have run out of space and PAPI must terminate. This occurrance is discussed below.

The *trail* stack is a stack of pointers to arguments in the *local* and *global* stacks that are not local to the current clause, but are bound during the matching process of the current clause. PAPI's backward execution relies on the binding information stored in the *trail* stack to unbind arguments while backtracking. As the arguments are unbound, the record in the *trail* stack for that argument is popped from the stack. Thus, the *trail* stack does not have holes. The *trail* stack is a static array in memory with an index pointer, *TRtop*, maintained by PAPI. The *trail* stack may also run out of space during forward execution and force the early termination of PAPI.

The *goal* stack is a stack of pointers that point to goal structures in the *goalist*. The *goal* stack records the goals executed by PAPI and the order in which they were executed. As a goal structure is allocated for a goal from the *goalist*, a pointer to that new goal structure is pushed onto the *goal* stack. The ordering of the pointers to the goal structures in the *goal* stack is important as it aids in

determining the location of goal structures in the proof tree without the overhead of following sibling and parent pointers across individual goal structures. During backward execution, PAPI removes goal structures from the proof tree, by reseting appropriate goal structures' parent and sibling pointers, and pops the goal structures' records from the *goal* stack. The *goal* stack is a static array with an index pointer, *goaltop*, maintained by PAPI. Due to the static declaration of the *goal* stack, PAPI may be forced to terminate execution if more space is required.

If PAPI runs out of space in any of the static arrays of memory allocated for the *goalist* or the *local, global, trail* or *goal* stacks during execution, an error message is issued and execution terminates. At this point, the user must redeclare the size of the offending static array. Note that this problem also occurs in C-Prolog.

The *local, global, trail,* and *goal* stacks encounter fewer problems with this memory allocation scheme than the *goalist* since the stack arrays, unlike the *goalist*, are cleaned during backward execution. There is no unuseable empty space where outdated bindings of arguments or old goal structures reside in the stacks, rather, records in the stacks are reallocated in forward execution. Reallocation of memory is much easier in the *local, global, trail,* and *goal* stacks as the stack characteristic is maintained. Due to semi-intelligent backtracking, goal structures are not allocated from the *goalist* in a stack-based order. As a result, reusing memory in the stack structures is efficient, but is not efficient in the *goalist*.

4.2. Forward Sequential Execution

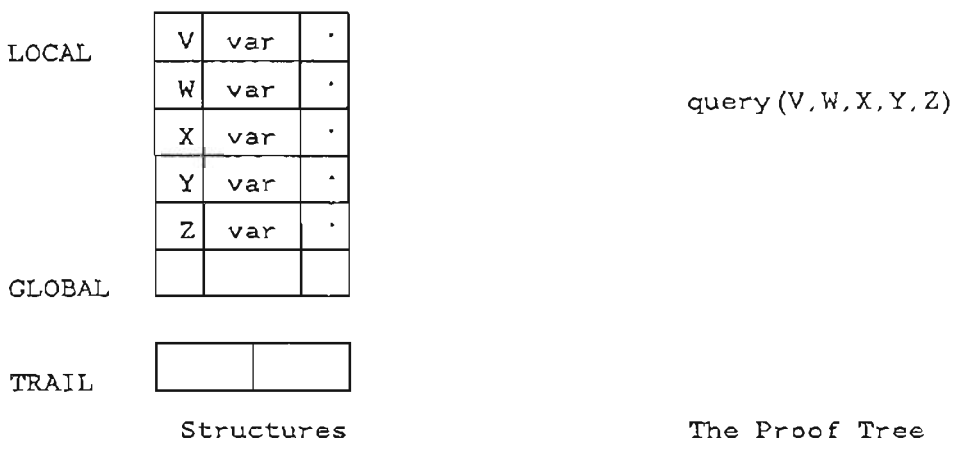
Forward execution of PAPI is the mechanism for matching goals, binding variables, and expanding the proof tree. It continues until all goals in the proof tree are proven `true` and hence, a solution is found, or until a goal fails and backward execution is initiated.

Forward execution begins with the query, the first goal, that the program will prove `true`, fail to prove `true`, or solve. To execute the query, which is the root of the proof tree, PAPI searches for the first clause head that matches the query. The query and matching clause head are unified, the first goal in the clause body is made into a goal structure, and a pointer to the goal structure is put on the *goal* stack. The bindings made in unification are put on the *local* and *global* stacks. The goal structure is put in the proof tree in a depth first manner, built from left to right, making it the leftmost child of the query. The newest goal structure in the proof tree is executed next under the bindings made in the unification. When a goal is found `true`, (i.e., matched with a fact) PAPI moves up a level in the proof tree to the parent of the `true` goal. If there is a goal following the `true` goal in the `true` goal's parent's clause, that goal is made into a goal structure and put in the proof tree as the forward sibling of the current goal and this newest goal is executed. In the case that the `true` goal is the last goal in the clause body, PAPI moves up another level to the grandparent of the `true` goal and looks at that clause body for the next goal to execute. When PAPI returns to the root of the proof tree (the query) and finds that all of the goals in the query have been executed, PAPI declares a solution or *success*. See Figure 4.7 for an example of a proof tree.

An example of forward sequential execution of the program `mapcolor` follows. In order to avoid discussion of backward execution at this point, only the execution of successful goals is shown. Consequently, the *goal* stack is not relevant and will not be illustrated in the diagrams.

4.3. Forward Sequential Execution Example

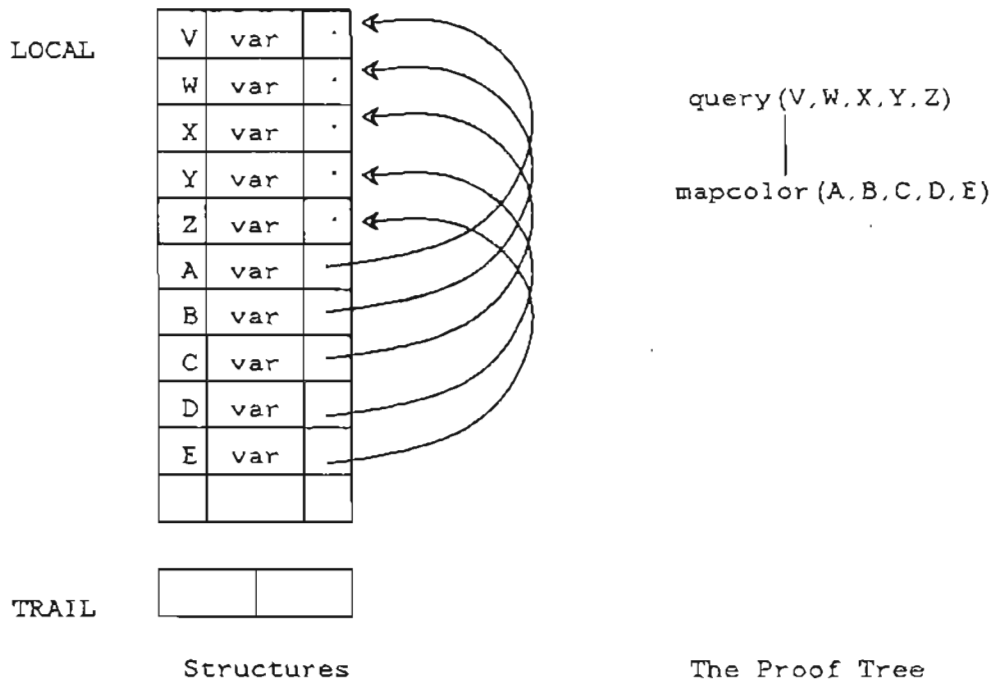
Before execution begins, the query's goal structure is created, and the *local* stack is initialized for the query's arguments.



The Structures and Proof Tree
Figure 4.1

The query `mapcolor(V, W, X, Y, Z)` matches clause head `mapcolor(A, B, C, D, E)`. A goal structure is created for the `mapcolor(A, B, C, D, E)` goal

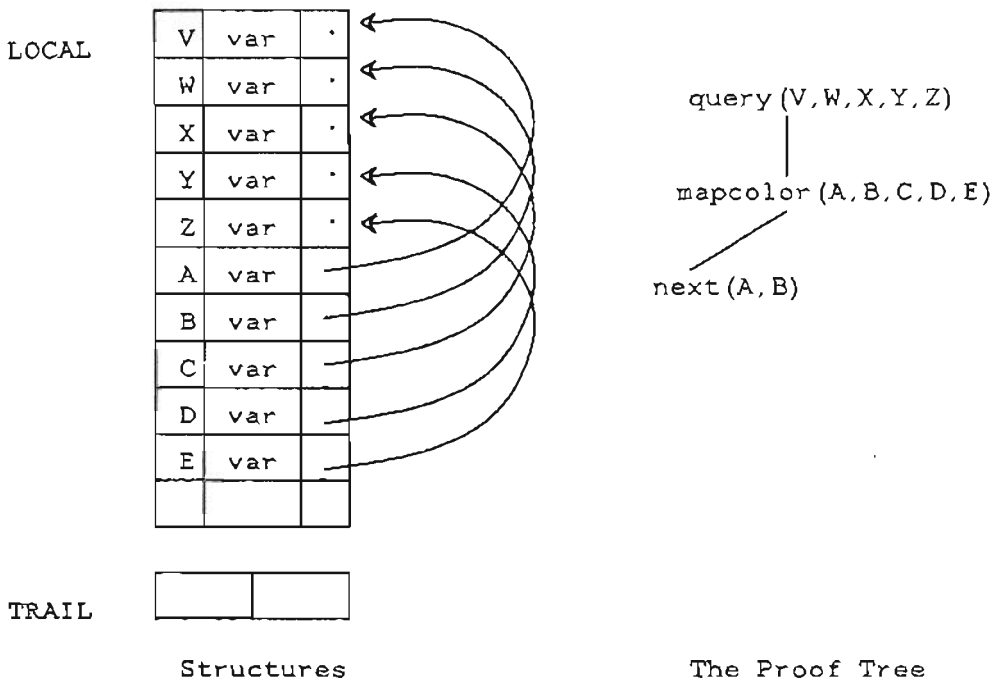
and put in the proof tree as the left child of `query(V, W, X, Y, Z)`. The arguments `A, B, C, D` and `E` are put on the *local* stack with type *variable* and reference pointers pointing to the corresponding arguments of `query(V, W, X, Y, Z)` on the stack. In order to keep the examples simple, the example programs do not contain lists or functions. Thus, the *global* stack is not utilized and hence, is not illustrated in the diagrams.



The Structures and Proof Tree
Figure 4.2

The first goal in the clause body, `next(A, B)`, is the current goal and it is put in

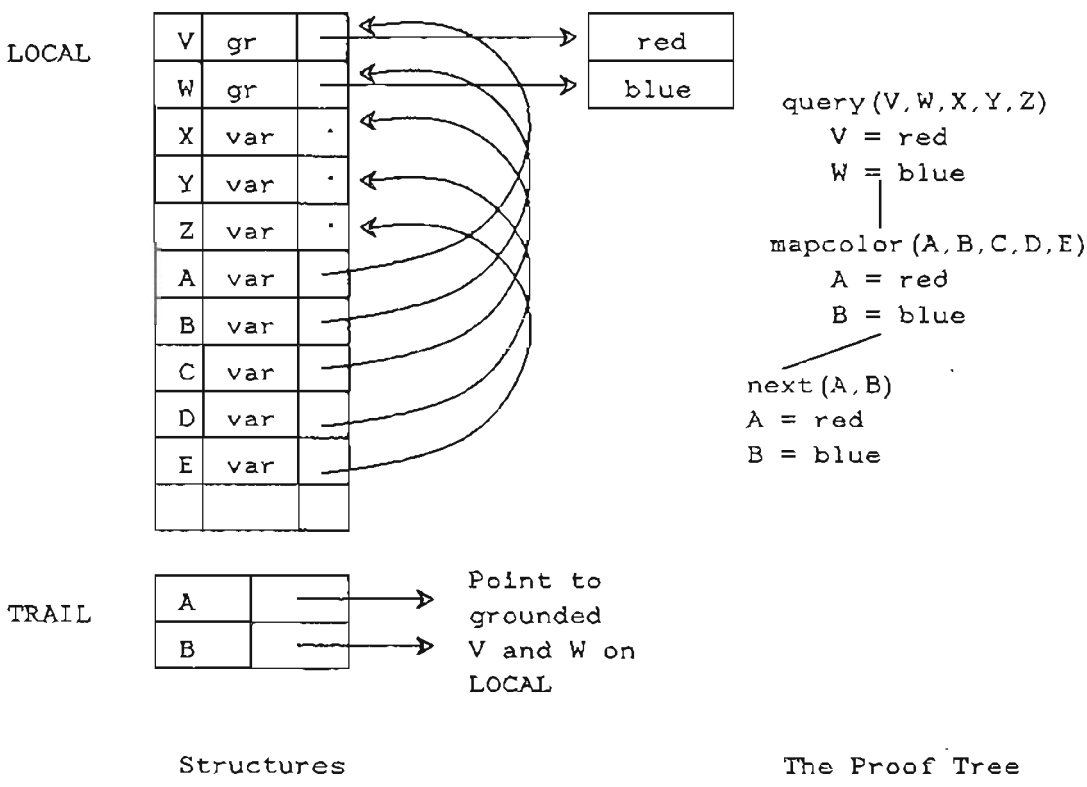
the proof tree as the first child of `mapcolor(A, B, C, D, E)`. Since `next(A, B)`'s arguments are variables of `mapcolor(A, B, C, D, E)`, they are already on the *local* stack.



The Structures and Proof Tree
Figure 4.3

The current goal `next(A, B)` matches the fact `next(red, blue)`, binding `A` to `red` and `B` to `blue`. The binding is made by following the reference pointers of `A` and `B` in `mapcolor(A, B, C, D, E)`'s arguments and continuing up the *local* stack to the query's arguments `V` and `W`. These topmost variables are bound to

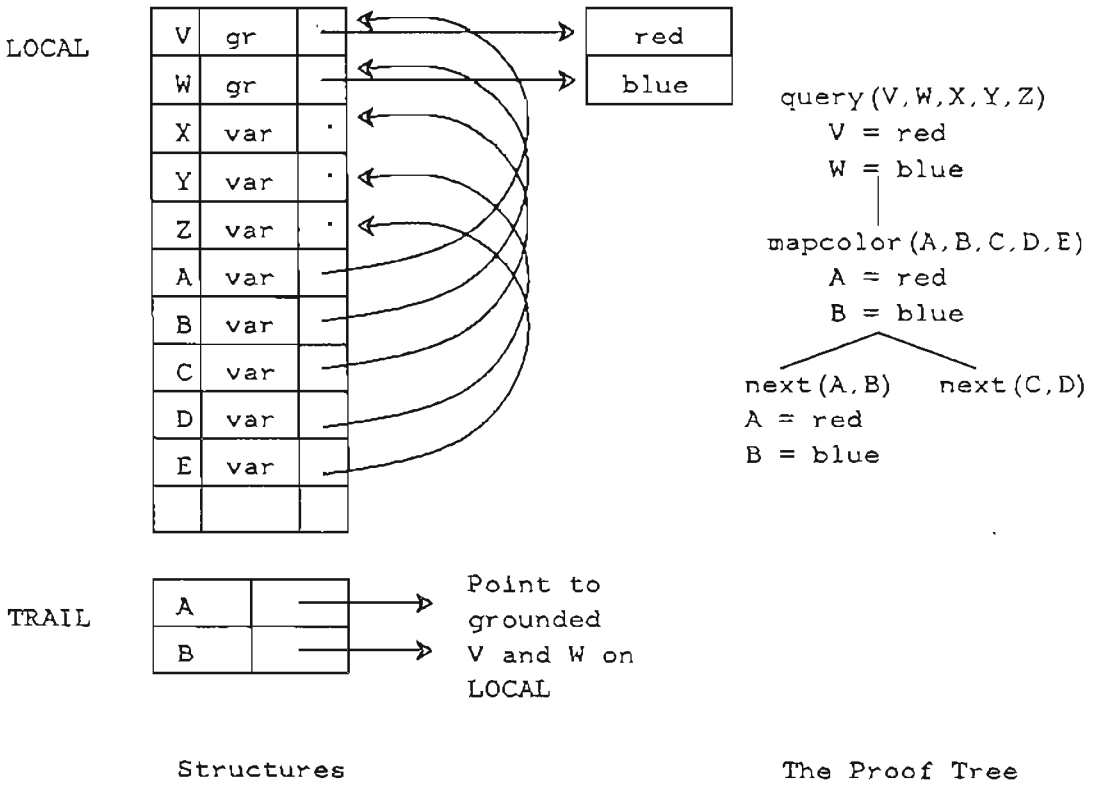
red and blue respectively, by changing their type to *ground* and their reference pointers from *nil* to the values red and blue. After binding these variables, PAPI records the bindings on the *trail* stack by creating records for the variables A and B and pointing the reference pointers to the newly bound variables.



The Structures and Proof Tree
Figure 4.4

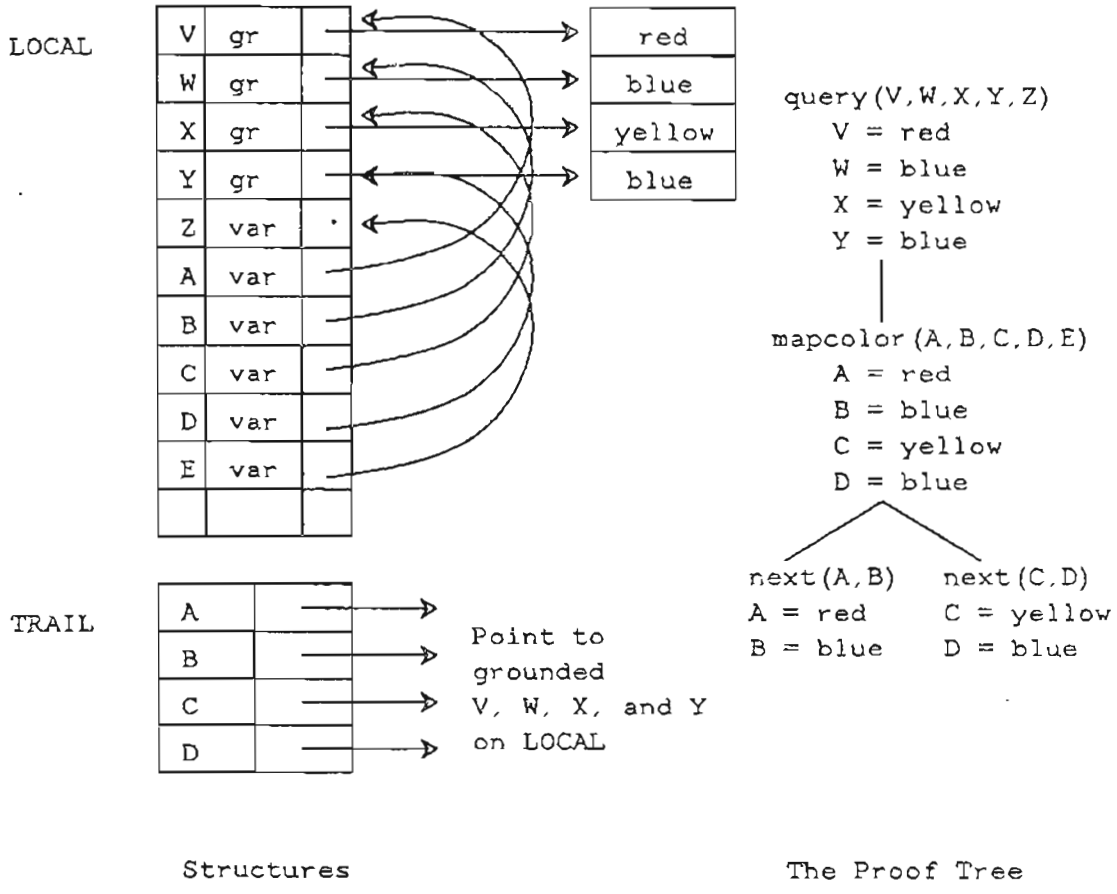
Since the goal next(A, B) succeeded, PAPI moves up a level in the proof tree to mapcolor(A, B, C, D, E) and looks for the next goal in the body of its clause.

The goal next(C, D) is put in the proof tree and again, its arguments are already on the stack when it is executed.



The Structures and Proof Tree
Figure 4.5

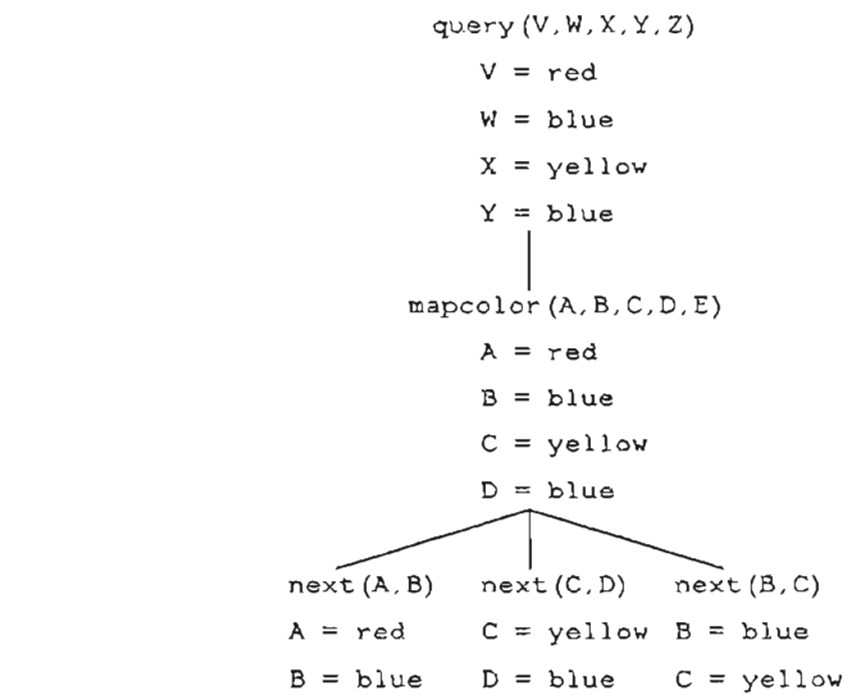
The goal next(C, D) matches the fact next(yellow, blue), binding C to yellow and D to blue in the same manner as A was bound to red and B was bound to blue.



The Structures and Proof Tree
Figure 4.6

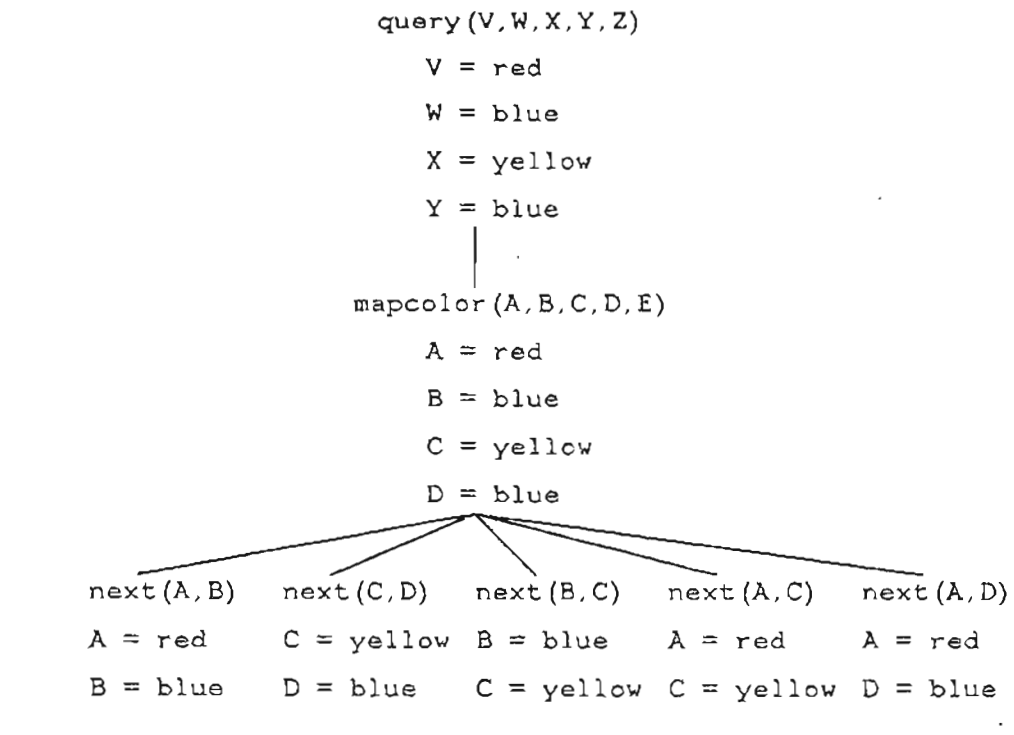
Again, PAPI moves up a level in the proof tree from the `true` goal to `mapcolor(A, B, C, D, E)` and looks for the next goal in the clause to execute. The next goal, `next(B, C)`, matches the fact `next(blue, yellow)` and is put in the proof tree. Since the arguments `A`, `B`, `C`, and `D` are bound, the *local* stack

and *trail* stack remain as in Figure 4.6 until argument E in the goal next(B, E) is bound. Thus, only the proof tree will be presented in the diagrams until next(B, E) is executed.



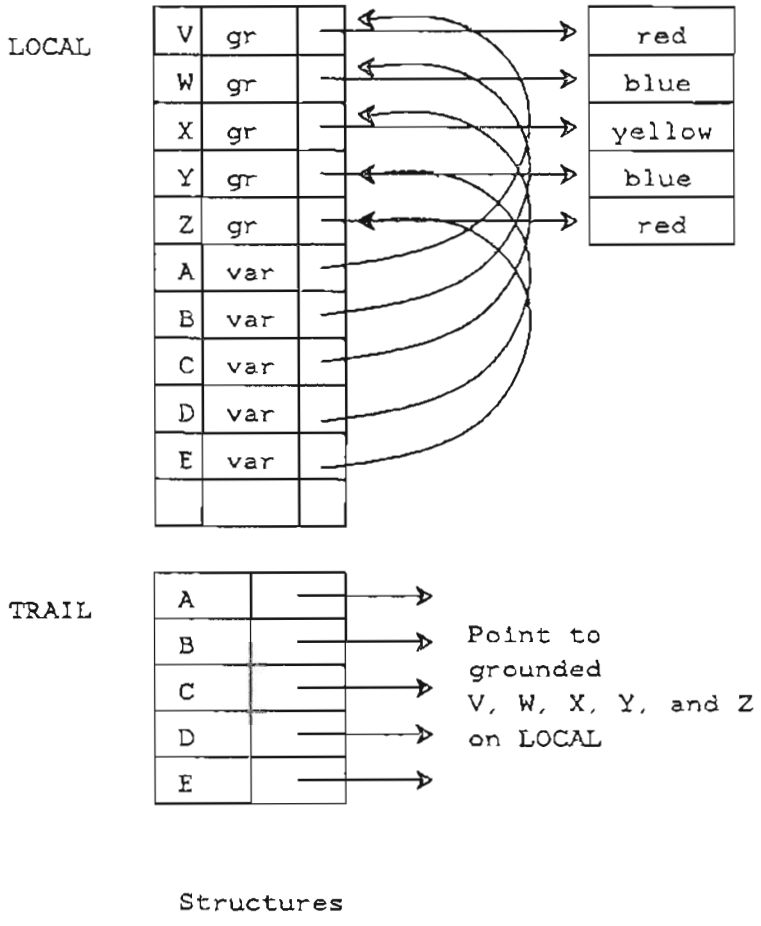
The Proof Tree
Figure 4.7

PAPI continues forward execution with the next goal in mapcolor(A, B, C, D, E)'s clause, next(A, C), which matches the fact next(red, yellow) and then executes next(A, D), which matches the fact next(red, blue). These goals are put in the proof tree, expanding the proof tree as illustrated in Figure 4.8.



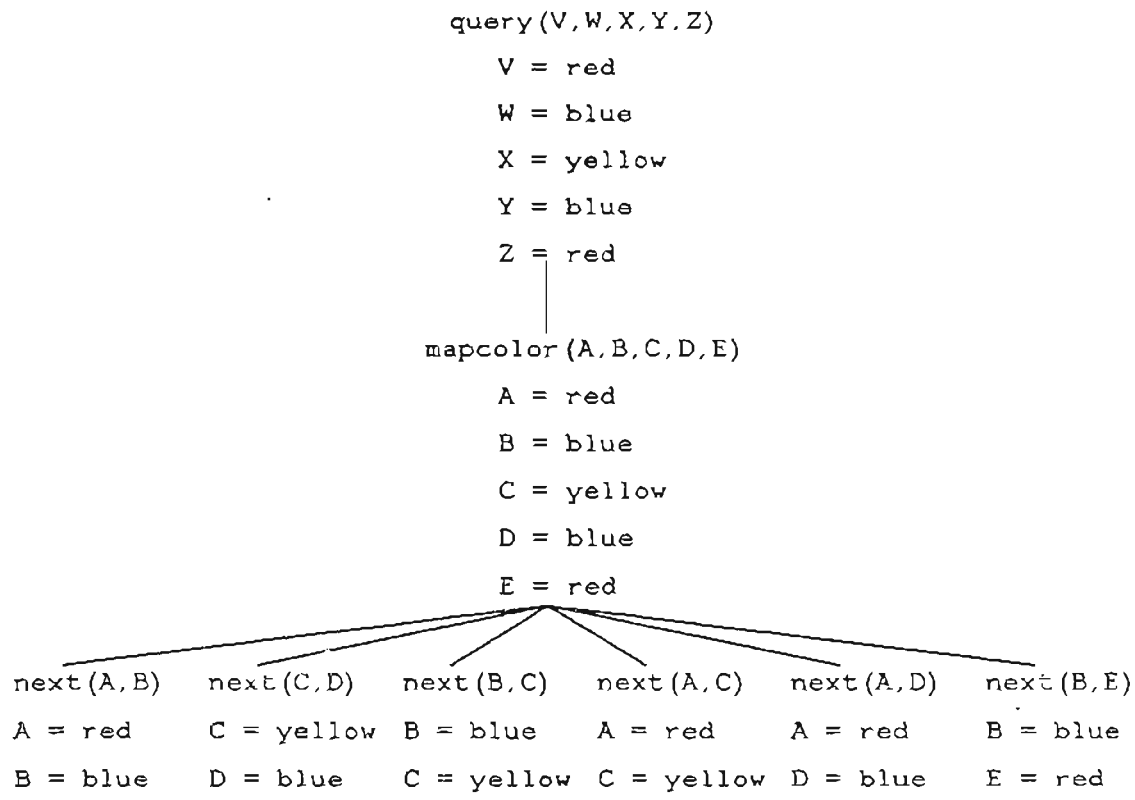
The Proof Tree
Figure 4.8

The next goal PAPI executes is `next(B, E)`. `Next(B, E)` matches the fact `next(blue, red)` and `E`, already on the *local* stack, is bound to `red` and put on the *trail* stack.



The Structures
Figure 4.9

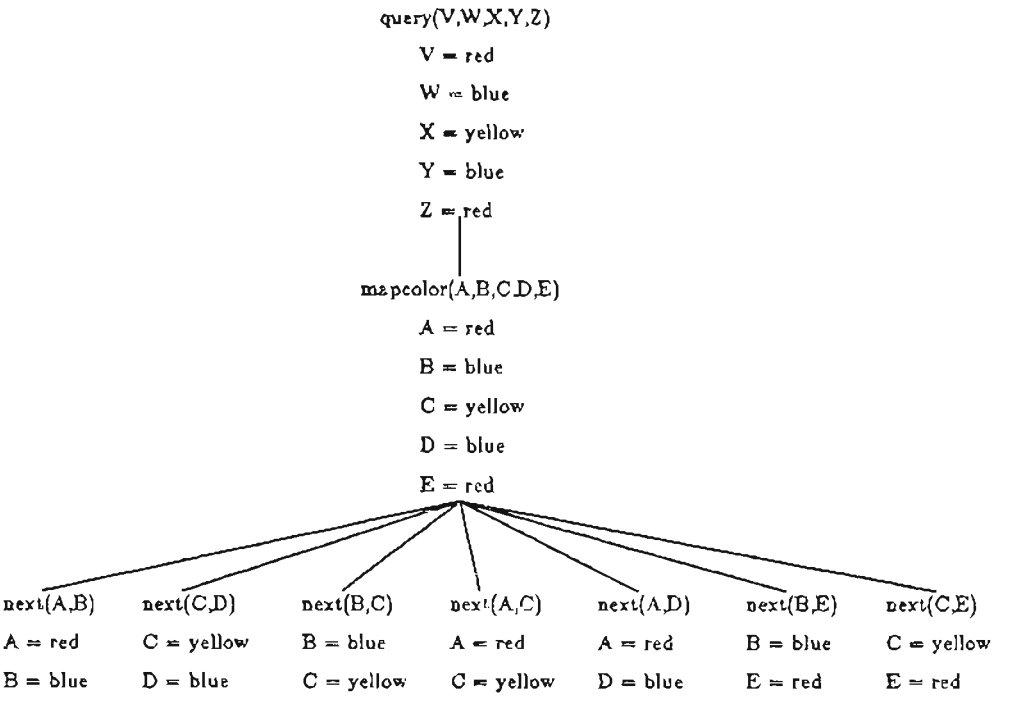
The goal `next(B, E)` is put in the proof tree as `mapcolor(A, B, C, D, E)`'s rightmost child.



The Proof Tree
Figure 4.10

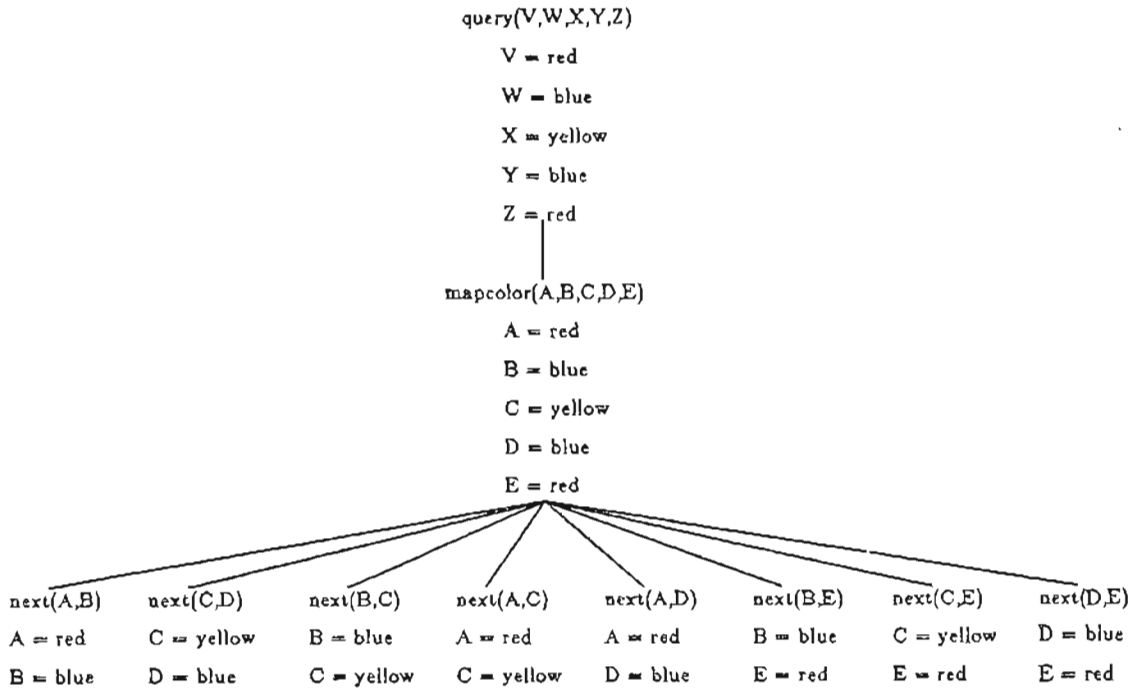
At this point, each of the query's arguments are bound and the *local* and *trail* stacks do not change as the remaining goals are executed. Thus, these stacks will not be illustrated in the final diagrams.

PAPI again moves up a level of the proof tree to `mapcolor(A, B, C, D, E)`, the parent of the `true` goal, and executes the next goal, `next(C, E)`. Goal `next(C, E)` matches the fact `next(yellow, red)` and is put in the proof tree.



The Proof Tree
Figure 4.11

Finally, the last goal in the clause, `next(D, E)`, is put in the proof tree and executed.



The Proof Tree
Figure 4.12

This last goal succeeds by matching the fact `next(blue, red)` and PAPI moves up a level of the proof tree to the goal `mapcolor(A, B, C, D, E)`. Since each of the goals in the clause have been executed, PAPI moves up another level of the proof tree to the query. At this point, all goals in the program have been executed successfully, PAPI claims a success, and returns the solution:

```
A = red
B = blue
C = yellow
D = blue
E = red
```

for the program `mapcolor`.

4.4. Multiple Process Memory Management

4.4.1. The Balance Series

In the Balance Series, a process's memory contains an area of shared memory and an area of private memory. The process's shared region of memory is accessible to all processes while the process's private region of memory is only accessible to the corresponding process. Thus, shared memory serves as a mechanism for communicating data among processes. This form of communication, however, requires a means of synchronizing the processes that alter the shared data in order to prevent collisions among these processes. The simplest mechanism for synchronization available on the Balance Series is the *spinlock* type of semiphore.

The spinlock is a lock used to ensure that only one process has access to a shared variable or a shared data structure at a time. Before a process attempts to access a shared data object, the process must wait until the spinlock associated with the shared data object is unlocked. The process locks the spinlock, accesses the shared data object, and then unlocks the spinlock after completeing its task. If a process attempts to lock a locked spinlock, the process spins in a spin loop until the

lock is unlocked. Due to the hardware characteristics of the spinlock, it is impossible for more than one process to lock a spinlock at the same time. Throughout the remaining chapters, references to locks and locking of data structures implies the locking and unlocking of the spinlocks associated with the mentioned data structure.

4.4.2. Data Structures

The parallel execution model maintains the stack-based methodology observed in the sequential execution model by distributing *trail, goal, local,* and *global* stacks to the shared memory of each process. *Goalists* are also distributed to the shared memory region of each process. Although these data structures reside in the process's shared memory, they cannot be expanded or cleaned by other processes. Rather, each process pushes and pops its own stacks and allocates goal structures from its own *goalist* in the same manner described previously in the sequential model. A process may, however, examine the contents of another process's shared data and set reference pointers within the stacks to bind variables during unification. Restricted access to shared data and the RAP scheme's forced sequential execution of dependent goals (i.e., one process binds a variable and modifies the appropriate stacks at a time) prevents collisions within the above shared data structures and eliminates the locking requirement for these shared data structures.

Despite the distribution of *goalists* among processes, the proof tree remains centralized, yet consists of goal structures residing in various processes' memory. The process that allocates a goal structure from its *goalist* records the goal

structure's bindings on its *local, global,* and *trail* stacks and pushes a pointer to the goal structure onto its *goal* stack.

In addition to the data structures presented for sequential execution, each process maintains an *availist,* a *stolenlist,* and an *intlist* in its shared memory region. The *availist* and the *stolenlist* were created as a result of the distributed work scheme; *free* processes steal work from processes with extra work. The *availist* is a list of pointers to goal structures that are available for other processes to steal and the *stolenlist* is a list of pointers to goal structures that have been stolen by other processes. A pointer to a goal structure is never in a *stolenlist* and an *availist* at the same time, since a goal structure cannot be available for a process to steal and stolen. Moving the pointer to a goal structure from the *availist* to the *stolenlist* as the goal structure is stolen ensures that one goal structure is not stolen by more than one process.

A process's *availist* and *stolenlist* are frequently accessed and altered by other processes. As a result, each of these structures must be locked and unlocked as it is examined or altered by a process, including the process that owns the memory in which the list resides. In this case, locking these structures not only prevents collisions among processes, but it also ensures that 1) a goal structure is only on one list at each instant, and 2) a goal structure available for stealing is still on the *availist* by the time the process steals it (i.e., one process cannot steal an available goal structure as another process examines it on the *availist*).

Another shared data structure utilized in parallel execution is the *intlist,* a list of pointers to *interrupt* structures. An interrupt structure is created and initialized

by one process and passed to another during execution. Interrupt structures are required in PAPI since processes are not permitted to alter the stacks or goal structures owned by other processes. A process sends an interrupt structure to another process when 1) a backtracking process reaches a goal structure that is owned by another process, 2) a backtracking process iinvalidates another process's goal structure by undoing its bindings made by the backtracking process's goal structure, or 3) a process must terminate. An interrupt structure is allocated dynamically from the sending process's shared memory and a pointer to this interrupt structure is put on the receiving process's *intlist* by the sending process. The *intlist* structure, like the *availist* and *stolenlist* structures, is frequently altered and hence, must be locked.

Interrupts are required in PAPI since processes are not permitted to alter the goal structures or stacks owned by other processes. Thus, an interrupt is sent from one process to another to inform the receiving process of the goal structure where backward execution is to resume and what type of backward execution the first process was engaged in when it encountered the second process's goal structure. In addition, interrupts are sent to processes owning no longer valid goal structures that are to be removed from the proof tree and stacks.

The *availist*, *stolenlist*, and *intlist* are arrays of static memory, as in the case of the stacks and goal structures. A process maintains two indices, also in shared memory, to both its *availist* and *stolenlist*. One index points to the top-most occupied position in the array and the other index points to the bottom-most occupied position in the array. These indices prevent a process from searching the entire *availist* or *stolenlist*; instead, a process searches from the top index of either list to

the bottom index. A process also maintains an index for its *intlist* that points to the most recent interrupt added to the *intlist*. After a process completes the work required by an interrupt, the interrupt is finished and the process removes the finished interrupt structure from its *intlist* by decrementing the index value. If the finished interrupt is not the most recent interrupt in the *intlist*, then the newer interrupts are moved backward into the space occupied by the finished interrupt and the index is decremented. As goals structures are removed from the *availist* or *stolenlist*, the indices and remaining goal structures are shifted to reflect this deletion. Thus, a form of garbage collection is performed on the *availist*, *stolenlist*, and *intlist*. This type of maintanence is relatively inexpensive since these structures are typically small.

4.5. Forward Parallel Execution

Parallel execution begins with forking the number of processes specified by the user upon invoking PAPI. The parent process, proc0, is initialized to a *normal* state while the child processes, proc1, proc2, proc3 ..., are in a *new* state. A *normal* process executes the Prolog program, while *new* processes are put to sleep with the system call `sigpause()`. The *new* processes remain asleep until proc0 creates work and awakens them with the `kill` (SIGALRM) system call.

After the fork, the *normal* process, proc0, begins sequential execution of the query and continues until parallelism is specified, by `par` or the parallel evaluation of `ipar` or `gpar`. At this point, a goal structure is created for each of the parallel goals in the scope of the execution graph expression and put in the proof tree as

siblings. The first goal structure, oldest and leftmost sibling, is put on proc0's *goal* stack while the other goal structures are put on its *availist*. Before continuing forward execution with the new goal on its *goal* stack, proc0 records the number of goal structures created for parallelism, *no_of_forks*, and stores this value in the goal structure of the new goals' parent. The *no_of_forks* field of a goal structure is shared (since goal structures are shared) and must be locked by processes that access or alter its value. Each time a parallel goal is completed, *no_of_forks* is locked and decremented. When the value of *no_of_forks* is decremented to zero, the first goal beyond the scope of the execution graph expression is executable. For example, if a clause contains the goals:

```
par(a(A), b(B), c(C)), d(D) ...
```

then *no_of_forks* is set to 3 in `a(A)`'s parent's goal structure for the goals `a(A)`, `b(B)`, and `c(C)` in the scope of the `par` execution graph expression. After each of the parallel goals, `a(A)`, `b(B)`, and `c(C)`, are finished and *no_of_forks* is decremented to 0, `d(D)` becomes eligible for execution. If proc0 finishes its current goal before *no_of_forks* reaches zero, proc0 removes a goal structure from its own *availist*, puts it on its *goal* stack, and executes it. (Since proc0 took a goal structure from its own *availist*, the goal structure is removed from the *availist*, but it is not put on the *stolenlist*. The procedure of a process taking a goal structure from its own *availist* is called `getwork()`.)

After proc0 puts the parallel goals on its *availist* and sets *no_of_forks*, it awakens the *new* processes and puts them in a *free* state. *Free* processes search for

goals to steal from other processes' *availists* by following the *steal rule*. The *steal rule* maintains the stack-based characteristic of the stacks by specifying the conditions for goals that a process may steal. Basically, this rule requires that only goals newer than the last goal on the process's *goal* stack may be stolen, and if the *goal* stack is empty, any goal may be stolen. The *steal rule* is stated as follows:

The *steal rule*:

1) The first child is never stolen.

2) If the current proof tree were traversed inorder and each goal were assigned a number corresponding to its position in the proof tree, then any goal with a position number greater than that of the last goal executed (the top goal on the process's *goal* stack) may be stolen by that process.

3) If the process's *goal* stack is empty, then its last executed goal position number is zero.

Once a *free* process finds an acceptable goal to steal, the *free* process:

1) makes an exact copy of the acceptable goal structure that it is stealing

2) marks the original goal structure *stolen*

3) puts its copy of the goal structure in the proof tree as the *only* child of the *stolen* goal and then pushes a pointer to its copy of the *stolen* goal structure onto its *goal* stack

4) moves the pointer to the *stolen* goal structure from the victim process's *availist* to the victim process's *stolenlist*

5) changes its state from *free* to *normal* and begins forward execution of the goal on its *goal* stack

In some programs, there may be many acceptable goals for a *free* process to steal. It is important in these situations that a process steals a goal that will not prevent it from stealing other goals. For example, if a process steals the rightmost available goal in the proof tree, the *steal rule* prohibits the process from taking any other available goals that are to the left of it (since traversing the proof tree inorder reveals that the remaining available goals would have lower position numbers than the stolen goal). Thus, this process remains *free*, or idle, for the rest of the program's execution and the benefits of parallelism are greatly reduced. Backward execution may free an idle process by undoing its goals, hence making it available to steal acceptable goals again. Yet, if an inefficient scheme for choosing goals to steal is followed, the process will end up in the same idle situation again. Thus, a *stealing scheme* that directs the search throughout the proof tree for goals to steal, is necessary.

The stealing scheme implemented in PAPI (and hence, the stealing procedure `steal()`) begins the search for an acceptable goal to steal high in the proof tree at the ancestor of the last goal on the *free* process's *goal* stack. The motivation behind beginning the search for stealable goals high in the proof tree is to keep processes busy with large subtrees of goals. Since wasted time occurs when a process is idle and when it is looking for work, it is important to steal goals that will keep a process busy and not require it to steal often. By stealing goals high in the proof tree, it is more likely that the stolen goal will expand into a large subtree than if the goal were stolen low in the proof tree. Therefore, the process will spend more time computing than being idle. In addition to stealing high in the proof tree, the process

steals the closest or leftmost available goal in the desired level of the proof tree. Stealing the leftmost available goal permitted by the *steal rule* enables the *free* process to also steal that goal's forward available siblings. Although it is possible that a process may spend more time searching the proof tree for eligible goals under this scheme, there is also more of a chance that a goal stolen high in the proof tree will yield more work to the *free* process. Shallow proof trees will not suffer or benefit from this scheme as searching for work will not be as expensive, but it will occur more often.

Another means for a process to get more work is to `getwork()`. The `getwork()` procedure is executed when a process finishes executing its current goal and still has available goals on its own *availist*. Since `getwork()`, similar to a process stealing from itself, only involves one process, it is cheaper to execute than `steal()`. Thus, a process always tries to `getwork()` before it tries to `steal()`. As a process executes `getwork()`, it locks its *availist* and transfers the next goal to its *goal* stack. After transferring the goal, the process begins forward execution of the new goal. The new goal is not put on the process's *stolenlist* and copies are not made of the new goal since the new goal remained on the same process that created it.

Once a solution to the query is found, the process that completed the last goal and returned to the query declares a success. The other processes are notified that a solution has been found via an *interrupt* (discussed in Chapter 5). The solution to the query is printed to the screen and if another solution is requested, the succeeding process sends another interrupt to the same processes telling them to continue

execution. Otherwise, if another answer is not requested or the query fails (no more answers exist), the process sends an interrupt the other processes telling them to terminate and then terminates itself.

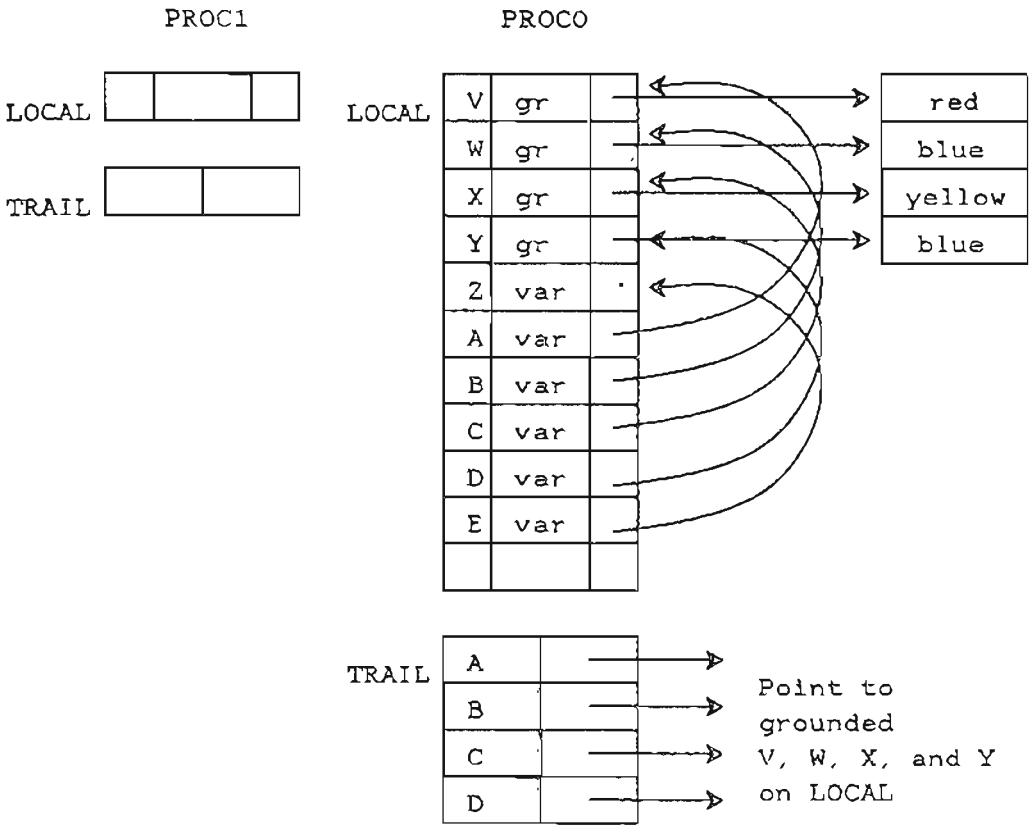
4.6. Forward Parallel Execution Example

Forward parallel execution with 2 processes for the program `mapcolor` is described in this section. The parallel version of the program `mapcolor` is provided below.

```
mapcolor(A, B, C, D, E):- gpar([A, B, C, D],
                               next(A, B),
                               next(C, D)
                              ),
                          par(
                               next(B, C),
                               next(A, C),
                               next(A, D),
                               seq(
                                    next(B, E),
                                    par(
                                         next(C, E),
                                         next(D, E)
                                        )
                                   )
                              ).
```

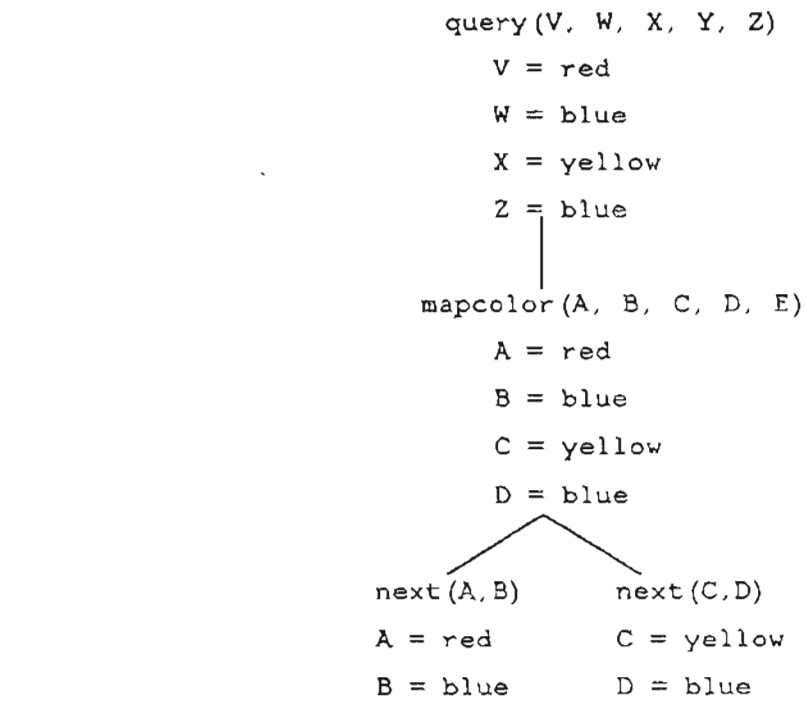
PAPI begins forward parallel execution by forking the child process, proc1, from the parent, proc0, and initialing the stacks and structures for both processes. Proc0 is in the *normal* state and begins forward execution while proc1, in the *new* state, sleeps.

Evaluation of the first execution graph expression, `gpar`, determines that sequential execution for the goals `next(A, B)` and `next(C, D)` is required, since the arguments `A`, `B`, `C`, and `D` are not *ground*. Thus, PAPI's forward parallel execution of `mapcolor` is the same as its forward sequential execution until the `par` execution graph expression is reached. After proc0 finishes the goal `next(C,D)`, proc1 is still sleeping and proc0's stacks are the same as those presented in the sequential execution example.



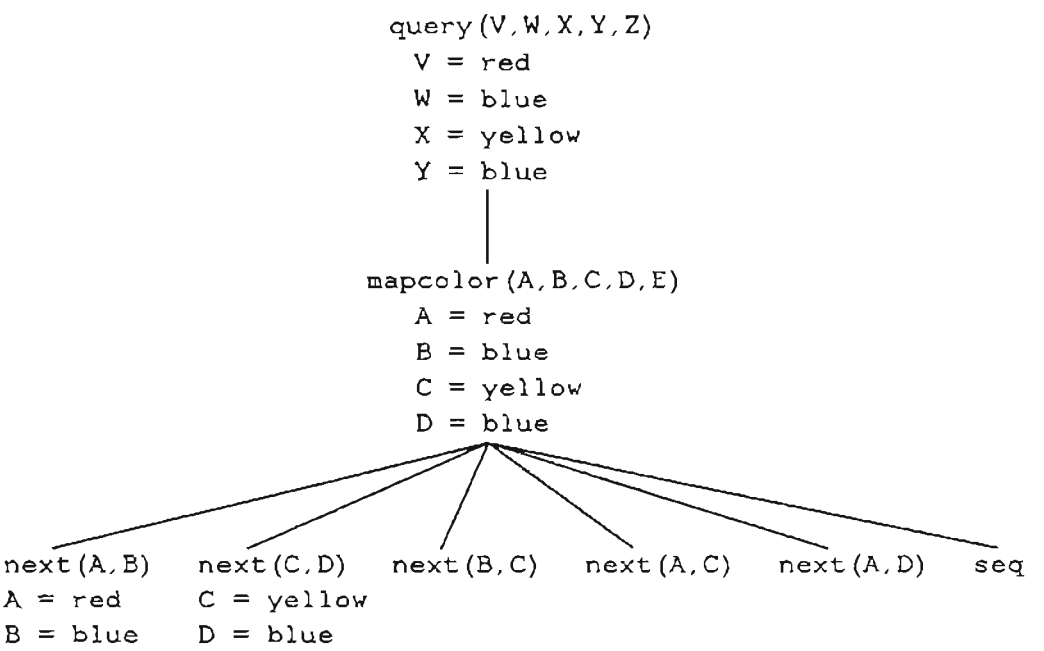
The Structures
Figure 4.13

The proof tree at this point of `mapcolor`'s execution is provided in Figure 4.14.

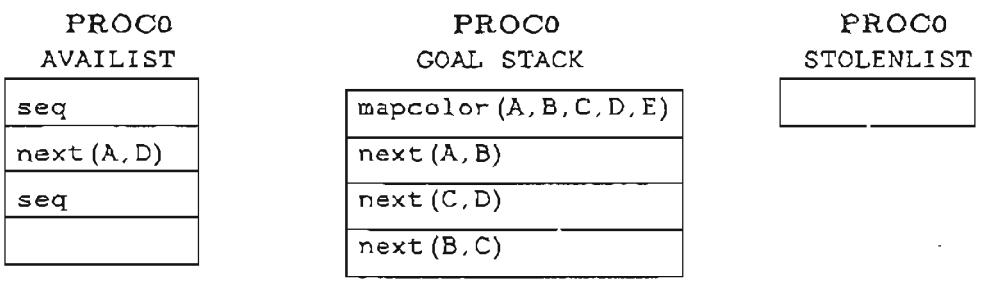


The Proof Tree
Figure 4.14

As proc0 executes the `par` execution graph expression, it allocates and initializes goal structures from its *goalist* for the goals within the scope of `par`. Proc0 puts the first goal structure, for the goal `next(B, C)`, on its *goal* stack and the remaining goal structures, for the goals `next(A, C)`, `next(A, D)`, and `seq`, on its *availist* for processes to take. These four goal structures are also put in the proof tree in the same order as if they were executed sequentially.



The Proof Tree



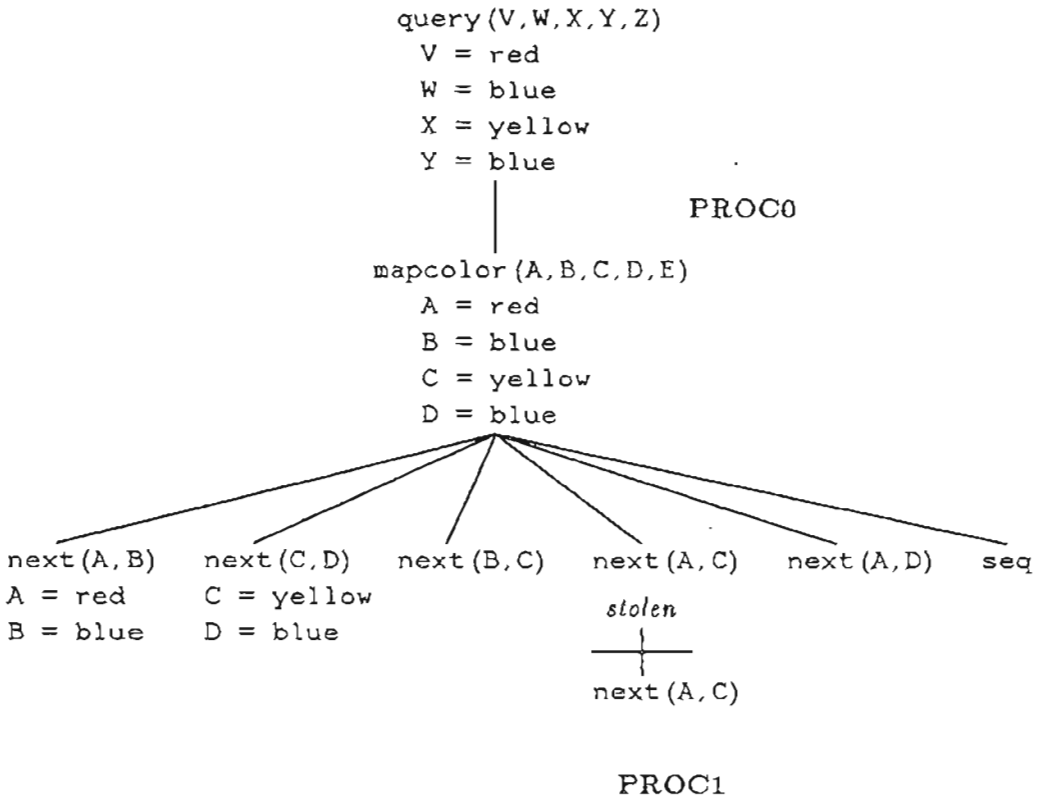
The Proof Tree and Proc0's Structures
Figure 4.15

Note in Figure 4.15 that a goal structure was created for the execution graph expression `seq`, but not for `par` or `gpar`. Since the `seq` execution graph expression is

within the scope of the `par`, `seq` must have a goal structure in the proof tree. The `par` execution graph expression, unlike `seq`, creates and positions goal structures in the proof tree *before* the goals are executed by PAPI, thus, `seq` holds a position in the proof tree for the goals in its scope and preserves the depth first ordering in the proof tree.

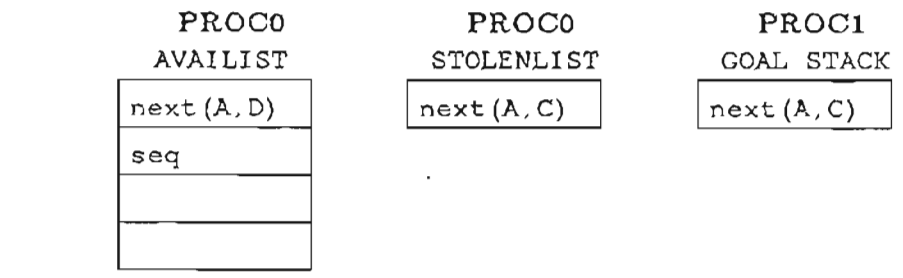
Proc0 also locks and sets the *no_of_forks* field in `mapcolor(A, B, C, D, E)`'s goal structure to 4, for the four goal structures created, and sends a `kill(SIGALRM)` system call to the sleeping proc1, to awaken it. Proc1 is put into the *free* state while proc0 continues forward execution with the oldest (leftmost) goal structure in the `par` scope, `next(B, C)`.

Proc1 is in the *free* state looking for work. Since its *goal* stack is empty, proc1 may steal any goal from proc0's *availist* (see the third *steal rule* in the previous section). Proc1 steals the leftmost available goal from proc0's *availist*, `next(A, C)`, by making a copy of the goal structure, marking the original goal structure *stolen*, and putting the new copy of `next(A, C)` in the proof tree.



The Proof Tree
Figure 4.16

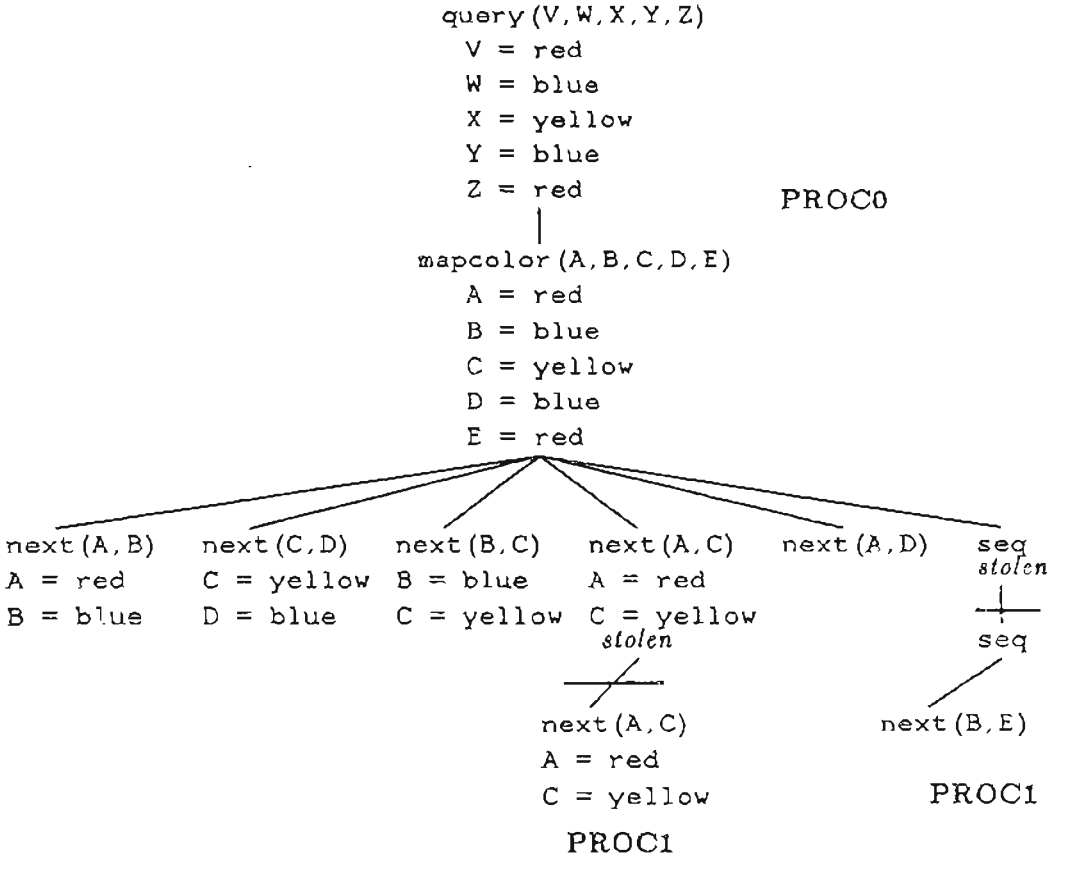
Proc0's *availist* and *stolenlist* are locked and modified to reflect this steal and the new goal structure is put on proc1's *goal* stack.



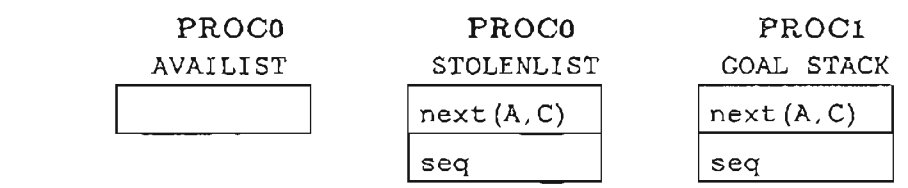
Proc0 and Proc1 Structures
Figure 4.17

Note that proc1's copy of `next(A, C)` contains the same pointers to the binding information maintained in the *stolen* goal, so proc1 is able to access this information during its forward execution.

Proc1 changes its state from *free* to *normal* and begins forward execution with the goal `next(A, C)` while proc0 finishes execution of the goal `next(B, C)`. When each process finishes its parallel goal or parallel goal's subtree, the executing process locks and decrements *no_of_forks* in the parent of the parallel goal and looks for more work. Thus, proc0 locks and decrements *no_of_forks*, in the goal structure `mapcolor(A, B, C, D, E)`, to 3 and executes `getwork()`. The goal `next(A, D)` is removed by proc0 from its *availist*, put on its *goal* stack, and executed while proc1 finishes the goal `next(A, C)`. Again, *no_of_forks* is decremented and proc1 steals the last goal on proc0's *availist*, `seq`.

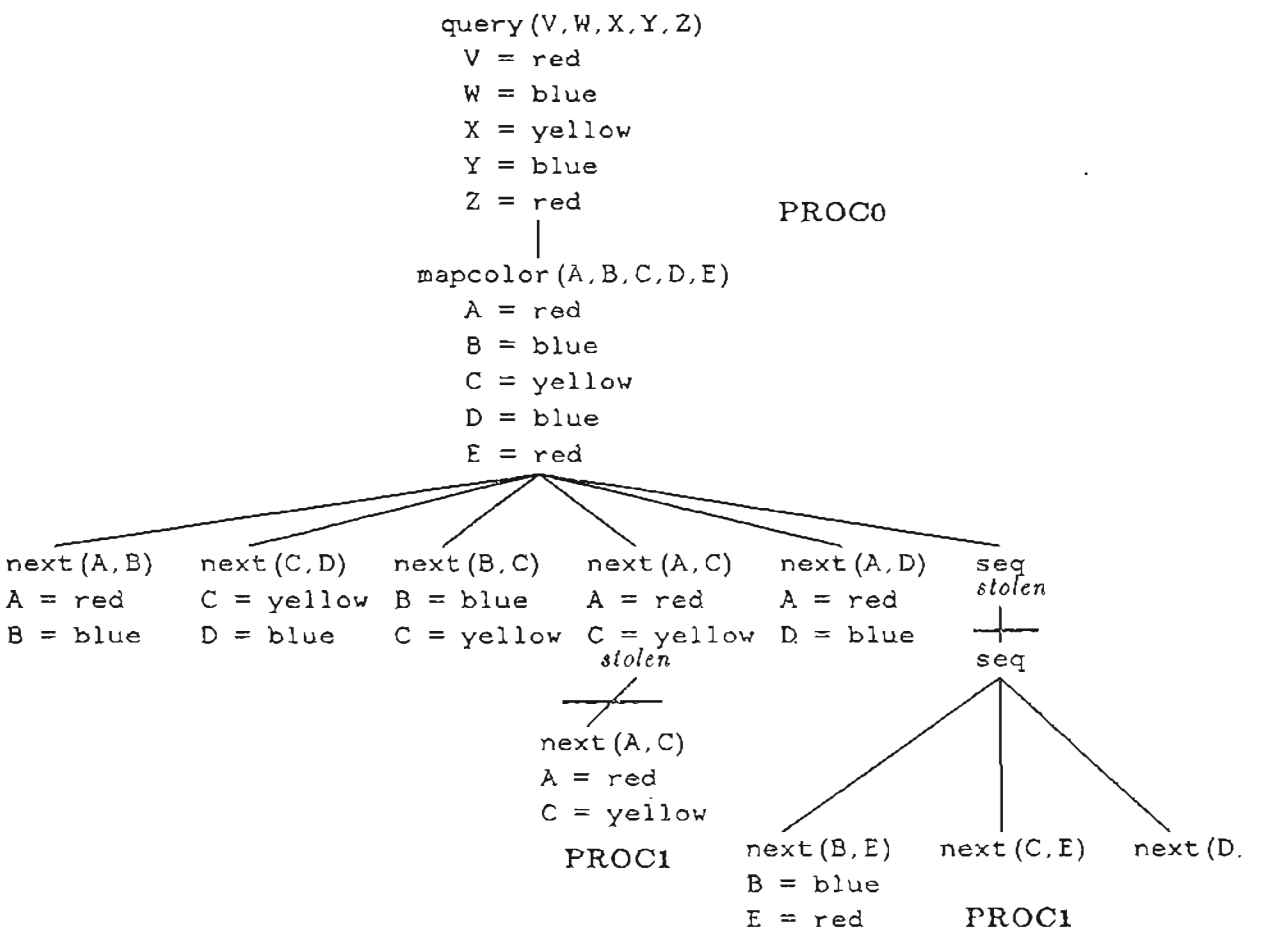


The Proof Tree
Figure 4.18



Proc0 and Proc1 Structures
Figure 4.19

Proc1 executes the no-operation `seq` and the goal `next(B, E)` as proc0 finishes its goal, `next(A, D)`. The `par` execution graph expression is executed by proc1 is the same manner that proc0 executed the previous `par`: goal structures are created for the parallel goals `next(C, E)` and `next(D, E)`, the goal structures are put in the proof tree and on the appropriate stacks, and *no_of_forks* in the `seq` goal structure is locked and set to 2.



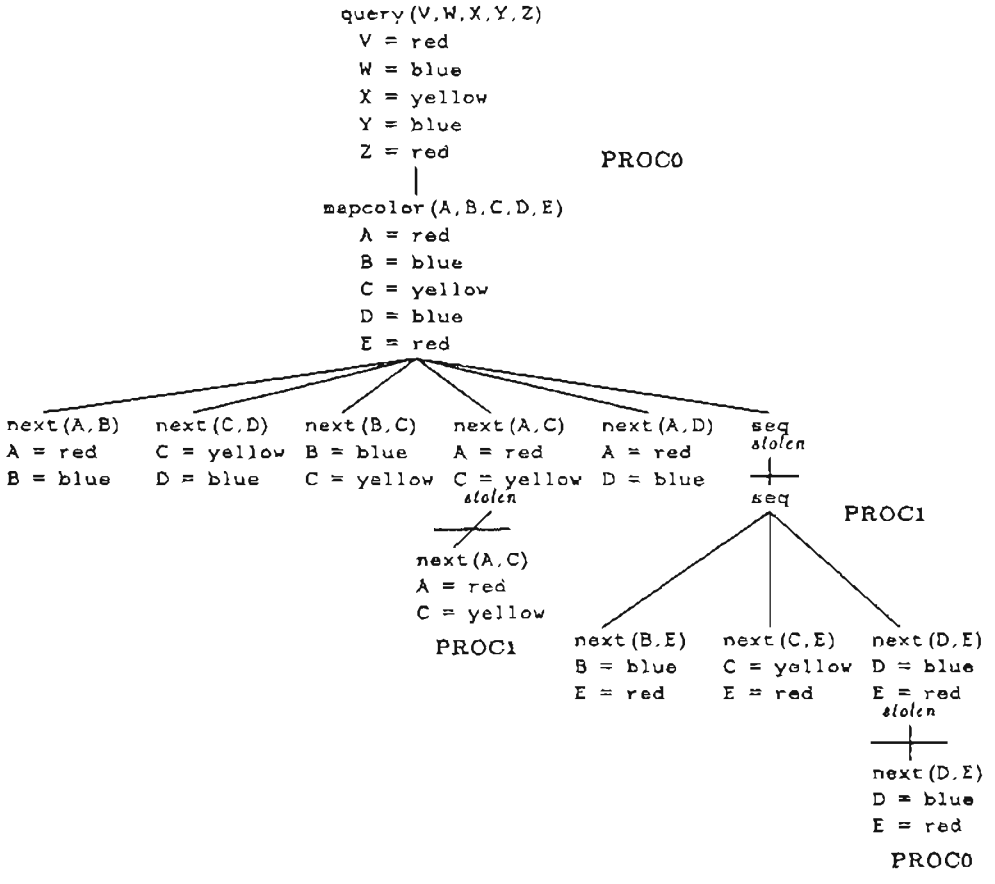
The Proof Tree
Figure 4.20

Proc1 Structures
Figure 4.21

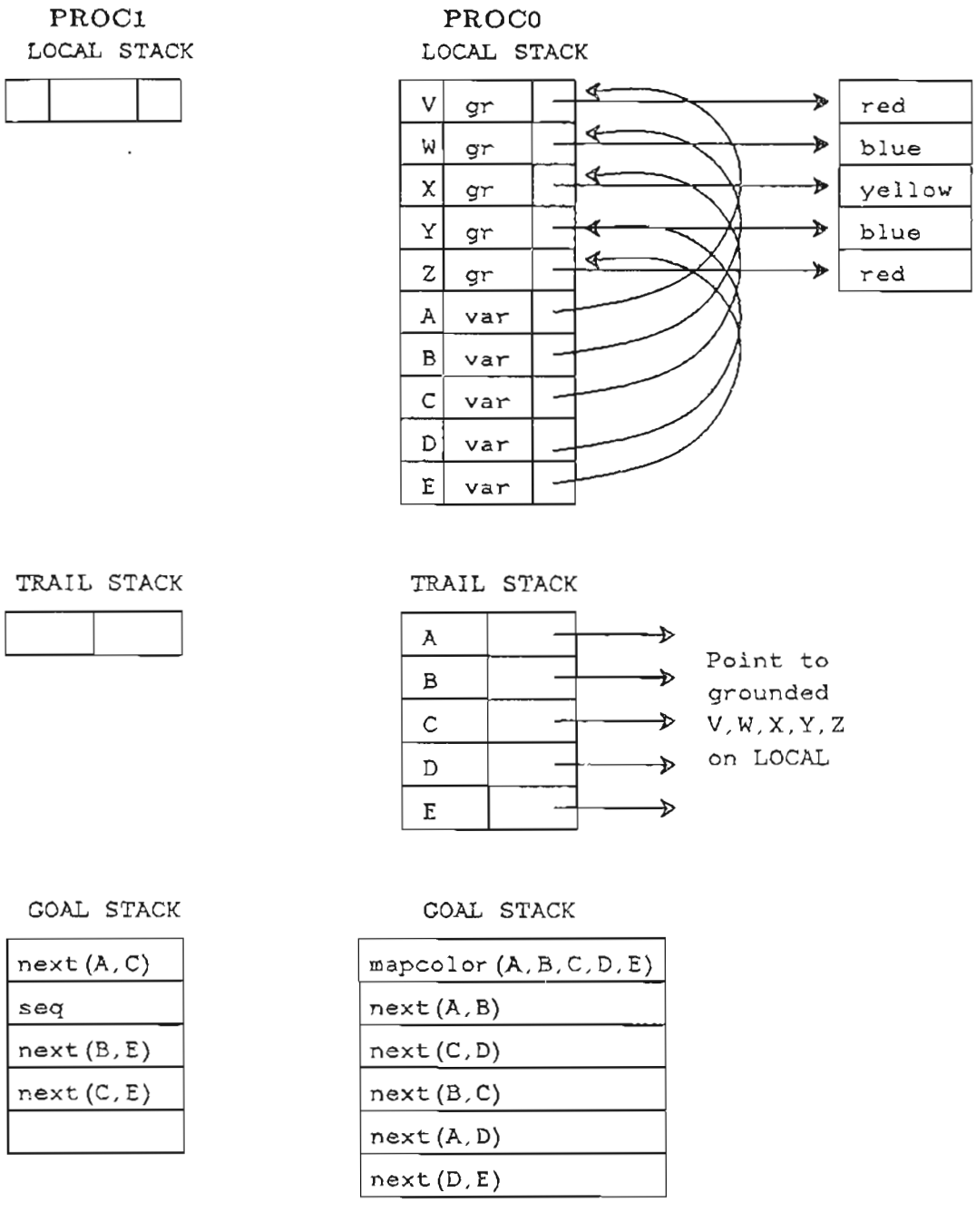
Proc1 executes the goal `next(C, E)` while proc0 steals the goal `next(D, E)` from proc1's *availist*. As proc0 and proc1 finish their goals, the processes lock and decrement *no_of_forks* in the goal structure `seq`. The last process to decrement *no_of_forks* sets its value to zero. This process, suppose it's proc0, continues execution by determining that the `seq` goal is finished and decrementing the *no_of_forks* in `mapcolor(A, B, C, D, E)`'s goal structure. Meanwhile, the other process, proc1, has finished all of its goals and is in the *free* state searching for available goals to steal.

Since *no_of_forks* in `mapcolor(A, B, C, D, E)` is also zero, proc0 examines the `mapcolor(A, B, C, D, E)` clause for the next goal after the scope of the `par`. There are no more goals so proc0 moves up a level in the proof tree to the parent of the `true` goal, `mapcolor(A, B, C, D, E)`. Each of the goals in the clause have been executed, so proc0 moves up another level in the proof tree to the

query. All goals in the program have been executed successfully and PAPI declares the solution presented in section 4.3.



The Final Proof Tree
Figure 4.22



The Final Structures
Figure 4.23

As an answer is presented to the user, the succeeding process "freezes" the other processes by issuing an interrupt. The user is then asked if more solutions are desired; if the answer is "no", then the processes are told to terminate via an interrupt, and the succeeding processes terminates, otherwise the other processes are "unfrozen" and backward execution begins.

CHAPTER 5

Backward Execution of PAPI

Backward execution, or backtracking, is the series of actions following a *failure*. A failure results when the current goal fails to match the current clause head or fact. Most sequential Prolog interpreters reduce a failure with *naive* backtracking; returning to the most recent *choicepoint* and restarting forward execution at that goal's next alternative[1]. Although naive backtracking is effective, it is very slow, and unnecessary work is often done. For example, if the goal:

```
f(1,2).
```

matches the clause:

```
f(X, Y) :- g(X), h(Y), i(X).
```

in a Prolog program where `g(X)` and `h(Y)` are choicepoints and the goal `i(X)` fails, naive backtracking resumes forward execution at the next alternative for `h(Y)`, the most recent choicepoint. Since `h(Y)` and `i(X)` are independent, it is useless to explore each of `h(Y)`'s alternatives in the attempt to satisfy `i(X)`. Instead, forward execution could restart at the next alternative for `g(X)` upon

[1] The most recent choicepoint is the most recent goal where alternative clauses are yet to be explored.

which `i(X)` depends. Thus, an efficient method for eliminating useless work in naive backtracking would reduce time spent in backward execution.

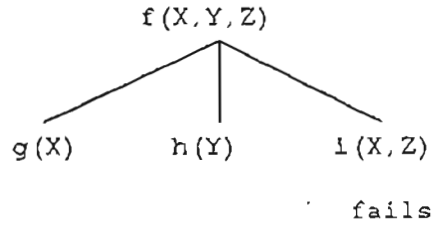
5.1. Improving Naive Backtracking

Intelligent backtracking schemes to remedy execution of useless work in naive backtracking have been studied by Cox, Pietrzykowski, Matwin [CoP81] and Brunooghe, Pereira, and Porto [BrP81] [PeP81]. These authors observe that naive backtracking always considers the whole proof tree as its *failure tree* when a goal fails. In their intelligent backtracking scheme, the authors attempt to reduce the inefficient search of naive backtracking by pruning the proof tree to a minimal failing deduction[2] subtree. A minimum failing deduction subtree is a subtree (of the proof tree) that can be determined to have caused the failure of the goal. In addition, unification is not possible in a minimum failing deduction subtree. For example, if a Prolog program containing the clause and goals:

```
f(X, Y, Z):- g(X), h(Y), i(X, Z).
g(2).
g(3).
i(3, 4).
```

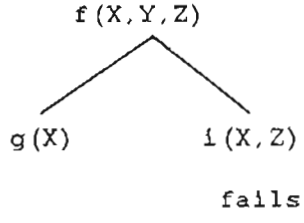
fails at the goal `i(X, Z)`, the deduction tree for the clause at the failure is:

[2] A deduction tree is a proof tree without substitutions for variables.



Naive Backtracking Failure Tree
Figure 5.1

Naive backtracking would consider the entire proof tree as the failure tree and backtrack throughout the proof tree for the most recent choicepoint. Intelligent backtracking, however, would prune the naive backtracking failure tree to the minimal failure subtree below:



Minimal Failure Deduction Subtree
Figure 5.2

and reduce the amount of useless work done in backward execution. The problem

with this intelligent backtracking scheme, however, is that it requires expensive runtime bookkeeping.

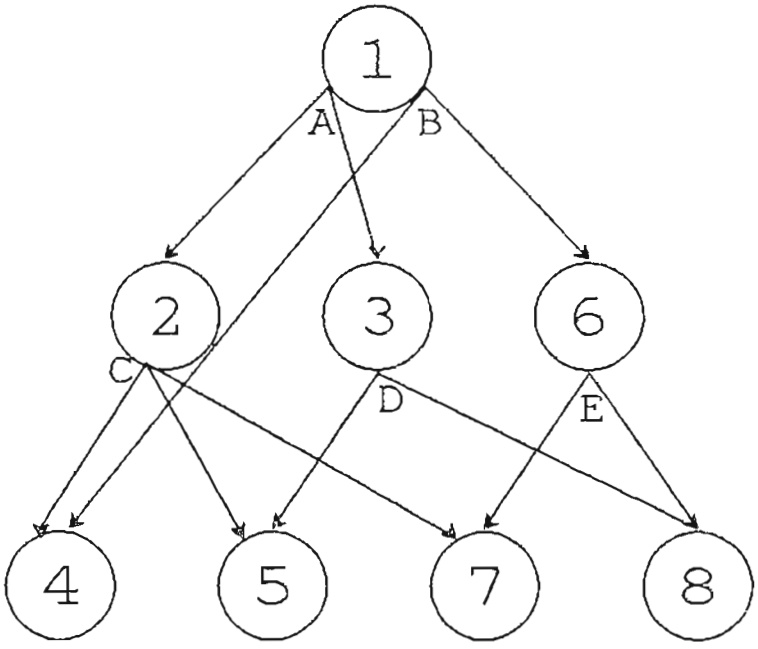
Cox, Pietrzykowski, and Matwin claim that the overhead associated with this bookkeeping is unacceptable for the sequential execution of most logic programs. Chang and Despain [ChD85] show that for deterministic programs, such as `fibonacci`, there is no backtracking, so the intelligent backtracking scheme only introduces overhead during execution. For other programs, such as `mapcolor`, a good ordering of the goals in the body of the clause by the programmer reduces the applicability of intelligent backtracking and again the overhead of intelligent backtracking outweighs its benefits. Thus, this intelligent backtracking scheme introduces high overhead without sufficient improvement in backtracking performance.

Another scheme, developed by Chang and Despain [ChD85], is *semi-intelligent* backtracking. Semi-intelligent backtracking is an improvement over naive backtracking, yet it requires very little runtime overhead. This method examines *data dependency graphs* [CDD85] created for each clause to determine variable dependencies among the variables in the clause. An analysis of the dependencies determines the backtracking path taken when a goal in the clause fails. These backtracking paths permit a semi-intelligent form of backtracking without high runtime overhead.

An example data dependency graph for the `color(A, B, C, D, E)` clause in the program `color` is illustrated in Figure 5.3. This `color` program is Chang and Despain's version of a Prolog program that solves a map-coloring problem for a map containing five regions and four colors.

```
color(A, B, C, D, E):- next(A, B),
                       next(A, C),
                       next(A, D),
                       next(B, C),
                       next(C, D),
                       next(B, E),
                       next(C, E),
                       next(D, E).
```

Chang and Despain's data dependency graph for this clause is depicted below. The circles in the graph represent a goal in the body of the `color(A, B, C, D, E)` clause and the number inside of the circles is the goal's position in the clause body, e.g., 1 is `next(A, B)`. The arcs illustrate the dependencies of the variables between the goals. Further analysis of this graph yields the semi-intelligent backtracking paths discussed by the authors.



Data Dependency Graph for `color` Clause
Figure 5.3

One limitation of Chang and Despain's approach is that, unlike intelligent backtracking, semi-intelligent backtracking behavior cannot cross clause boundaries since data dependency graphs only represent variables of one clause. Yet, the authors conclude that there are many programs for which this form of backtracking suffices. In addition, when compared to other more intelligent backtracking schemes, the semi-intelligent method proved more favorable, despite its limitations, due to its low runtime overhead. Furthermore, this scheme is called only when it is advantageous to do so, not for say, deterministic programs.

The advantages of the semi-intelligent backtracking method outweigh its limitations, and hence this scheme was chosen for improving backward execution in PAPI. Although PAPI does not implement data dependency graphs, the underlying ideas behind semi-intelligent backtracking have been modified for use in PAPI.

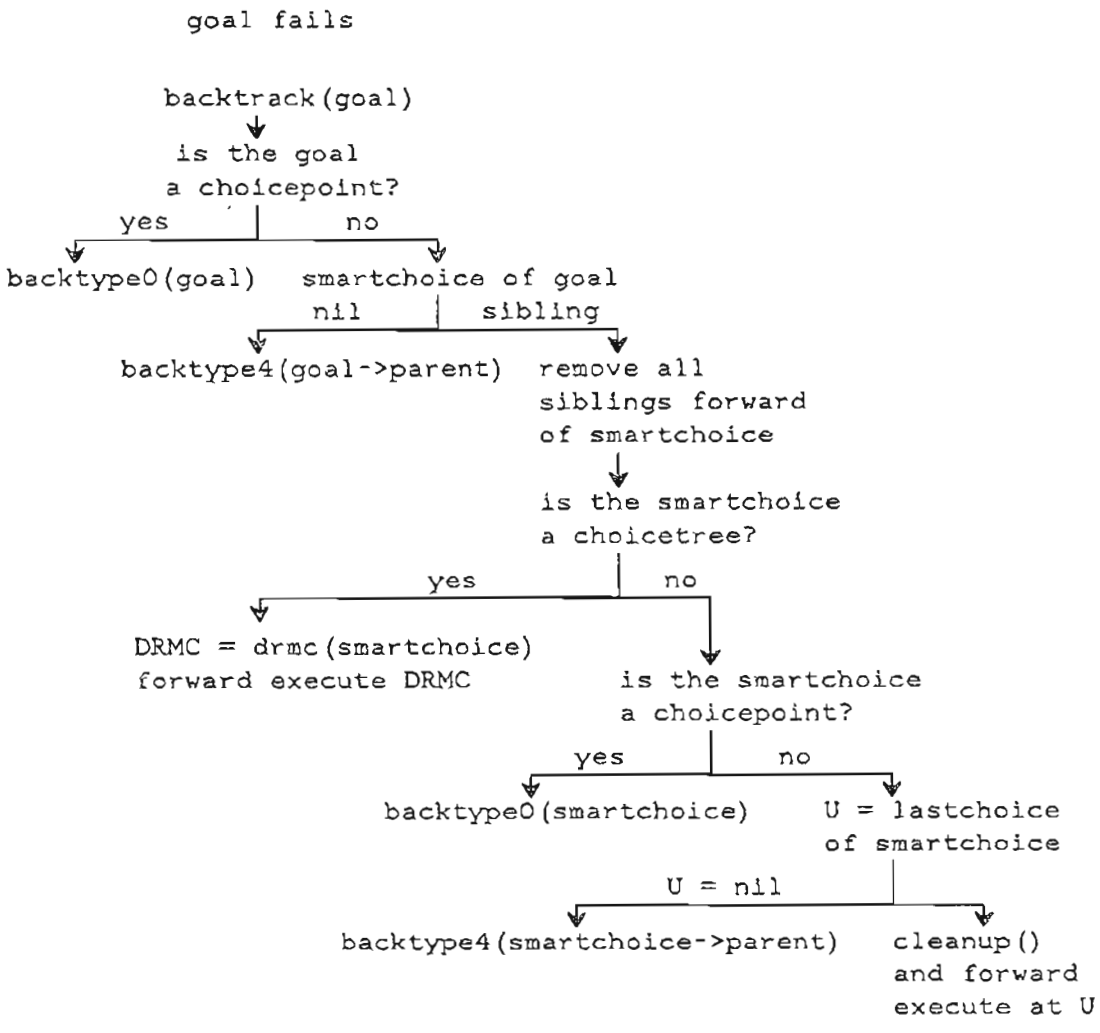
5.2. Backward Sequential Execution

Backward sequential execution begins when forward sequential execution encounters a failure. Upon reaching the failure, PAPI executes `backtrack()` which determines the backtracking path required for the failing goal. Once the backtracking path for backward execution is established, the relevant backtracking code is executed, the proof tree and goal structures are restored to the selected choicepoint's environment, and forward execution resumes in the new environment created by backward execution.

In order to analyze the failed goal and its environment, `backtrack()` examines several fields in the goal structure. The first field, *flabel*, is a pointer to the next alternate clause to try if the goal is a choicepoint when it fails, and the goal structure's *choicept* field records 1 if the goal is a choicepoint and 0 if it is not. The goal's *choicetree* field is 1 if there is a choicepoint in the goal structure's subtree and 0 otherwise. The *smartchoice* pointer points to the goal's closest back sibling on which it depends. If the goal is not dependent on a back sibling (it may not have a back sibling), then its *smartchoice* is *nil*. Furthermore, the goal's *lastchoiceptr* is a pointer to the most recent choicepoint *before* this goal structure in the proof tree. These fields in the goal structures are set and modified throughout forward and

backward execution.

With this information, `backtrack()` analyzes the failing goal and continues with the appropriate `backtype()` routine. These `backtype()` routines, similar to those in Chang and Despain's scheme, are backtracking paths directing the search for the choicepoint at which forward execution will resume. Throughout the search, the `backtype()` routines restore the structures and proof tree to the environment of a goal in the backtracking path and eventually to that of the choicepoint. Forward execution then resumes with the next alternative of the choicepoint, until another failure is encountered. Figure 5.4 summarizes the `backtrack()` routine and is followed by a short description of the backtracking routines.



Backtrack() Flow Chart
Figure 5.4

`cleanup()`:

The `cleanup()` routine removes specified goals from the proof tree and pops the *local, global, trail,* and *goal* stacks of the arguments belonging to those goals. This routine is responsible for resetting the environment to that of the choicepoint in order for forward execution to resume with the next alternative of the choicepoint.

`drmc()`:

Drmc() returns the *deepest rightmost choicepoint* in the subtree of the specified goal. The right child and sibling pointers in the goal structures are followed down the subtree of the specified goal and the choicepoint and *choicetree* fields of the goal structures determine which goal is the drmc. Once the drmc is found, `drmc()` resets the environment such that forward execution may resume at the drmc's next alternative.

`backtype0()`: (shallow)

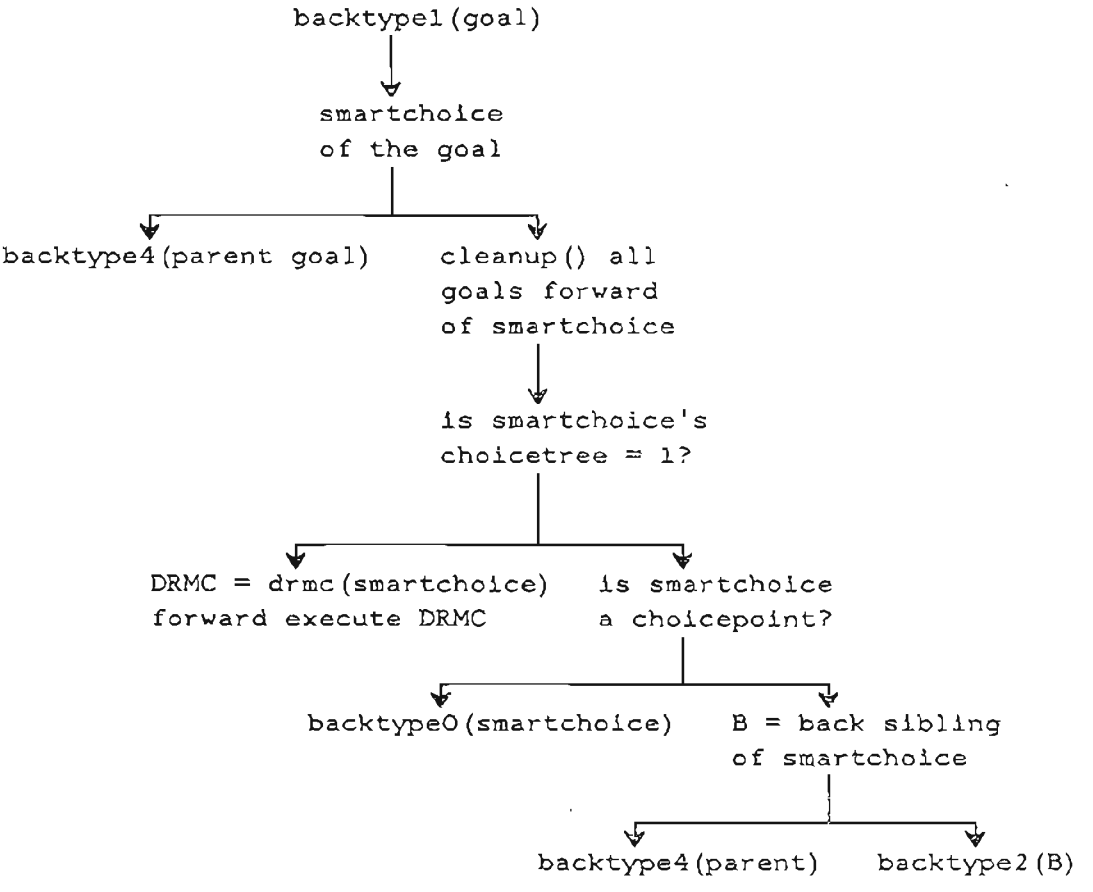
Type 0 backtracking is often referred to as *shallow* backtracking (vs. *deep* backtracking), since the failing goal is a choicepoint and is the goal at which forward execution will resume. The proof tree also remains the same. The *local*, *global*, and *goal* stacks are not popped, but the *trail* is unwound to its position before the failing match was attempted. Forward sequential execution resumes with the next alternate clause of the current goal stored in the goal's *flabel*.

`backtype1()`: (smartchoice to sibling)

Type 1 backtracking[3] begins backward execution by examining the failing goal's *smartchoice*. The *smartchoice* may be *nil*, in which case backward execution continues with the failing goal's parent in `backtype4()`. Otherwise, `cleanup()` removes all goal structures forward of the *smartchoice* from the proof tree and *goal* stack, and pops the bindings made by the removed goals from the *local*, *global*, and *trail* stacks. At this point, the *smartchoice*'s *choicetree* field is checked. If the *choicetree* is 1, then the drmc in the *smartchoice*'s subtree is returned by `drmc()` and `cleanup()` again cleans the environment to that of the choicepoint. Forward execution resumes after the choicepoint is sent to `backtype0()`. If the *smartchoice*'s *choicetree* is 0, the *smartchoice*'s *choicept* field is checked. In the case that the *smartchoice* is a choicepoint, forward sequential execution resumes with the alternate clause specified by the *smartchoice*'s *flabel*. If the *smartchoice* is not a choicepoint and has a back sibling, the back sibling is sent to `backtype2()`, otherwise the *smartchoice*'s parent is sent to `backtype4()`. A flow chart for this routine is in Figure 5.5. Note that only one *smartchoice* move can be made within a set of siblings; other backtracking moves between siblings must be naive.

---

[3] `Backtype1()` is never directly called by `backtrack()` during backward sequential execution, but a variation of `backtype1()` is incorporated in `backtrack()` and this routine is used in backward parallel execution. For this reason I chose to include `backtype1()` in these descriptions.



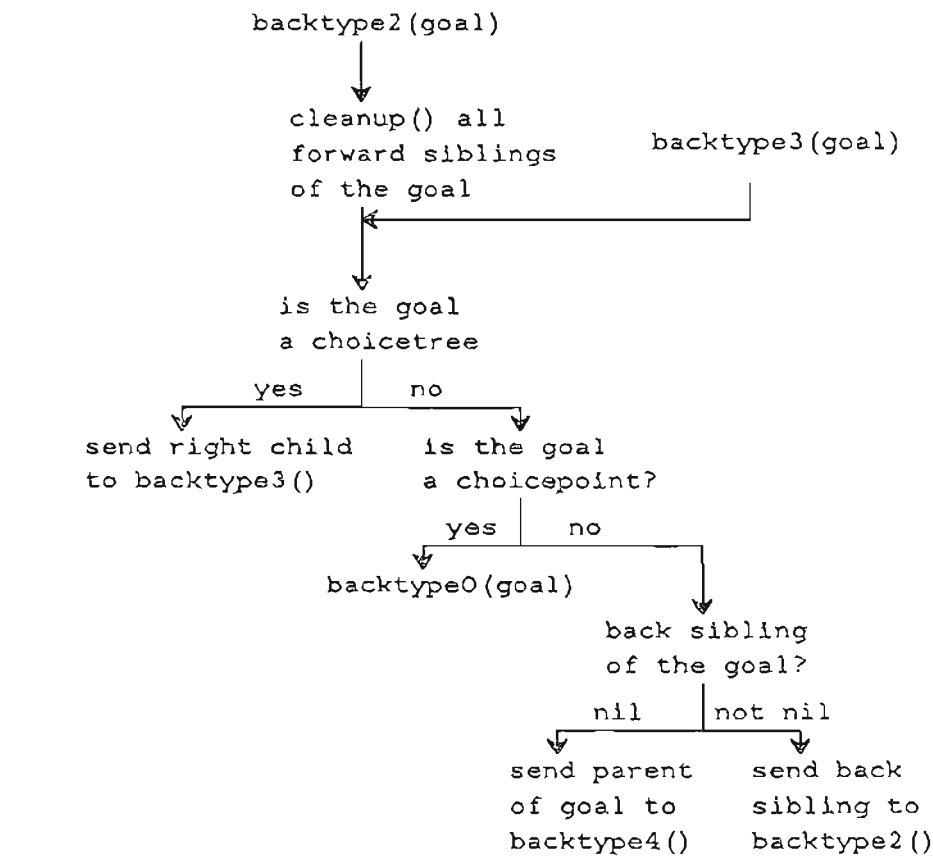
Backtype1 () Flow Chart
Figure 5.5

`backtype2()`: (naive to sibling)

Type 2 and type 3 backtracking, in sequential execution, search for the drmc in the subtree of the argument goal. The goal passed to `backtype2()` may be a choicepoint and not have any children, in which case `cleanup()` is executed and removes any forward goals of the argument goal. Forward execution then resumes with the *flabel* of the choicepoint. Otherwise the goal's *choicetree* field is checked and if it is 0, the goal's back sibling is passed to `backtype2()`, but if it is 1, the goal's rightmost child is sent to `backtype3()`. In the event that the back sibling is *nil*, the parent of the goal is passed to `backtype4()`. Figure 5.6 illustrates this routine.

backtype3 () : (backtrack to children)

Type 3 backtracking continues the search for the drmc. If the goal passed to backtype3 () is a choicepoint and is childless, then the structures and the proof tree are cleaned by cleanup () and forward execution continues as in the previous cases, but if the goal is not a choicepoint or has children, the *choicetree* is checked and the same action is taken as in the backtype2 () case. See Figure 5.6 below.

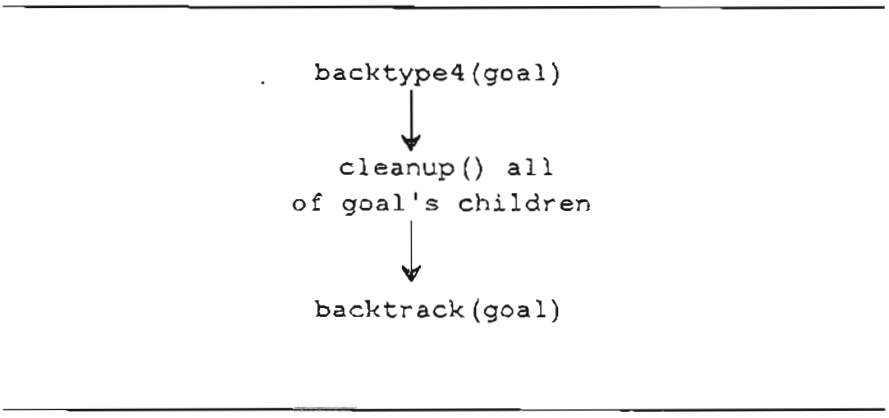


Backtype2 () and Backtype3 () Flow Chart
Figure 5.6

backtype4 () : (backtrack to parent)

Type 4 backtracking begins by removing all of the goal's (the goal passed to

backtype4()) children from the proof tree and removing their arguments and bindings from the stacks. Backtype4() continues by sending the passed goal to backtrack() where it is sifted through the flow chart again.



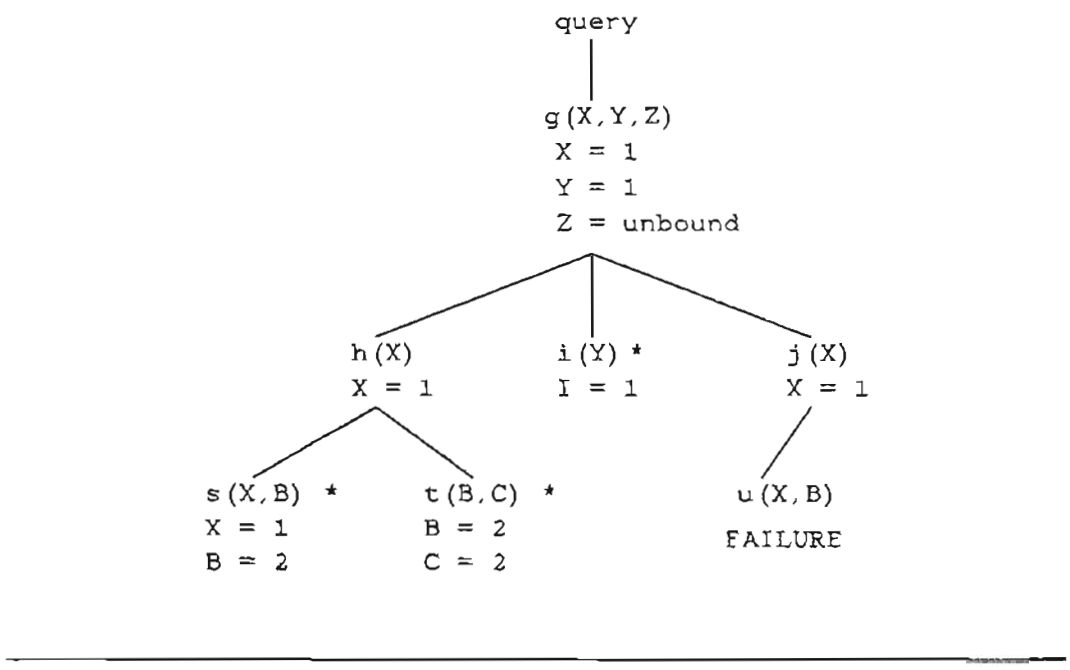
Backtype4() Flow Chart
Figure 5.7

5.3. Backward Sequential Execution Example

An example of backward sequential execution for the program G:

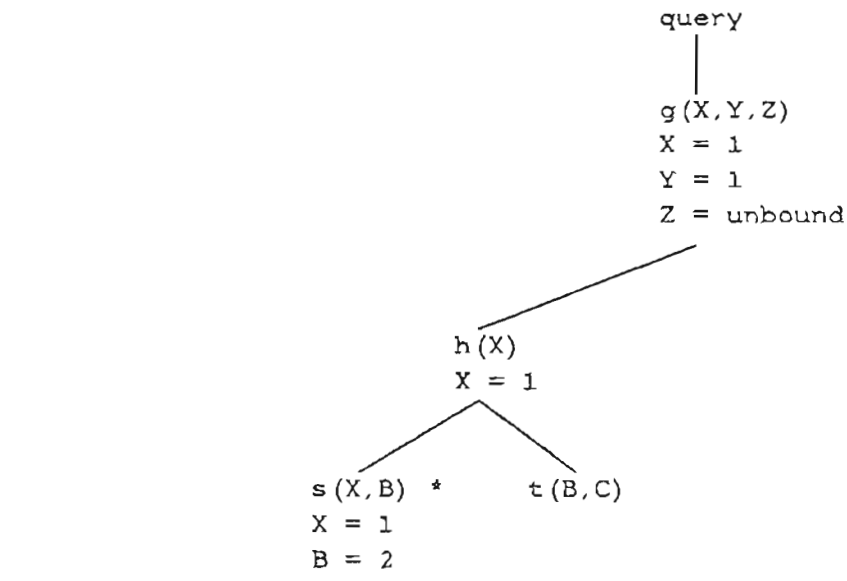
```
query :- g(X, Y, Z).

g(X, Y, Z):- h(X), i(Y), j(X), k(Z).
h(A)  :- s(A, B), t(B, C).
i(1).
i(2).
i(3).
j(A)  :- u(A, B), v(B).
k(A)  :- l(B), m(C), n(A, B, C).
s(1, 2).
s(2, 3).
t(2, 2).
t(3, 3).
u(2, 2).
v(2).
l(2).
m(1).
m(2).
n(1, 2, 2).
```

begins at the first failure in forward sequential execution with the proof tree and bindings illustrated below. Note that goals followed by a "*" are choicepoints.



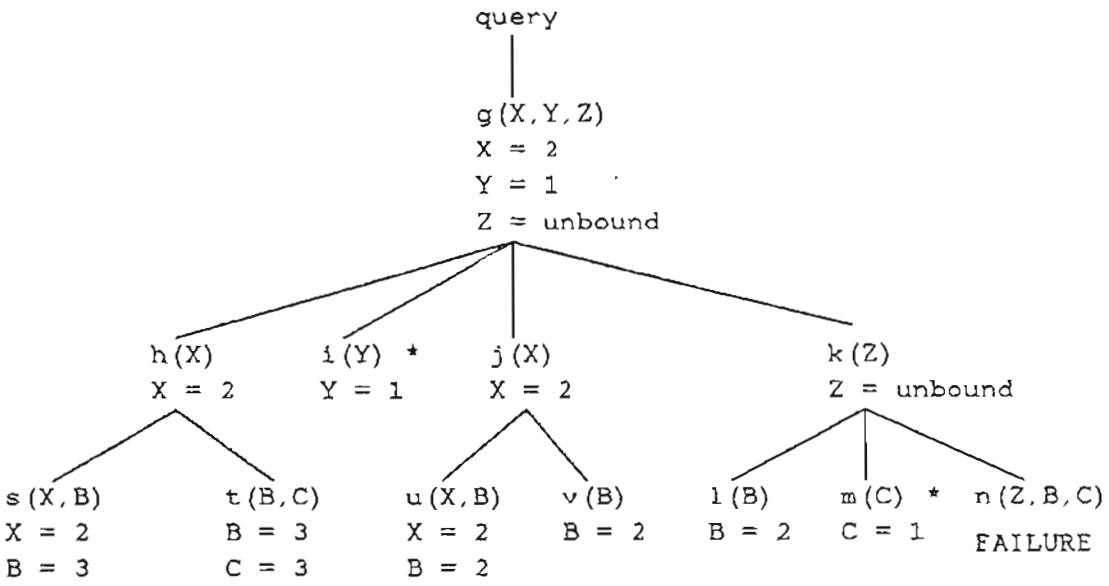
Proof Tree Before Backward Execution
Figure 5.8

Backward execution begins with the failing goal, `u(X, B)`, as its current goal in `backtrack()`. Since `u(X, B)` is not a choicepoint and its *smartchoice* is *nil* (`u(X, B)` does not depend on a back sibling), its parent, `j(X)` is sent to `backtype4()`. `Backtype4()` removes `j(X)`'s children from the proof tree, pops the stacks, and passes `j(X)` to `backtrack()`. The `backtrack()` routine examines `j(X)`'s *smartchoice*, `h(X)`, and removes the goals `i(Y)` and `j(X)` and their children from the proof tree and stacks. The goal, `h(X)`, is a choicetree so `drmc()` returns the drmc in `h(X)`'s subtree, `t(B, C)`. Forward execution resumes with the goal `t(B, C)` and its *flabel*, the fact `t(3, 3)`, and the proof tree:



Proof Tree After Backward Execution
Figure 5.9

The goal `t(B, C)` fails to match the fact `t(3, 3)` and is passed to `backtrack()`. As `t(B, C)` is no longer a choicepoint, its *smartchoice*, `s(X, B)`, is failed and the goal `t(B, C)` is removed from the proof tree and stacks. Goal `s(X, B)` becomes the current goal, the stacks are reset, and forward execution continues at `s(X, B)`'s *flabel*, `s(2, 3)`, until another failure is encountered at the goal `n(Z, B, C)`.

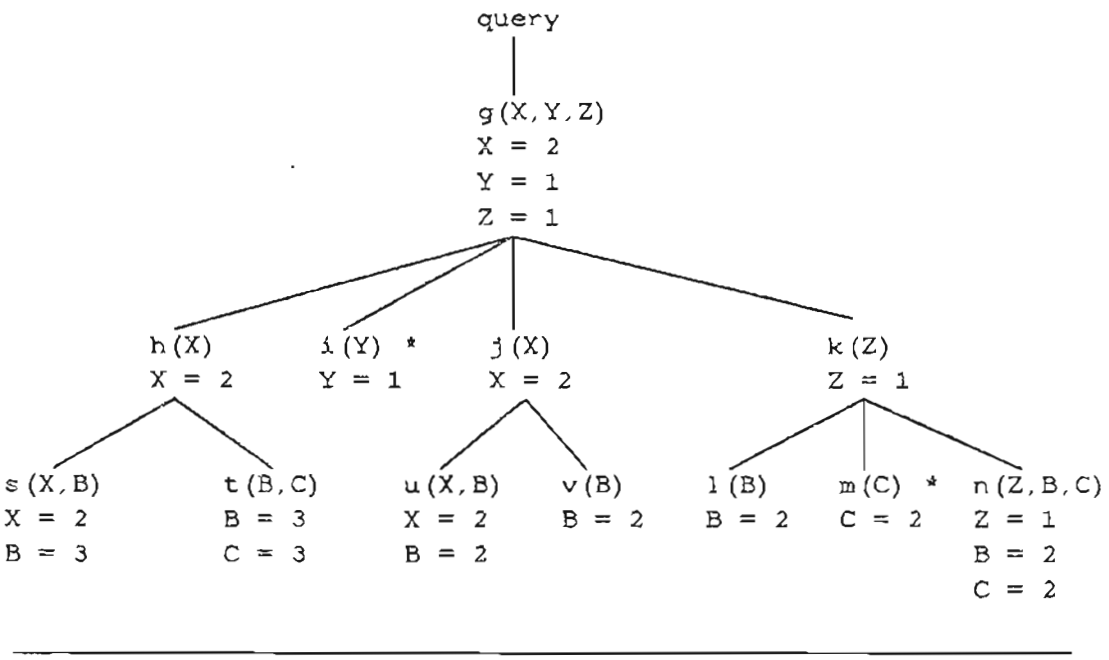


Proof Tree at Second Failure
Figure 5.10

Since the failing goal is not a choicepoint, `backtrack()` removes `n(Z, B, C)` from the proof tree and the structures, and then examines `n(Z, B, C)`'s *smartchoice*, `m(C)`. This goal is a choicepoint, the stacks are reset, `m(C)` becomes the current goal, and forward execution resumes at `m(C)`'s *flabel*, `m(2)`, and continues until the solution:

```
X = 2
Y = 1
Z = 1
```

is declared for the program. The proof tree for this solution to `G` is:



Proof Tree after Success
Figure 5.11

Although the semi-intelligent backtracking scheme described earlier does not eliminate all unnecessary work in backward execution, it does provide advantages over the naive backtracking scheme implemented in most Prolog interpreters. One example is its "skipping" over the goal `i(Y)` when backtracking `u(X, B)`. Since the goal `u(X, B)` did not depend on `i(Y)`, PAPI was relieved from using each of `i(Y)`'s alternatives in useless work. This scheme, however, would not save work in this particular program if the next alternative for the goal `t(B, C)` were the fact `t(2, 3)`. In this case, the local variable binding would change but the global variable binding for `X` would remain the same (`X = 1`) and the goal `u(X, B)` would fail

again since it requires X to be bound to 2 for a success. Therefore, the semi-intelligent backward execution would require the same amount of work as naive backward execution. Yet, when the semi-intelligent scheme does eliminate unnecessary work, it is a significant improvement over naive backtracking. The benefits of this scheme are also evident in the parallel version of backward execution.

5.4. Backward Parallel Execution

Backward parallel execution involves backward execution over one or more processes [Bor86]. Since each process is responsible for its own goal structures and stacks, backward execution often crosses process boundaries when it searches the proof tree for a choicepoint. For example, backward execution initiated on one process may require stacks and goal structures owned by other processes to be cleaned and removed from the proof tree, or perhaps, another process to continue backward execution until a choicepoint is found or that process reaches another process's boundary. That is, a process may not execute a `backtype()` routine with another process's goal. Instead, the process issues an *interrupt* to the processes owning the foreign goal and that goal's process continues backward execution in the specified routine. For this reason, a means of communication enabling a process to tell another process which routines to execute is required.

In addition, a *free* process needs to determine which processes are backtracking and which are not when searching for a goal to steal. A backtracking process is in the *backtracking* state and a process that has finished backward execution but is waiting for the other processes to finish backtracking is in the *waiting* state. No

process is allowed to steal goals from the *availist* of a *backtracking* or *waiting* process. When a choicepoint is found and before forward execution resumes, all *waiting* processes are put back into a *normal* or *free* state by the routine, `resetcpus()`. (At this point, all processes involved in this particular failure are in a *waiting* state). The *backtracking* and *waiting* states prevent goals that are no longer valid from being stolen and executed by other processes.

Communication among processes is accomplished via the interrupt system. This mechanism does not interrupt processes at any point. Instead, processes check for interrupt structures throughout forward execution and while looking for goals to steal. The interrupt system consists of interrupt structures that are put on other processes' *intlists* and each process locks and checks its own *intlist* for interrupt structures at specific points during forward execution. The interrupt structure contains several fields: *receiver*, the goal that the interrupt effects; *sender*, the sending process's current goal; *type*, the type of interrupt structure; *nosent*, keeps track of the number of interrupts the sending process issued with the same *sender* goal (used for special *types* only); and *choicefound*, set to 1 if the *receiver*'s process finds a choicepoint while servicing the interrupt.

An interrupt is issued when the sending process allocates shared memory for the interrupt structure, initializes the fields in the interrupt structure, and puts a pointer to the interrupt structure on the *intlist* of the receiving process, the process owning the *receiver* goal. Throughout forward execution, each process locks and checks its *intlist* for an interrupt structure. If a process encounters an interrupt structure on its *intlist*, the process stops its execution and services the interrupt by

executing the routine associated with the *type* of interrupt. After finishing the interrupt, the process executes `retinterrupt()`, which locks and removes the interrupt structure from the process's *intlist*, and determines if the process will continue forward execution in a *normal* state where it left off, or look for a goal to steal in a *free* state. The process's continuation state depends on the *type* field of the interrupt serviced and whether or not the process's current goal was removed as a result of the interrupt. Processes completing `idie` interrupts do not not execute `retinterrupt()`.

The interrupt *type* corresponds to the routine that the sending process requires the receiving process to execute. For example, the `iwait` interrupt corresponds to the `wait()` routine. The receiving process executes the specified routine with the *receiver* goal as its current goal. After issuing an interrupt, the sending process continues backward execution or waits for the receiving process to finish servicing the interrupt, depending on the *type* of interrupt it issued. The routines and interrupt types are summarized below. Note that these backward execution routines differ from those implemented in backward sequential execution.

`drmc()`

The `drmc()` routine is a combination of the `backtype2()` and the `backtype3()` routines that searches a goal's subtree for the deepest rightmost choicepoint. If the search crosses a process boundry as it moves deeper into the proof tree, an `ibacktype3` interrupt is issued to the process that will continue the search, and if the process crosses a process boundry as it moves across the proof tree, an `ibacktype2()` interrupt is issued. The sending process is then put into a *waiting* state.

`wait()`:

The `iwait` interrupt is issued to all other processes by the process that finds a solution to the query. It is issued after the solution is printed and suspends the receiving processes until the user specifies whether or not another solution to the query is desired. If no more solutions are requested, the sending process issues an `idie` interrupt to all processes and terminates itself. Otherwise, the receiving processes go to `retinterrupt()`, where the interrupt is removed from each process's *intlist* and the process continues execution where it was when it received the `iwait` interrupt.

`die()`:

The `idie` interrupt is issued to all processes if the user does not want any more solutions to the query. The receiving process terminates as a result of the `idie` interrupt.

`cancelwait()`:

The `icancelwait` interrupt is unique in that it is the only interrupt that a goal issues to itself only. It serves an administrative role for process by recording the number of `icancel` interrupts sent by the process. An `icancelwait` interrupt is always issued before the `icancel` interrupts. As each of the `icancel` interrupts are issued by the sender, the *nosent* field of the `icancelwait` interrupt structure is locked and incremented, and as each `icancel` is completed by the receiving process, the *nosent* field of the `icancelwait` interrupt structure is locked and decremented. When the `icancelwait`'s *nosent* field returns to 0, the `icancelwait` interrupt is removed from the process's *intlist* and the process may continue backward execution.

The `cancelwait` interrupt ensures that a backtracking process does not continue backtracking until the environment is reset to the appropriate state, as dictated by the backtracking routines. That is, when a backtracking process is responsible for resetting the environment to a specific state, that process may not continue backtracking until the state is completely restored and all assisting processes finish.

`cancel()`:

The `icancel` interrupt is issued when the goal to be "cancelled" is owned by another process. The process receiving the `icancel` interrupt cancels a goal, which is specified as the *receiver* in the `icancel` interrupt structure, by removing the goal and its descendents from the proof tree, the process's *local, global,* and *trail* stacks, the process's *goal* stack, and the process's *stolenlist* or *availist*. Each of the *receiver* goal's children are cancelled, which in turn cancel their children. If any of the *receiver* goal's children were stolen by another process, the

process executing `cancel()` issues a `icancelwait` to itself and then `icancel` interrupts to the processes owning the stolen children. After each of the processes complete their `icancel` interrupt, the processes lock and decrement the *nosent* field in the *cancelwait* interrupt associated with their `icancel` interrupt. When *nosent* reaches zero, the process that sent the `icancel` interrupts continues backward execution.

`backtrack()`:

The `ibacktrack` interrupt is issued from a process to another process when it is necessary for the receiving process to `backtrack()` with the *receiver* as its argument. The sequential `backtrack()` routine has been expanded for parallel backtracking across boundaries. When the process executing `backtrack()` reaches a process boundry, it issues an interrupt. The `icancel` interrupt is to cancel goals, the sending process must wait for all `icancel` interrupts to complete before continuing its execution in `backtrack()`. Otherwise, the process continues backward execution, or if it has finished, it is put in the *waiting* state until forward execution restarts. A flow chart summarizing the `backtrack()` routine is shown in Figure 5.12. Note that in the diagram, `calltype1()`, `calltype2()`, `calltype3()`, `calltype4()`, and `calldrmc()` are routines that determine if the executing process continues with the routines `backtype1()`, `backtype2()`, `backtype3()`, `backtype4()`, and `drmc()` respectively, or if the executing process must issue an interrupt to another process to execute the corresponding routine.

`backtype0()`:

Type 0 backtracking is often referred to as *shallow* backtracking (vs. *deep* backtracking), since the failing goal is a choicepoint and remains the current goal. The proof tree also remains the same. The *local*, *global*, and *goal* stacks are not popped, but the *trail* is unwound to its position before the failing match was attempted. Forward sequential execution resumes with the next alternate clause of the current goal stored in the goal's *flabel*.

`backtype1()`:

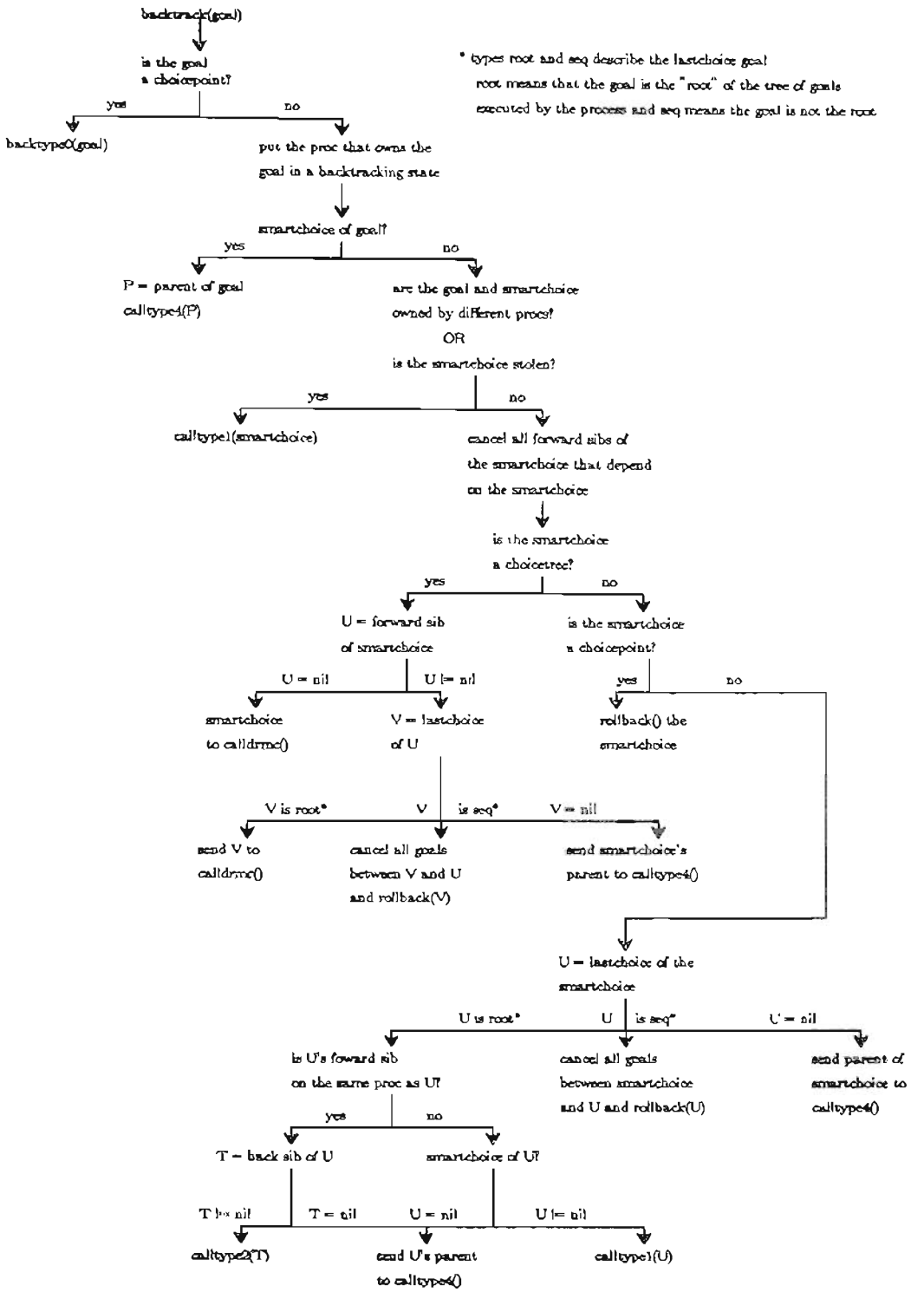
`backtype2()`:

`backtype3()`:

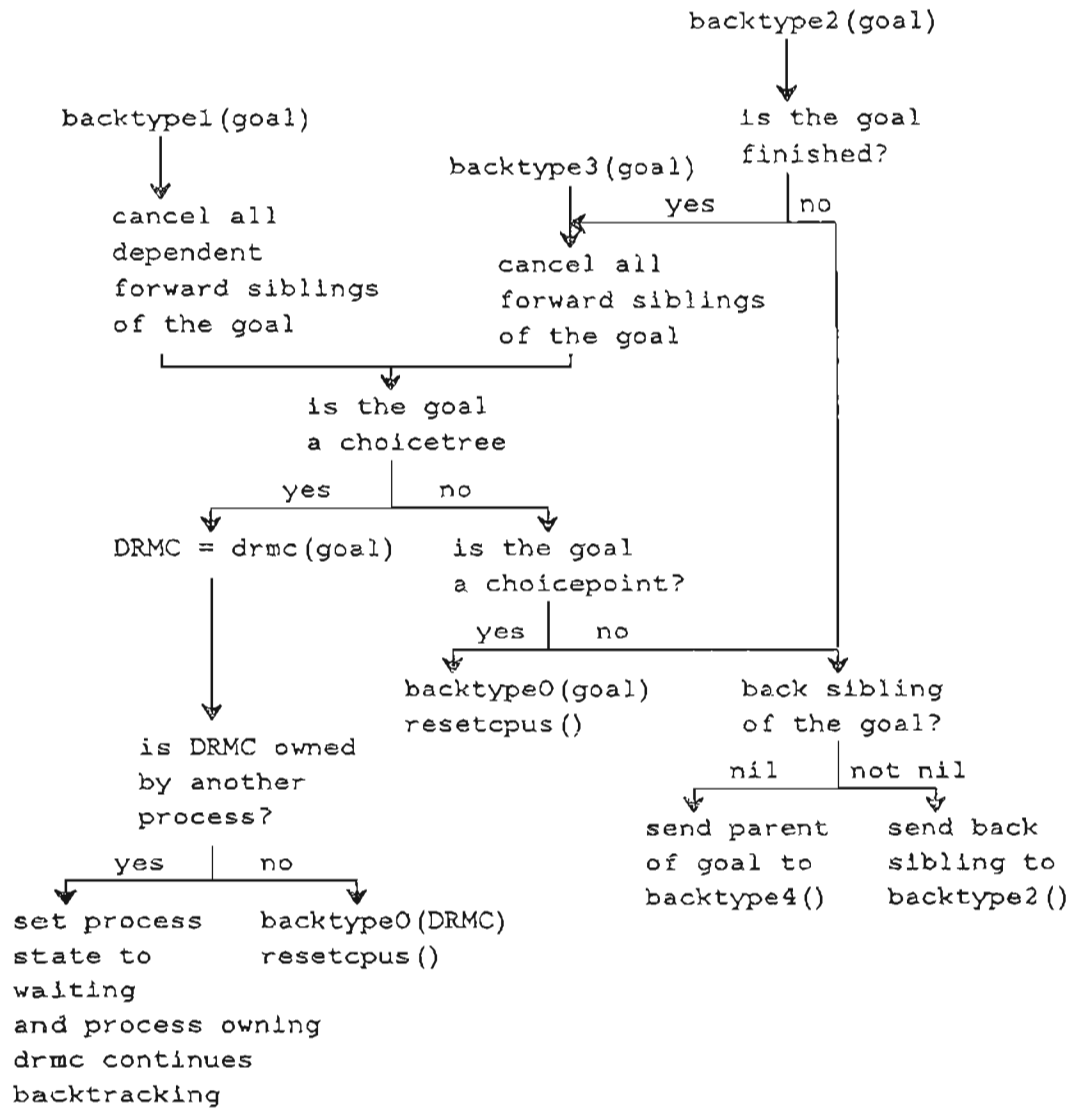
The `ibacktype1`, `ibacktype2`, and `ibacktype3` interrupts are issued to the process owning the current goal when a process in backward execution reaches `backtype1()`, `backtype2()`, or `backtype3()` and the current goal is owned by another process. After receiving the interrupt, the process owning the *receiver* executes the appropriate `backtype()` with the *receiver* as its argument. Each of these routines require the executing process to issue an

`icancelwait` interrupt to itself and cancel the current goal's forward siblings. The forward sibling goals must be cancelled by the process owning them, thus the executing process may have to issue `icancel` interrupts. If `icancels` are issued, the executing process must wait for all `icancels` to complete before continuing. In addition, whenever a `backtype()` routine crosses a process boundary, the executing process issues an interrupt and is put into a *waiting* state until forward execution resumes. See Figure 5.13 for a flow chart of these routines.

Unlike sequential backward execution, `backtypel()` is a routine called by `backtrack()` (via `calltypel()`) in parallel backward execution. This call to `backtypel()` in `backtrack()` results from the possibility that the process executing `backtrack()` does not own the current goal at the point where `backtypel()` is executed in `backtrack()`. In this situation, an `ibacktypel` interrupt must be issued to the process owning the current goal for backward execution to continue.



Backtrack () Flow Chart
Figure 5.12

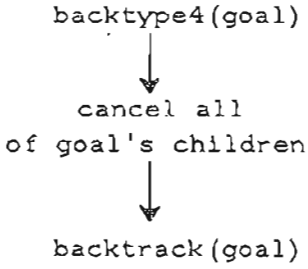


Parallel Backtype() Routines
Figure 5.13

backtype4():

The ibacktype4 interrupt is issued when the process engaged in backward execution reaches backtype4() and the current goal is owned by another

process. The process executing `backtype4()` begins by canceling each of the current goal's children and continues by sending the current goal to `backtrack()`. See Figure 5.14 for a flow chart illustrating `backtype4()`.



Parallel `Backtype4()` Routine
Figure 5.14

5.5. Backward Parallel Execution Example

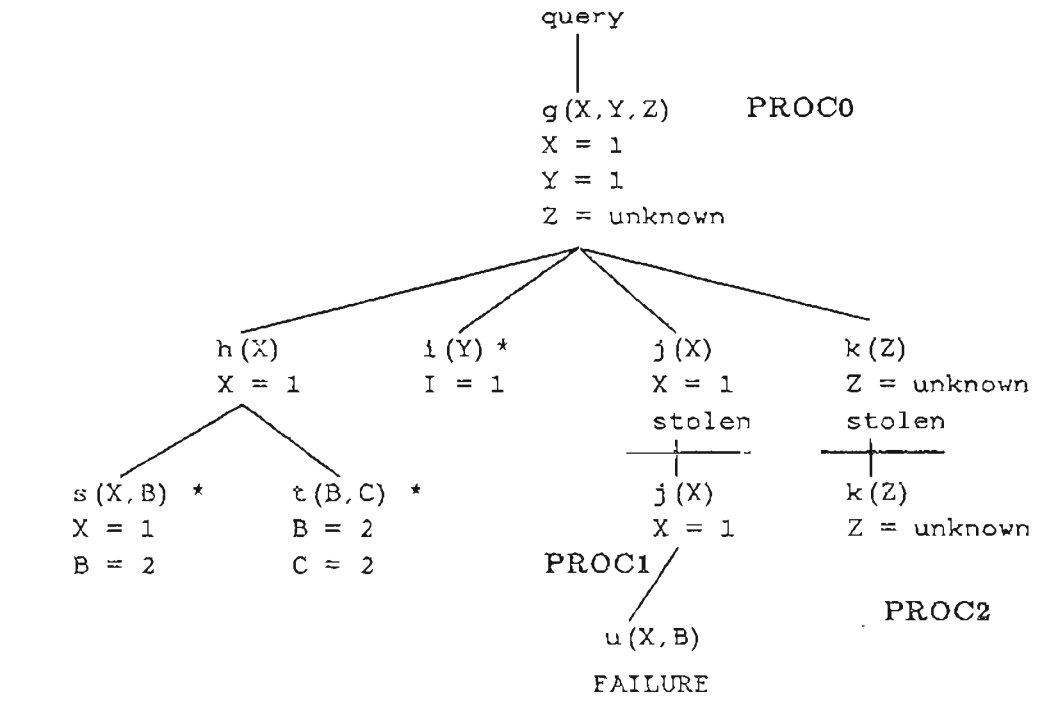
The parallel version of the program `G` is:

```
query :- g(X, Y, Z).

g(X, Y, Z):- h(X),
             par( i(Y), j(X), k(Z) ).

h(A)  :- s(A, B), t(B, C).
i(1).
i(2).
i(3).
j(A)  :- u(A, B), v(B).
k(A)  :- par( l(B), m(C) ).
n(A, B, C).
s(1, 2).
s(2, 3).
t(2, 2).
t(3, 3).
u(2, 2).
v(2).
l(2).
m(1).
m(2).
n(1, 2, 2).
```

This example of backward parallel execution of G involves three processes; proc0, proc1, and proc2. As stated earlier, backward execution begins when forward execution encounters a failure. When more than one process executes a program, it is not certain which process will execute a failing goal first, hence, the synchronization of the processes is difficult to determine. In this example, two processes execute failing goals, but the order in which the processes fail is not critical for demonstrating backward execution. Thus, the order in which processes fail is not the key issue.

Proc0 begins forward execution of the program G and when it executes the execution graph expression, par, the goals j(X) and k(Z) are put on it's *availist*. The processes proc1 and proc2 steal the goals j(X) and k(Z), respectively, from proc0's *availist* and begin forward execution. The first failure occurs on proc1 at the

goal u(X, B) while proc0 executes i(Y) and proc2 executes k(Z). The proof tree at this point is:



Proof Tree after icancel Interrupts
Figure 5.15

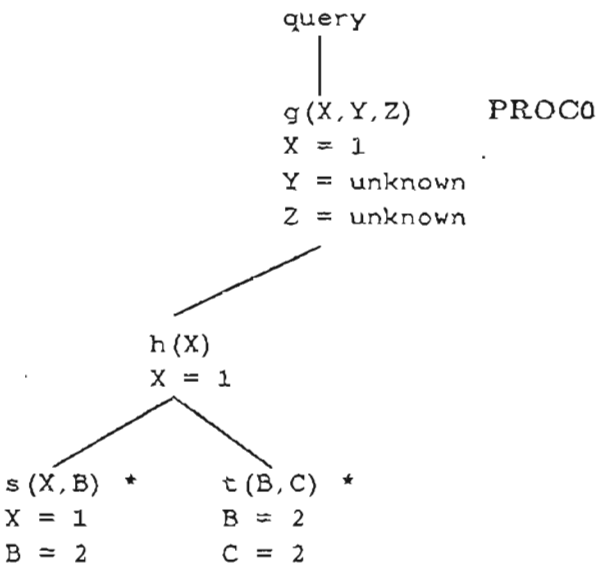
As proc1 fails, proc0 and proc2 continue forward execution of their current goals.

Proc1 executes the backtrack() routine and changes its state from *normal* to *backtracking*. Backtrack() determines that the failing goal, u(X, B), is not a choicepoint and it does not have a *smartchoice*. Thus, the parent, j(X), is sent to calltype4(). The calltype4() routine determines that both the failing goal

and its parent are owned by proc1, thus proc1 executes `backtype4()` to continue backward execution. In `backtype4()` proc1 cancels its child `u(X, B)` and then sends the goal `j(X)` to `backtrack()`, which determines that the *smartchoice* of `j(X)` is `h(X)`. Since `h(X)` is owned by proc0, proc1 continues in `backtrack()` to `calltype1()` with `h(X)` as its argument. Once in `calltype1()`, proc1 issues an `ibacktype1` interrupt to proc0 with `h(X)` as the *receiver* goal, and changes its process state to *waiting*. Proc1 has completed its part in backward execution and waits for forward execution to resume.

Proc0 finds the `ibacktype1` interrupt on its *intlist*, changes its state to *backtracking*, and executes `backtype1()` with `h(X)` as its argument. In `backtype1()`, proc0 cancels each of its *dependent* forward siblings; `i(Y)`, `j(X)`, and `k(Z)`. Although, in this example, `i(Y)` and `k(Z)` are not directly dependent on `h(X)`, they could be dependent on `h(X)` and must be cancelled. Proc0 cancels the goal `i(Y)`, and since `j(X)` and `k(Z)` are stolen, proc0 cancels its copy of the original stolen goals, issues an `icancelwait` interrupt to itself, and issues `icancel` interrupts to proc1 and proc2 for their copies of the goals.

After the `icancel` interrupts are completed, proc1 and proc2 lock and decrement the *nosent* field in proc0's `icancelwait` interrupt and are put into the *waiting* state. The proof tree after the `icancel` interrupts are completed is reduced to that in Figure 5.16.



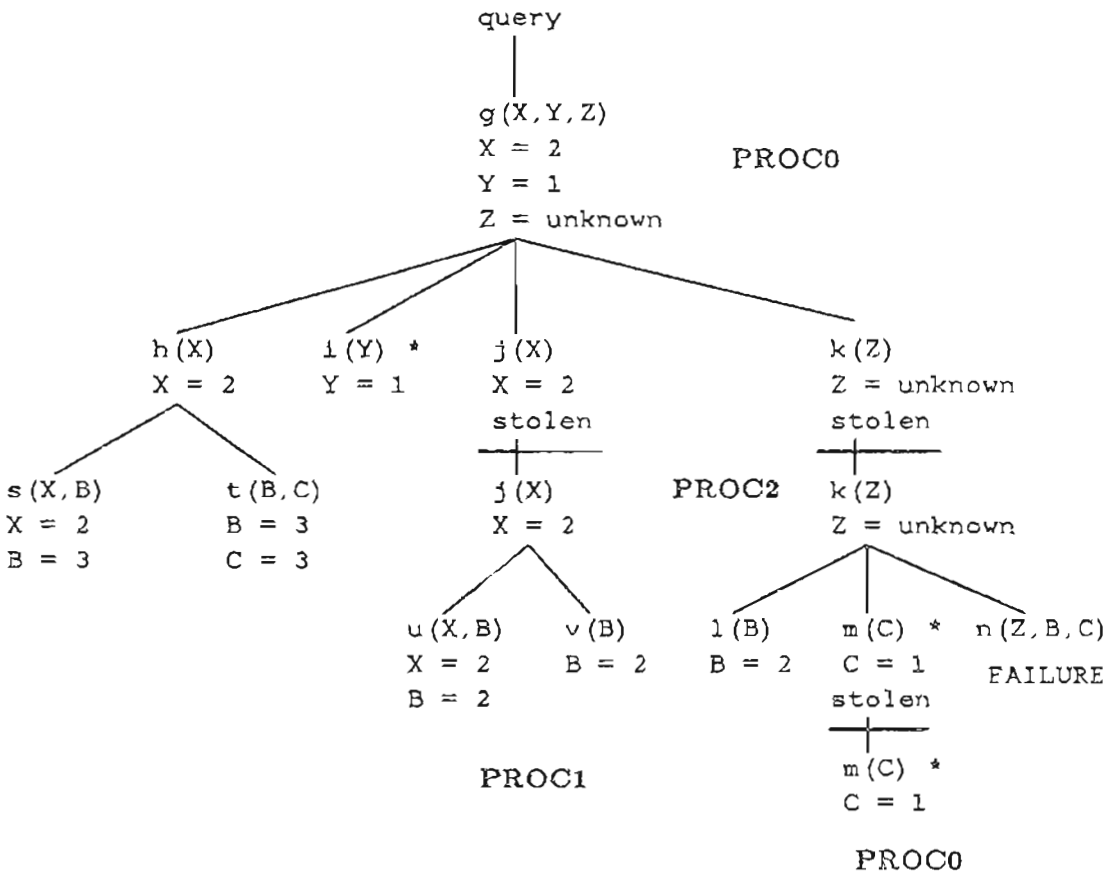
Proof Tree at First Failure
Figure 5.16

Proc0 continues backward execution by determining that `h(X)` is a choicetree and hence, searches for the drmc in `h(X)`'s subtree. The goal `t(B, C)` is the drmc and is owned by proc0. Proc0 continues backward execution in `backtype0()` where the structures are reset and `t(B, C)` becomes proc0's current goal. Proc0 then executes `retinterrupt()` and `resetcpus()`. `Retinterrupt()` removes the interrupt structure from proc0's *intlist* and `resetcpus()` changes proc0's process state back to *normal* and proc1 and proc2's process state to *free*. At this point, the proof tree is the same as Figure 5.9 in the backward sequential execution example and forward execution is ready to continue on proc0 at `t(B, C)`'s

flabel, `t(3, 3)`. The other processes, proc1 and proc2, are *free*, searching for goals to steal, while proc0 is *normal*, executing `t(B, C)`.

Proc0 continues forward execution and puts the goals `j(X)` and `k(Z)` on its *availist* after executing `par`. Again, proc1 steals `j(X)` and proc2 steals `k(Z)`. The goal `u(X, B)` does not fail with these new bindings, thus, when proc1 and proc0 finish their work they must wait for proc2 to complete its tasks and decrement *no_of_forks* in `g(X, Y, Z)` to zero.

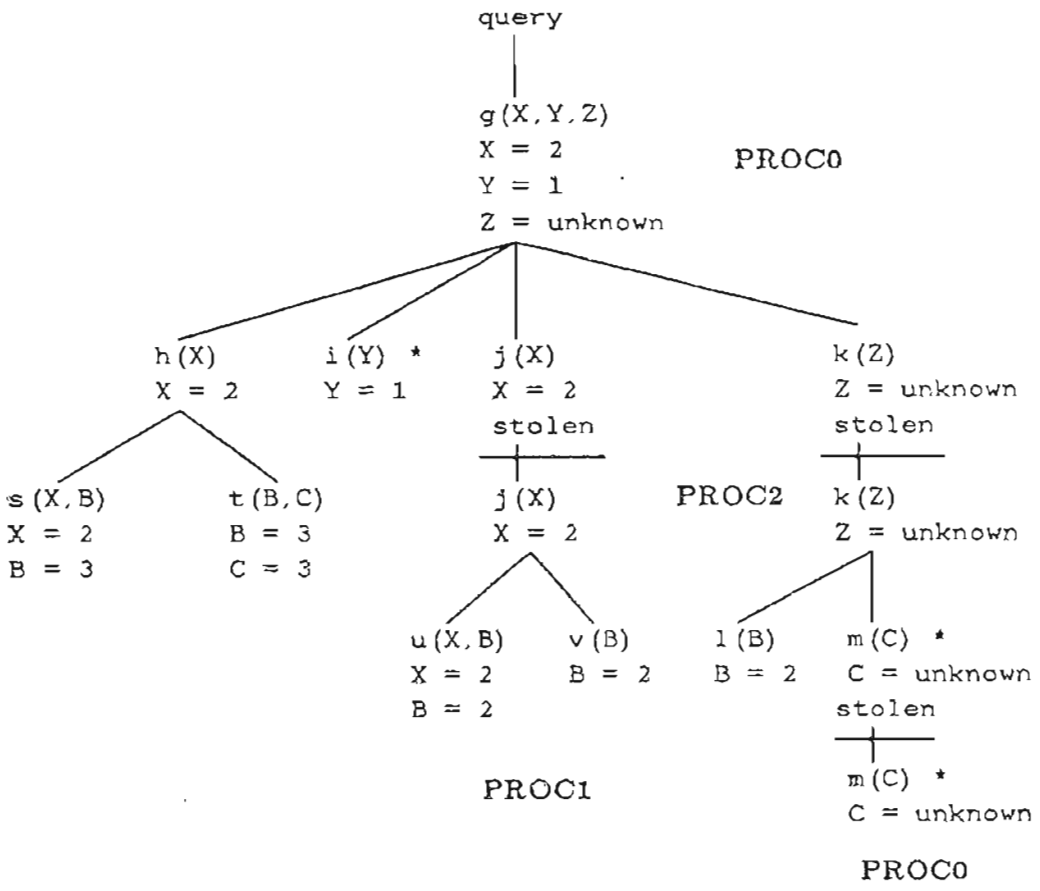
During its forward execution, proc2 reaches a `par`, sets *no_of_forks* in `k(Z)` to 2, and puts the goal `m(C)` on its *availist* for a *free* process to steal. Either proc0 or proc1, or both, may be *free* at this point. If neither proc0 or proc1 are *free*, then proc2 executes the goal itself. For this example, proc0 steals the goal `m(C)` and executes it while proc2 executes `l(B)`. After finishing its goal `l(B)`, proc2 locks and decrements *no_of_forks* in `k(Z)`'s goal structure and takes the goal `n(Z, B, C)` from its *availist*. Proc0 also finishes its goal, `m(C)`, locks and decrements `k(Z)R's` *no_of_forks*, and is put in the *free* state since there are no more goals on its *availist*. Proc2 fails at the goal `n(Z, B, C)` in the proof tree below.



Proof Tree at Second Failure
Figure 5.17

Backward execution begins with proc2, which changes its process state to *backtracking* and executes `backtrack()` with the failing goal as the argument. The failing goal, `n(Z, B, C)`, is not a choicepoint, but its *smartchoice*, `m(C)` is. Proc0 owns the *smartchoice*, so proc2 issues an `ibacktype1` interrupt to proc0 with `m(C)` as the *receiver*, and changes its state to *waiting*. Proc0 receives the

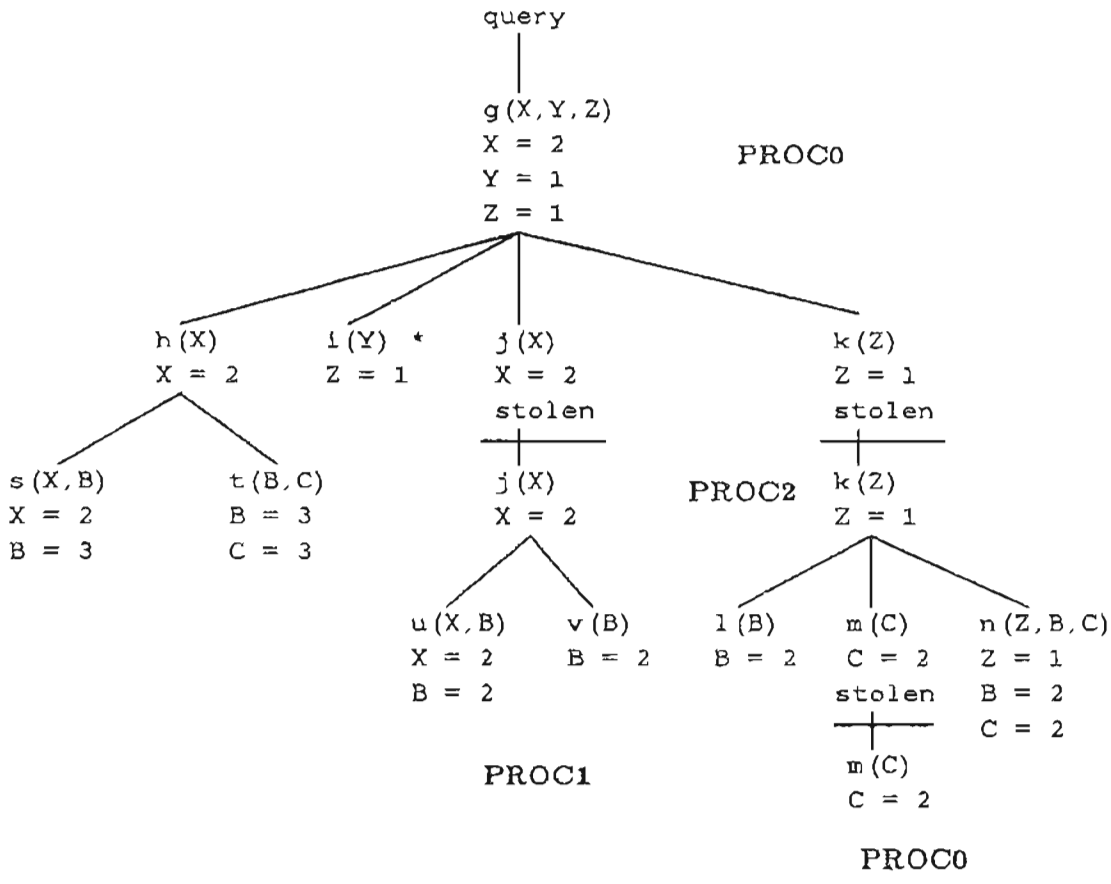
ibacktype1 interrupt and executes backtype1(). In backtype1(), proc0 cancels the goal m(C)'s forward dependent sibling, n(Z, B, C) by issuing an icancelwait interrupt to itself and an *icancel* interrupt to proc2. Proc2 receives the icancel interrupt and removes n(Z, B, C) from the environment. The *nosent* field in proc0's icancelwait is locked and decremented to zero and proc0 continues with backtype0(). At this point, the structures are reset and m(C) becomes proc0's current goal in the proof tree as depicted in Figure 5.18.



The Proof Tree after icancel Interrupt
Figure 5.18

Proc0 executes retinterrupt() ro remove the ibacktype1 interrupt from its *intlist* and continues forward execution with the current goal, m(C) at its *flabel*, m(2). Meanwhile, proc2 waits for k(Z)'s *no_of_forks* to decrement to zero and proc1 looks for goals to steal. After proc0 finishes m(C), its state is changed to *free* and proc2 continues forward execution with n(Z, B, C), which succeeds. At this point, k(Z) is finished so g(X, Y, Z)'s *no_of_forks* is decremented to zero

and proc0 completes forward execution by claiming a success. The proof tree for this success is:



Success Proof Tree
Figure 5.19

After the solution is presented, the user may request PAPI to search for more solutions to the query. If more solutions are desired, then the deepest rightmost child of the entire proof tree, `n(Z,B,C)` is failed and the process owning `n(Z, B,`

C) begins backward execution in `backtrack()` to find another solution to the query. This involves much more backward execution, therefore, it is not included as part of this example. Walking through the proof tree in Figure 5.19 and following the parallel `backtrack()` and `backtype()` routines closely, however, will eventually lead the reader to the next solution:

```
X = 2
Y = 2
Z = 1
```

The example above covered the situation in which proc1 failed first and then after all backward execution completed, proc2 failed. This simplistic situation is not always the case for parallel programs, but makes parallel program examples easier to present. It often happens that processes fail while other processes are backtracking. This situation does not create problems since 1) processes do not check their *intlists* for interrupts until they are in a position to service them and 2) if the receiving goal has been removed already (i.e., the process removed the goal for some other reason while the interrupt was waiting to be serviced), then the process simply ignores the interrupt and removes it in `retinterrupt()`. Therefore, if proc1 is backtracking while `n(Z, B, C)` fails on proc2, proc1 and proc2 issue their interrupts to proc0, and these are put on proc0's *intlist* in a first-come-first-served order. If proc2's `ibacktype1` interrupt is received first, then proc0 cancels `n(Z, B, C)` and resumes forward execution before finding proc1's *backtype1* interrupt. This time, proc0 cancels `i(Y)`, `j(X)`, and `k(Z)` and then resumes forward execution at `t(B, C)`'s *flabel*. Note that while there is no clashing if proc2's interrupt is serviced

first, the work done by the first interrupt will be redone since the goal m(C) will fail again as a result of k(Z) being cancelled.

CHAPTER 6

Benchmark Testing and Analysis

This Chapter analyzes the performance of PAPI based on its parallel execution of several benchmark tests. The Prolog programs selected as benchmarks are `fibonacci`, `quicksort`, `mapcolor`, `deriv`, and `partiming`.[1]. The `quicksort` program, which sorts a list of elements, and the `fibonacci` program are deterministic programs that exhibit PAPI's "number crunching" abilities, while `mapcolor` helps in analyzing PAPI's semi-intelligent backtracking and communication schemes. The `partiming` and `deriv` programs, which analyze parallel execution and determine the derivative of a value, yield results that are compared to those obtained by Hermenegildo's AND-parallel Abstract Machine simulator [Her86].

The benchmark tests were compiled and assembled on Oregon Graduate Center's VAX 11/780 using Prolog V2.5 and the opcode instruction files were executed by PAPI on mkt3, a Balance 21000 computer made by Sequent Computer Systems. Mkt3 runs DYNIX and 15 processors were available at the time that the benchmark tests were executed. Throughout benchmark testing, the load average was approximately 0.0 and no other users were using the mkt3 machine.

[1] See Appendix C for the Prolog source code of the benchmark tests.

6.1. Preliminaries

Parallel execution of an assembled Prolog program using PAPI is initiated with the command line:

`execute` <*file*> [*flag*] [*datafilename*]

where *file* is the opcode file for the Prolog program created by the assembler described in Chapter 2, *flag* is any character or string indicating that a datafile is to be created, and *datafilename* is the name for the datafile. *Flag* must be set to save timing data returned by PAPI.

As PAPI `fork()`s the number of processes specified by the user, DYNIX puts each process on a separate processor, as long as there is a processor available. Since there were no other users on the machine during benchmark testing, all 15 processors were available. At this point, forward execution and data collection begins.[2]

6.2. Data

Data collected by PAPI consists of: the initialization time, *total_init_time*; the time to execute the program, *total_time*; the ideal time in which there is no charge for system calls, *ideal_time*; and the number of goals that each processor steals, *goals_stolen*. In addition, a value for *final_time*, *total_time* minus *total_init_time* is presented with the data above. Note that the execution time spent by each individual processor in forward execution, backward execution, and looking for work is not

[2] For the remainder of this discussion, "process" and "processor" are synonomous.

dual processor in forward execution, backward execution, and looking for work is not recorded due to the large amount of overhead resulting from implementing a timing scheme over many short periods of time. As a result, analysis of the benchmark results can only suggest where most of a processor's execution occurs.

Total_time and *total_init_time* are collected using the system call `getrusage()`, which is accurate to 10 milliseconds. *Total_time* is the duration of PAPI's execution of the Prolog opcode program, but does not include user response time (to PAPI's queries) or the creation and initialization of the datafile. *Total_init_time* is PAPI's initialization time: the amount of time it takes to fork the child processes, put the child processes to sleep with the `sigpause()` system call, and load the Prolog opcode instruction file. Both *total_time* and *total_init_time* are broken into the components of user time, *usr_time*, and system time, *sys_time*. The *total_time*'s *sys_time* and *usr_time* times are accumulated throughout execution in *total_sys_time* and *total_usr_time*.

Final_time is *total_time* minus *total_init_time*. Since *total_init_time* increases as the number of processors increase and consists almost entirely of `fork()`ing time, it is subtracted from *total_time*. Thus, *final_time* represents how long it takes PAPI to execute the Prolog program. *Ideal_time* is *final_time* minus *total_sys_time* and represents the ideal world of no charge for system calls. *Ideal_time* is important to examine since *total_sys_time* varies for the number of processors. For example, *total_sys_time* includes the time to lock and unlock structures (but not the amount of time that a processor spins in the spinlock), to print the answer, and to wake the child processors. As a result, it is expected that *total_sys_time* will increase as the

systems, thus an ideal world in which PAPI would not depend on an operating system for these services is represented through *ideal_time*.

6.3. A Sample Run

As an example, the program `mapcolor` is executed and a datafile is created. The execution and datafile are presented below.

```
Script started on Sat Jul  4 12:49:37 1987
% execute tests/mapcolor.as p map


                Parallel Prolog Opcode-Interpreter

Enter number of processes desired (1 - 15) -> 6

Creating map


query succeeds on proc 0
var0 = red
var1 = blue
var2 = yellow
var3 = blue
Backtrack on proc0? (y/n)->  y

query succeeds on proc 0
var0 = blue
var1 = red
var2 = yellow
var3 = red
Backtrack on proc0? (y/n)->  y

query succeeds on proc 0
var0 = yellow
var1 = red
```

```
var2 = blue
var3 = red
Backtrack on proc0? (y/n)->  y

query succeeds on proc 0
var0 = red
var1 = yellow
var2 = blue
var3 = yellow
Backtrack on proc0? (y/n)->  y

query succeeds on proc 0
var0 = blue
var1 = yellow
var2 = red
var3 = yellow
Backtrack on proc0? (y/n)->  y

query succeeds on proc 0
var0 = yellow
var1 = blue
var2 = red
var3 = blue
Backtrack on proc0? (y/n)->  y

query fails on proc0
%



% cat map

Date: Sat Jul  4 12:50:07 1987
Program: tests/mapcolor.as
Number of Procs: 6
Machine: mkt3

Initialization:
usr_time = 160 milliseconds
sys_time = 3160 milliseconds
total_time = 3320 milliseconds


        var0 = red
        var1 = blue
```

```
        var2 = yellow
        var3 = blue

usr_time = 470 milliseconds
sys_time = 40 milliseconds
total_time = 3830 milliseconds


        var0 = blue
        var1 = red
        var2 = yellow
        var3 = red

usr_time = 160 milliseconds
sys_time = 0 milliseconds
total_time = 3990 milliseconds


        var0 = yellow
        var1 = red
        var2 = blue
        var3 = red

usr_time = 230 milliseconds
sys_time = 20 milliseconds
total_time = 4240 milliseconds


        var0 = red
        var1 = yellow
        var2 = blue
        var3 = yellow

usr_time = 300 milliseconds
sys_time = 30 milliseconds
total_time = 4570 milliseconds


        var0 = blue
        var1 = yellow
        var2 = red
        var3 = yellow

usr_time = 200 milliseconds
sys_time = 30 milliseconds
```

```
total_time = 4800 milliseconds


        var0 = yellow
        var1 = blue
        var2 = red
        var3 = blue

usr_time = 140 milliseconds
sys_time = 30 milliseconds
total_time = 4970 milliseconds


Query fails
usr_time = 190 milliseconds
sys_time = 0 milliseconds
total_time = 5160 milliseconds



PAPI Timing Totals
total_time = 5160 milliseconds
total_init_time = 3320 milliseconds
total_sys_time = 150 milliseconds
final_time     = 1840 milliseconds
ideal_time     = 1690 milliseconds

proc[0]: 0 goals_stolen
proc[1]: 28 goals_stolen
proc[2]: 29 goals_stolen
proc[3]: 32 goals_stolen
proc[4]: 31 goals_stolen
proc[5]: 29 goals_stolen
```

6.4. Test Results

This section presents the results of the benchmark testing. The benchmark programs were executed several times and the run exibiting the best results[3] for the

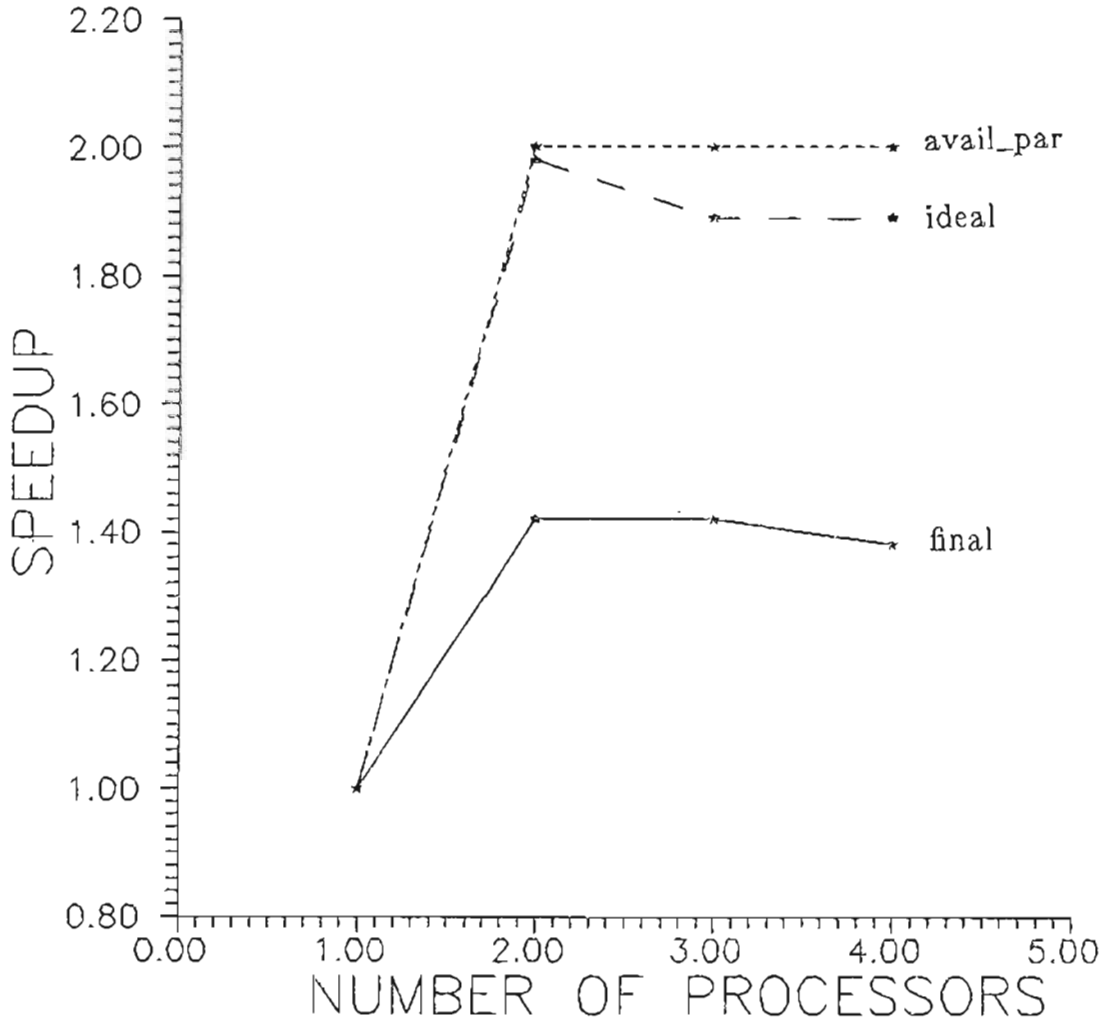
benchmark is graphed below. Each graph, except mapcolor's, contains three curves; the program's *final* curve, the program's *ideal* curve, and the program's theoretical speedup, *avail_par*. The *final* and *ideal* curves illustrate the programs' speedup derived from its *final_time* and *ideal_time* respectively. The *avail_par* curve for a program is determined from the amount of parallelism available in the program and illustrates the theoretical speedup that can be obtained from the program. The *avail_par* curve calculation for a program involves an analysis of: the execution graph expressions in the program, the number of parallel and sequential goals in the program, and the number of processors executing the program. Detailed calculations for each of the *avail_par* curves below are presented in Appendix D. Due to the difficulty in determining the amount of parallelism in backward execution, mapcolor's *avail_par* curve was not calculated.

6.4.1. Partiming

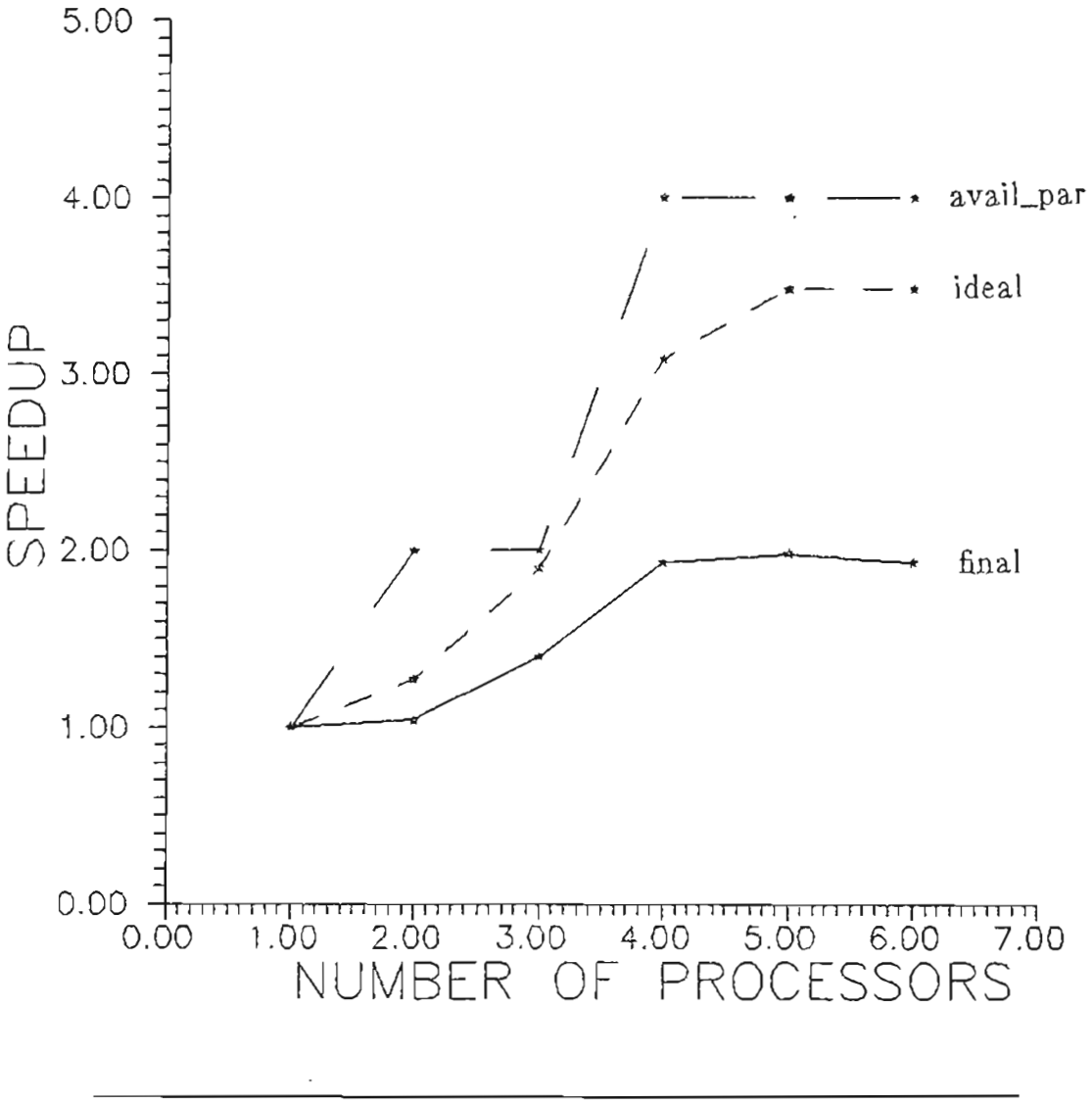
The partiming programs were designed to have easily detectable and exploitable parallelism such that parallel execution of PAPI can be tested accurately. This easily exploitable parallelism in the partiming programs is illustrated in the partiming4 benchmark program:

[3] The fastest run for the benchmark programs is only 10 to 20 milliseconds faster than its slowest run.

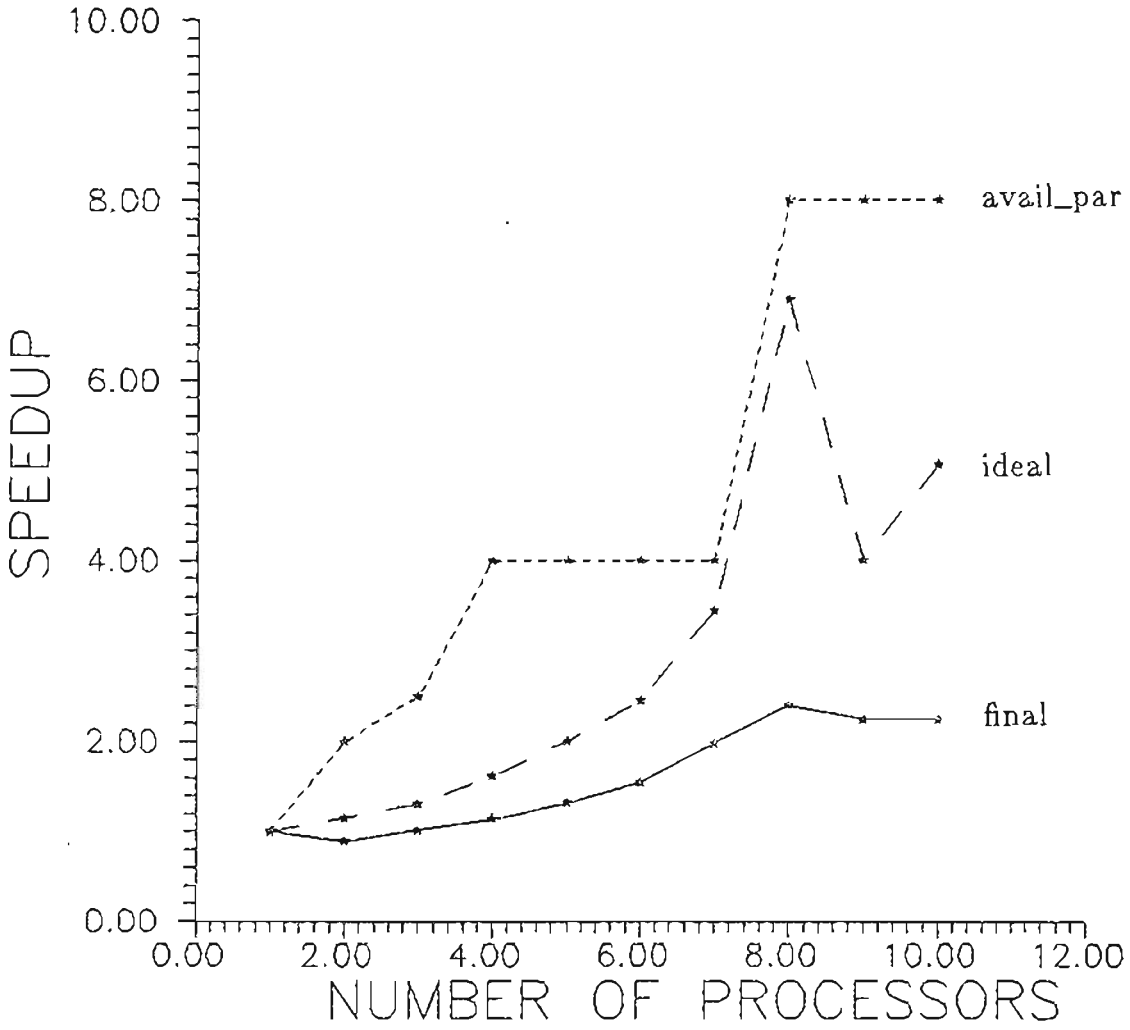
```
query:- times4(X), gpar([X],
                        p(X), p(X), p(X), p(X)
                       ).

p(0).
p(X):- Y is (X-1), p(Y).

times4(32).
```



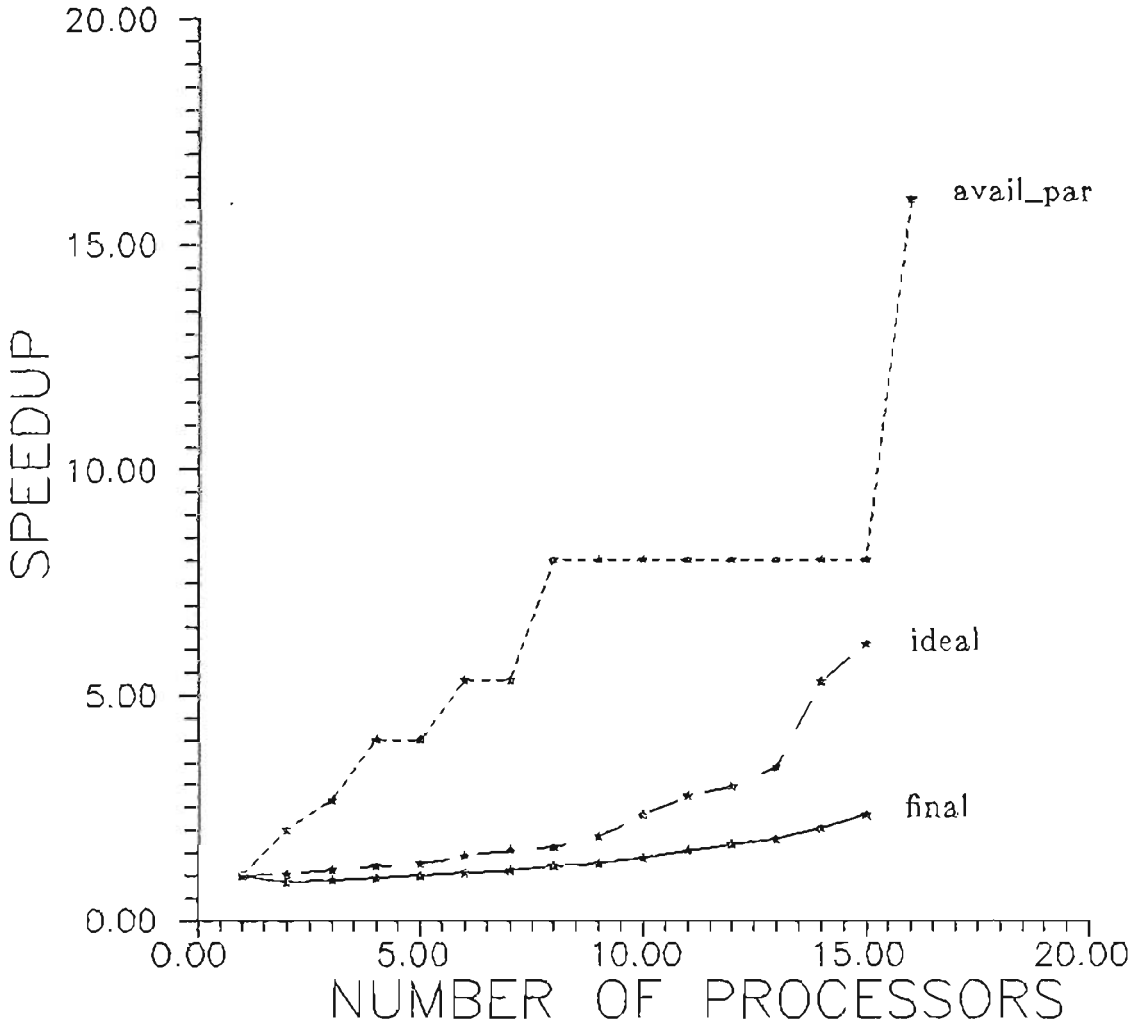
Partiming2's Speedup Graph
Figure 6.1



Partiming4's Speedup Graph
Figure 6.2



Partiming8's Speedup Graph
Figure 6.3



Partiming16's Speedup Graph
Figure 6.4

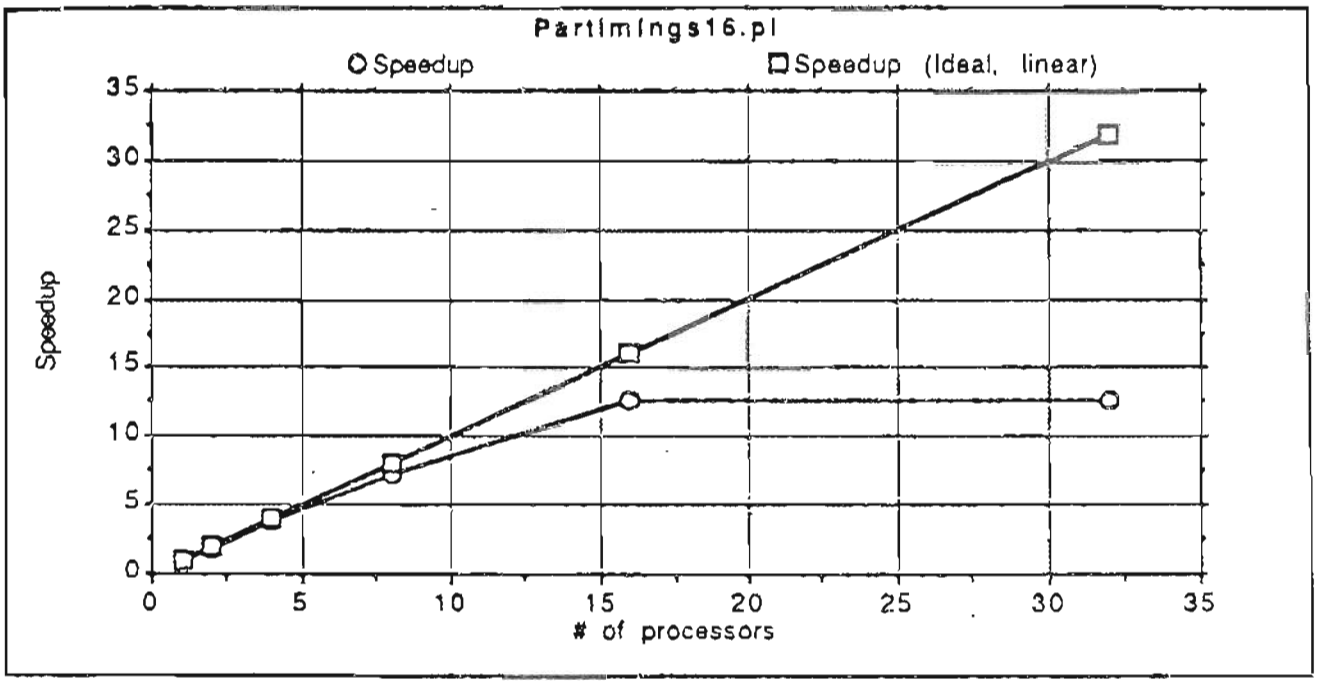
PAPI's execution of the `partiming` benchmark programs was generally more successful than its execution of the other benchmark programs. The decrease towards the end of the *ideal* `partiming` curves (Figures 6.1 through 6.3) reflects an increased overhead when there is not enough work to distribute to each of the

processors executing the program. That is, one or more processors remain in the *free* state throughout the entire execution of the benchmark program. In addition, the gap between the *ideal* and *final* curves indicates that a significant amount of the program's execution time is spent servicing system calls (i.e., waking up the child processors and locking and unlocking shared data structures). For the remaining discussion, a benchmark's *ideal* curve will be analyzed rather than its *final* curve, since the *ideal* curve is indicative of PAPI's performance without the overhead associated with the DYNIX operating system.

The *ideal* curves for `partiming2` and `partiming4` in Figures 6.1 and 6.2 increase with the *avail_par* curve as expected. `Partiming8`'s *ideal* curve indicates a large overhead in `partiming8`'s execution by 1 to 7 processors, but a much smaller overhead for execution by 7 and 8 processors. Since timing data was not collected for the individual processors, it is not possible to determine exactly how much time was spent by each processor in `steal()`. However, it appears, based on `partiming8`'s *ideal* curve and the complexity of `steal()`, that stealing is expensive when a processor steals more than one goal. Thus, executing a program with too few processors may introduce more overhead than executing the program with the ideal number of processors.

The *ideal* curve for `partiming16` closely resembles that of `partiming8` in Figure 6.3. The greatest overhead in `partiming16` occurs from 1 to 14 processors and the best speedup occurs using 15 processors. Unfortunately, only 15 processors were available at the time of testing, thus the speedup for `partiming16` using the ideal number of processors, 16, was not obtainable.

As stated in the beginning of this Chapter, the `partiming` programs were tested as benchmarks to compare PAPI's performance to that of Hermenegildo's Abstract Machine. Hermenegildo executed `partiming16`, from 1 to 32 processors, using a multiprocessor simulator. His results are given in Figure 6.5 below.



Hermenegildo's Simulation Results for `Partiming16`
Figure 6.5

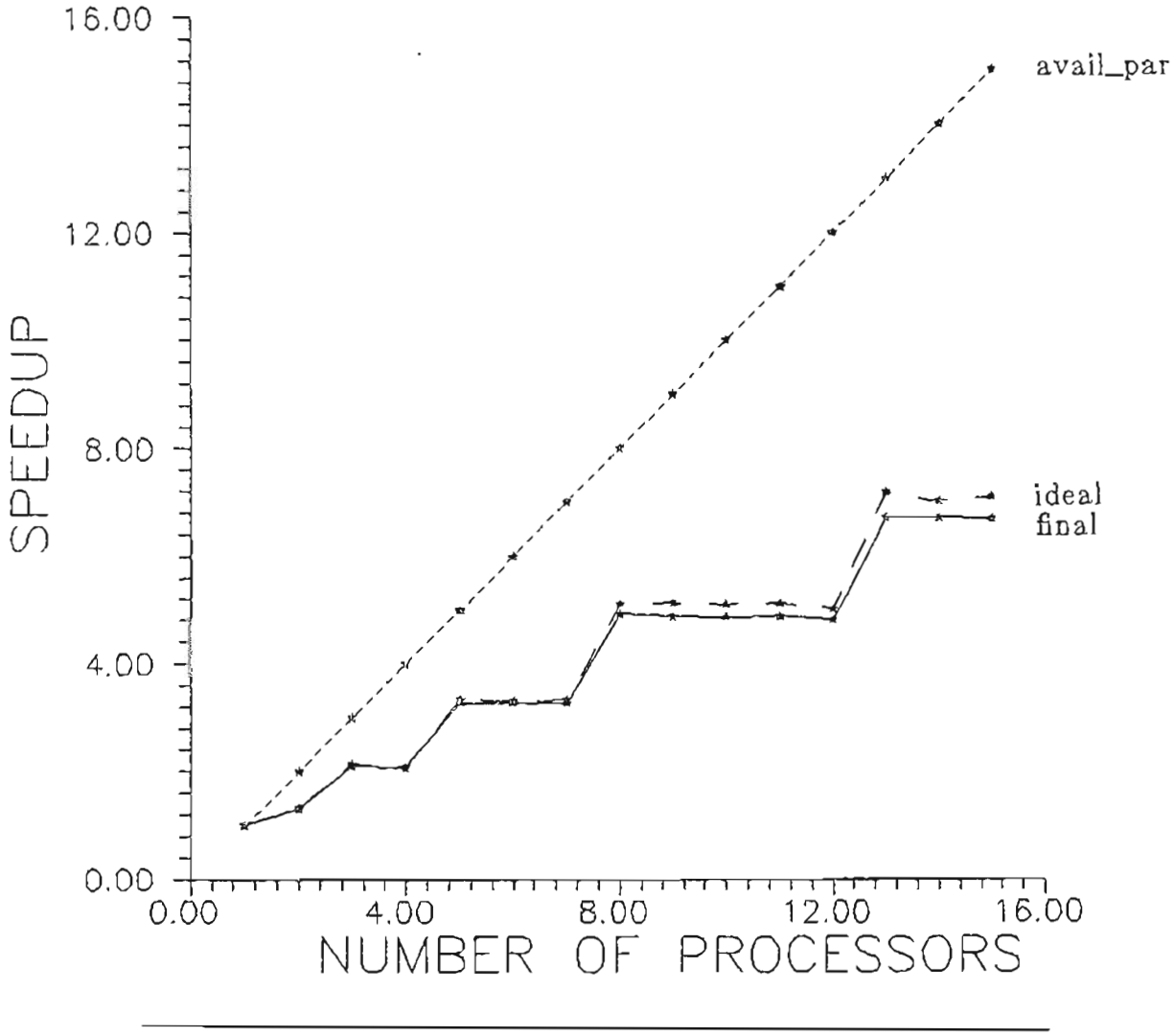
Note that the *ideal* curve in Figure 6.5 and the *avail_par* curve in Figure 6.4 do not agree. According to his graph, Hermenegildo expects linear speedup from this benchmark, but I do not believe that this is feasible, based on the calculations made in Appendix D. Figure 6.5 also indicates that Hermenegildo's simulation

demonstrated a speedup from 8 to 15 processors that exceeds the speedup I found attainable for this program. Further analysis of Figure 6.5, however, indicates that measurements for Hermenegildo's simulator's execution of `partiming16` were only made for 1, 2, 4, 8, 16, and 32 processors, and then these values were connected to create the close to linear speedup curve. Therefore, only the measured values of Hermenegildo's results can be compared to those of PAPI.

Hermenegildo's simulation does not reflect the costs of parallelism and his scheme to find work for *free* processors does not search the proof tree or lock data structures. Rather, his work distribution scheme contains all available goals in a centralized location and *free* processors are given work by a "master" work distributer. As a result, PAPI's overhead from stealing, unequal work distribution, and parallelism costs is not present in Hermenegildo's system.

Hermenegildo's execution of `partiming16` experiences closer to ideal speedup with fewer than 8 processors than it does with 16 processors. PAPI's performance is the opposite. In PAPI's case, as the amount of parallelism in the `partiming` benchmarks and the number of processors executing the benchmark become equal, PAPI's performance peaks. Again, due to the lack of individual processor timing data, I conjecture that this peak performance is related to an equal distribution of work among processors (guaranteed by the benchmark), processors only stealing one goal, and stolen goals yielding a fair amount of work to overcome the overhead of `steal()`.

6.4.2. Fibonacci



Fibonacci's Speedup Graph
Figure 6.6

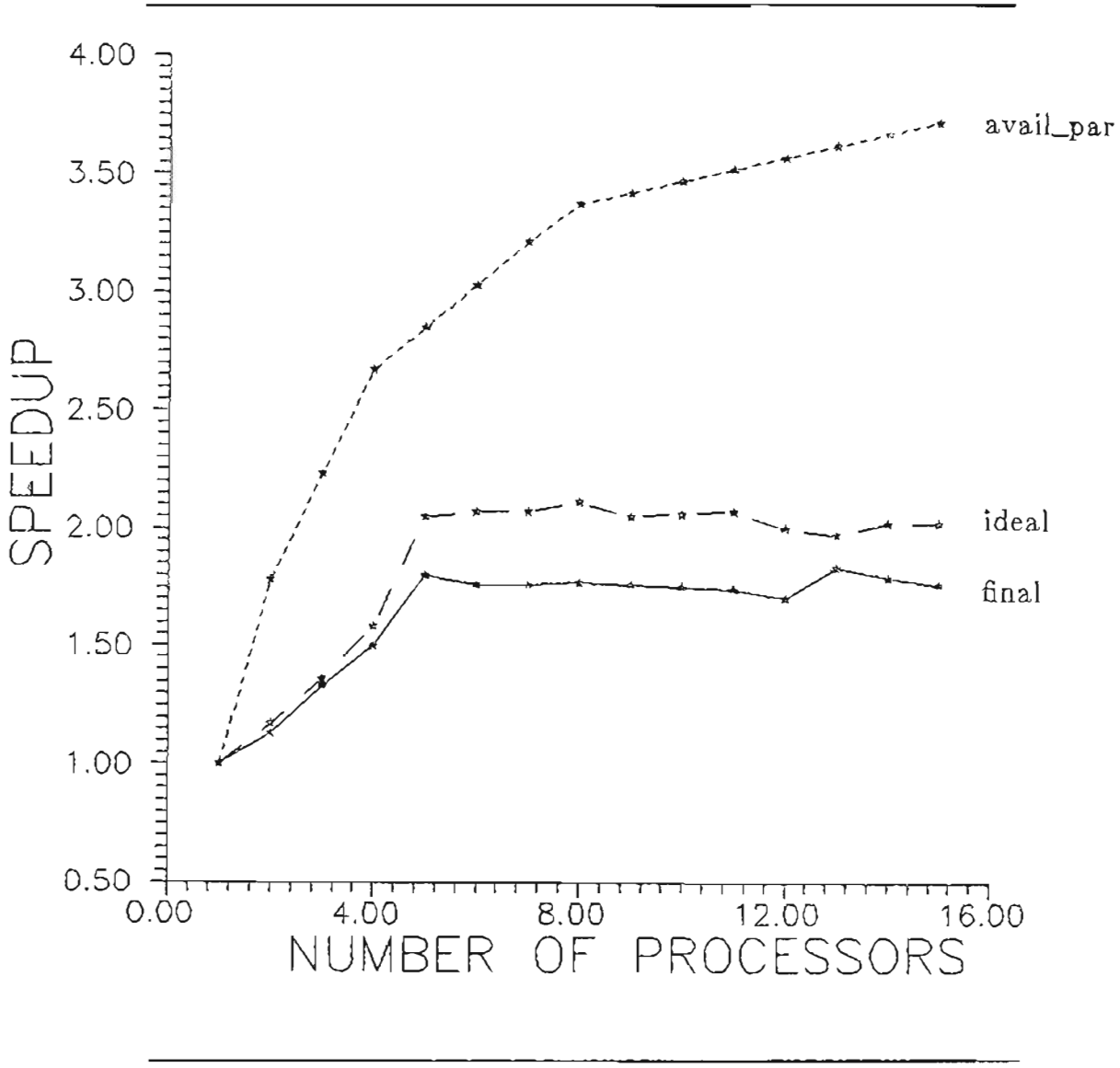
The *ideal* and *final* curves for `fibonacci` in Figure 6.6 reflect an increase in PAPI's performance as more processors are added to its execution, and a small amount of execution time spent executing system calls. The step-like shape of these curves most likely results from the 10 millisecond error for each call to `getrusage()`. This error is a consequence of the limitation of the `getrusage()` system call. The best-fit curve for Figure 6.6 has a slope of approximately 1/2, one half of that of `fibonacci`'s *avail_par* curve.

The `fibonacci` benchmark is a deterministic program that creates a deep proof tree containing several available goals on every other level of the proof tree. Although these available goals yield large subtrees of work for the stealing processor, *free* processors must search this large proof tree for available goals to `steal()`. Since a *free* processor begins its search high in the proof tree and moves down a level if an available goal is not found, this search may become expensive as the proof tree grows. In addition, it is believed that as the number of *free* processors searching for work at the same time increases, the time spent in `steal()` also increases. Therefore, it is suggested that the less than optimal speedup in PAPI's execution of the `fibonacci` benchmark may be contributed to the overhead of `steal()` and perhaps an unequal distribution of work.

6.4.3. Quicksort

The *ideal* and *final* curves in Figure 6.7 indicate the sharpest increase in speedup from 2 to 5 processors and then a less dramatic speedup from 5 to 15 processors. `Quicksort`'s *avail_par* curve illustrates a similar pattern, except that it

follows a logarithmic curve and continues to move slightly upward after 5 processors. The roughness in `quicksort`'s *ideal* and *final* curves is again attributed to the 10 millisecond error in `getrusage()`.

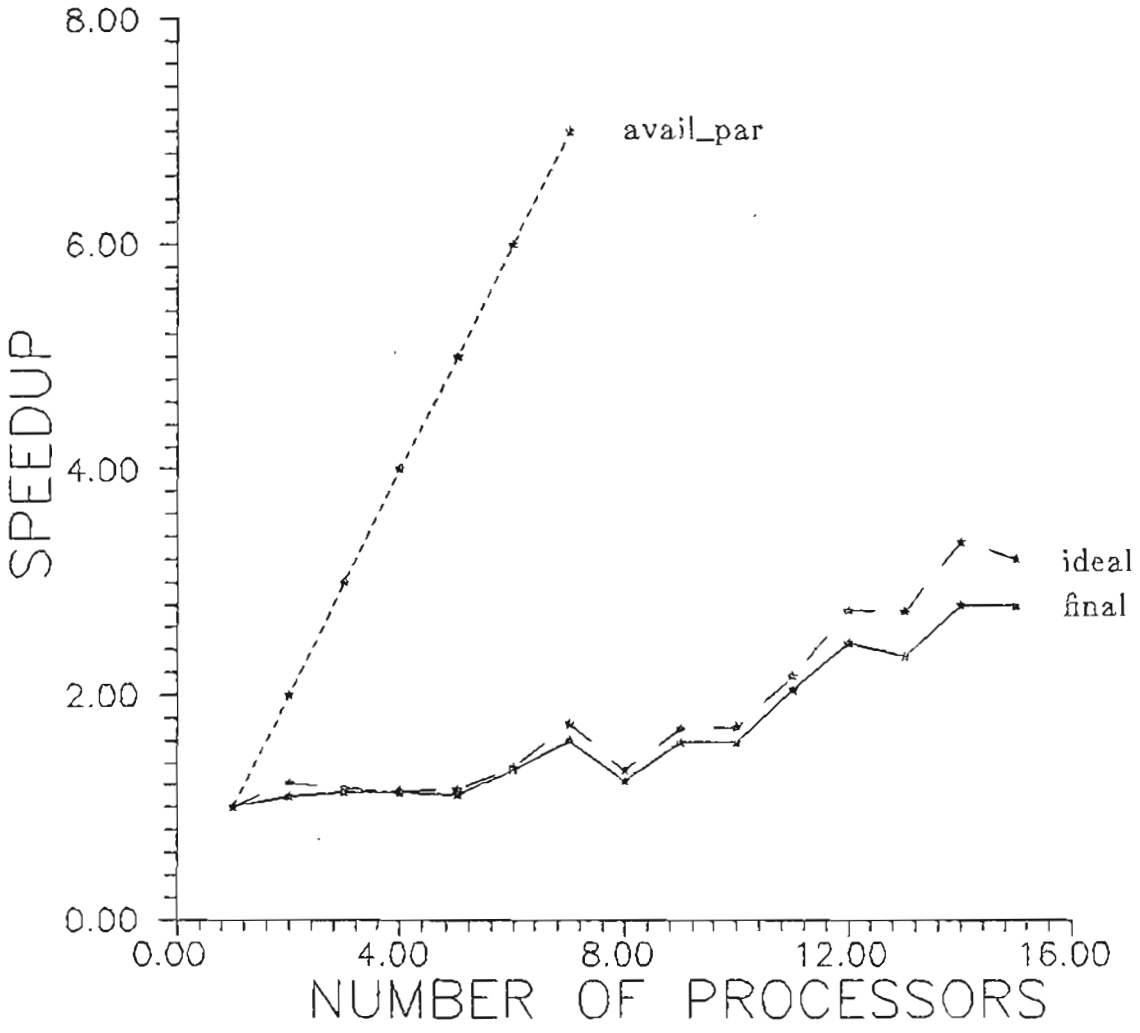


`Quicksort`'s Speedup Graph
Figure 6.7

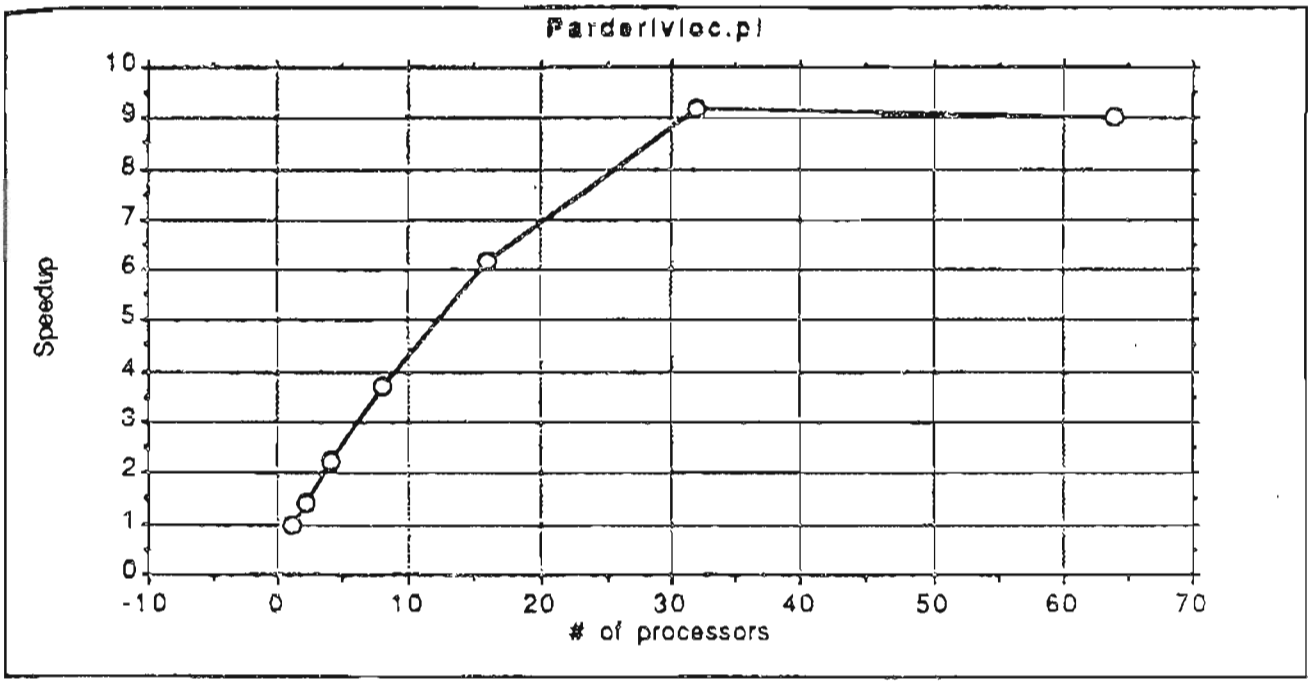
Unfortunately, the `quicksort` benchmark contains a list of only 100 elements, rather than the 128 elements expected when the *avail_par* curve was calculated. In addition, these 100 elements were not listed in any specific order. Hence, the `quicksort` benchmark was not written to ensure that the elements in the list would be distributed evenly among processors or that each goal would render an equal amount of work. Thus, these shortcomings and the lack of timing data for each individual processor prevents a thorough analysis of PAPI's execution of the `quicksort` benchmark.

6.4.4. Deriv

The *ideal* and *final* curves for `deriv`, in Figure 6.8, are not as smooth as those of the other benchmark programs. The roughness of the curves is most likely the effect of `getrusage`'s error and the perhaps the costs associated with parallelism on `deriv`'s short execution time. Note that the *final* and *ideal* curves are close together, indicating that a small amount of PAPI's execution is time spent servicing system calls.



Deriv's Speedup Graph
Figure 6.8



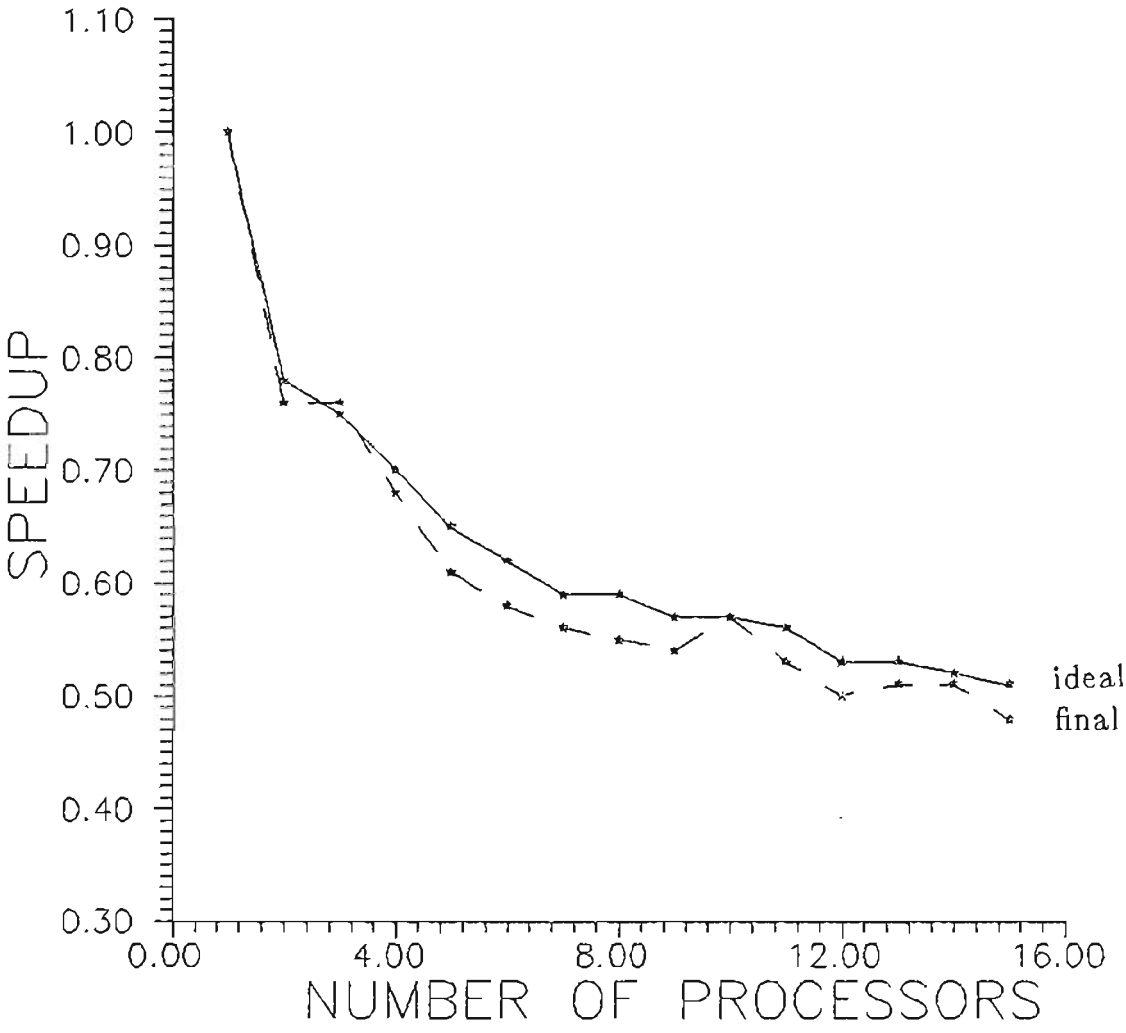
Hermenegildo's Simulation Results for `Deriv`
Figure 6.9

Hermenegildo's simulation results of the `deriv` benchmark, Figure 6.9, illustrate a marked improvement over PAPI's execution, but does not reach the *avail_par* curve calculated for `deriv`. Rather, Hermenegildo claims a speedup of 10 when 32 processors are used. In his analysis, Hermenegildo refers to `deriv` as "a small and not particularly parallel" problem, which contributes to his simulator's sub-linear speedup. He also concludes that the goals available for stealing yield little work for the stealing processors. Thus, processors are forced to steal more often, creating a greater overhead than can be overcome by the work available in the stolen goals.

As discussed in previous sections and in the following section, stealing goals that yield only a small amount of work to the *free* processors may be too expensive. Thus, overhead from `steal()` combined with the costs of parallelism in a real system, rather than a simulation, may attribute to the differences between PAPI's performance and Hermenegildo's simulation of the `deriv` benchmark.

6.4.5. Mapcolor

The *ideal* and *final* curves in Figure 6.10 illustrate a slowdown (i.e., increased execution time) as more processors are added to the execution of `mapcolor`. These curves drop sharply at 2 processors, less dramatically from 2 to 7 processors, and even less dramatically from 7 to 15 processors.



Mapcolor's Speedup Graph
Figure 6.10

The decreased speedup in Figure 6.10 may be explained by the following observations. First, mapcolor requires deep backtracking when a goal fails and when the user requests another solution to the query (which causes the last goal on the proof tree to fail). Parallel backward execution is expensive since it often

requires that only one processor may backtrack or cancel goals while other processors wait for this processor to complete its task. For example, a backtracking processor, proc1, may reach a goal owned by another processor, proc2, or require that a goal that is owned by another processor, proc2, be cancelled. In both cases, an interrupt is issued to proc2. Depending on the type of interrupt issued, proc1 either waits for proc2's return before continuing its backward execution, or waits until forward execution restarts. The worst situation in parallel backtracking, however, is when proc1 issues an *icancel* interrupt to proc2 and then proc2 must issue *icancel* interrupts to other processors, proc3 and proc4. This occurs when descendents of proc2's goal that it is canceling have been stolen by proc3 and proc4. In this case, proc2 must wait for proc3 and proc4 to cancel their copies of the stolen goals (and their stolen goal's descendents) before continuing to cancel its goal and its goal's descendents. Meanwhile, proc1 waits for proc2 to complete its *icancel* interrupt before continuining backward execution.

Second, `mapcolor` creates a shallow proof tree, thus the goals in this program do not expand into deep subtrees of work. `Steal()`ing goals most likely creates a a fair amount of overhead which may be overcome if *free* processors steal goals that keep the processor busy for a significant amount of time. `Mapcolor`'s shallow proof tree suggests that this benchmark program is not able to compensate for the overhead in `steal()`.

In addition to the overhead, `steal()` may indirectly be responsible for an unequal distribution of work among processors. *Free* processors with an empty *goal* stack can steal any goal on any processor's *availist*, but *free* processors with at least

one goal on their *goal* stack only steal specific goals, and *free* processors with goals on their own *availist* do not `steal()`, but `getwork()`. As a result, *free* processors that `getwork()` from their own *availist* may get work faster than those that search the proof tree for work and are more likely to execute the goals on their own *availist*. Thus, `steal()` may also affect the distribution of work by putting stealing processors at a disadvantage to those that `getwork()`.

Finally, there is a price to pay for parallel execution; referred to as the cost of parallelism. Although this cost was not measured, it is conjectured that the overhead from copying structures from one processor's memory to that of another processor, putting processors to sleep, waking processors, locking structures in shared memory, and the synchronization of processors has some effect on PAPI's performance. In addition, this cost will most likely increase as the number of processors executing the Prolog program increases.

Combining these areas of especially high overhead in `mapcolor`'s execution may explain the discouraging situation depicted in Figure 6.10. The sharp decrease at 2 processors for example, may be attributed to one processor waiting for another during backward execution and perhaps overhead from `steal()`. The continued downward slope after 2 processors graph suggests an increase in overhead as the number of processors increase. Hence, based on the results of this benchmark, it appears that it is not advantageous to execute Prolog programs that require extensive backward execution using PAPI.

6.5. Analysis

Execution of the benchmark programs using 1 to 15 processors permitted the evaluation of PAPI's performance in specific circumstances. Although the individual processor timing data was not recorded, analysis of the benchmark results suggests that the most expensive areas in PAPI's execution include parallel backward execution and `steal()`. Parallel backward execution forces sequential execution and, in the case of *icancel* interrupts, forces many backtracking processors to wait for one processor to cancel its goals. The cost of the interrupts and communication was difficult to ascertain since it was lost in the high backtracking overhead and was not directly measured.

Stealing was conjectured to be expensive for several reasons: 1) *free* processors may spend a fair amount of their time searching for available goals, 2) *free* processors may bottleneck in *availist* spinlocks, 3) it is possible that *free* processors with non-empty *availist* may `getwork()` faster than other processors *steal()*, resulting in an unequal distribution of work among processors, and 4) the goals stolen may not always provide enough work to overcome the overhead of stealing. The ideal situation for `steal()` occurs when the number of available goals is equivalent to the number of processors executing the program.

PAPI's worst performance occured executing a program that requires backtracking, while its best performance occured executing deterministic programs with deep proof trees and plenty of parallelism. It is suggested that if `steal()` is improved such that its overhead is reduced and work is distributed evenly, PAPI's overall performance would demonstrate a marked improvement.

CHAPTER 7

Conclusions and Future Research

7.1. Conclusions

The task of designing a parallel Prolog opcode-interpreter that exploits AND-parallelism, and implementing the opcode-interpreter on a shared-memory multiprocessor architecture was the focus of this thesis. Throughout PAPI's design, areas presenting a potential for high overhead were examined and in several situations, algorithms that avoid or reduce this overhead were incorporated. For example, variable-binding conflicts were discussed and a modified RAP algorithm to detect and avoid these conflicts was adopted. Backward execution, another area of high overhead, was examined and a semi-intelligent backtracking scheme was implemented. Finally, a means of communication among processors was incorporated into the opcode-interpreter.

The opcode-interpreter's performance was analyzed by executing a series of benchmark programs using the 1 to 15 processors available on the Balance 21000. This analysis suggested that parallel backward execution and `steal()` are the main areas of wasted time in the opcode-interpreter's execution. The forced sequential execution of processors in backward execution indicates that a logic programming language that does not require backward execution would derive more benefits from this system than Prolog does. In order to improve upon backward execution, it

may prove profitable to incorporate a centralized controller that backtracks for the entire shared-memory multiprocessor, rather than requiring each processor to backtrack for itself.

The execution time lost by *free* processors searching the proof tree for available goals to steal and waiting in spin-locks for other *free* processors searching for work, indicates that a centralized, rather than distributed scheme may be more appropriate for this application. A centralized work controller responsible for distributing work among the *free* processors would most likely reduce the overhead of searching and locking structures.

Further improvement for work distribution may result by surrendering the pure stack-based model of this implementation. By allocating new stacks for the goals stolen and maintaining the stack in segments rather than as a whole, processors would be freed of the strictness of the steal rule and work distribution overhead would be reduced as processors become eligible to steal any available goal. In the pure stack-based model, processors are only permitted to steal those available goals which maintain the stack property of their stacks.

In addition to the areas of high overhead discussed above, the opcode-interpreter's perfomance suffered as a result of variable dependencies in the goals of a Prolog clause. In order to avoid the excessive overhead that may occur from executing dependent goals in parallel, goals with dependent variables in a clause were executed sequentially, hence reducing the amount of parallelism derivable from that clause and the program. The reduction of parallelism resulting from avoiding variable-binding conflicts is illustrated in the *avail_par* curves in Chapter 6.

Therefore, given the areas of high overhead in this parallel Prolog opcode-interpreter and the limited amount of parallelism derivable from a Prolog program, due to AND-parallelism and variable dependencies, it is concluded that AND-parallelism alone is likely to yield only moderate degrees of parallelism, which would be best attained under a centralized work controller. These results of implementing a distributed work allocation scheme also indicate that implementing a similar scheme on a message-passing multiprocessor would result in excessive overhead.

7.2. Future Research

It is hoped that this research has made the reader aware of some of the basic issues and problems involved with implementing an AND-parallel Prolog interpreter on a multiprocessor architecture. Listed below are several areas in which PAPI may be improved and expanded.

Automatic Generation of Execution Graph Expressions: As mentioned in Chapter 3, the compiler may be updated to supplement the execution graph expressions provided by the programmer or generate all execution graph expressions for a Prolog program. This scheme would enable a programmer to supply some, none, or all execution graph expressions in the Prolog source code.

Centralized Pool of Work: Maintaining goals available for execution in a centralized location rather than on each processor's *availist* may aid in reducing the overhead associated with stealing goals. In the present implementation of `steal()`, high overhead results from *free* processors searching the proof tree for

available goals to steal, and *free* processors locking and unlocking *availists* of other processors. This proposed scheme would reduce the amount of time spent by *free* processors finding work and the frequency of locking structures if the number of processors executing the program were kept to an acceptable amount for this implementation of distributing work.

Non-Backtracking Languages: As revealed in Chapter 6, PAPI does not perform well during backward execution. As a result, it may prove beneficial to modify PAPI to support other logic programming languages that do not engage in backward execution.

Support OR-parallelism: PAPI may serve as a starting point for an AND-OR-parallelism system. Note that incorporating both AND-parallelism and OR-parallelism is traditionally a difficult task.

REFERENCES

[Bor84]
Borgwardt, P., Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors, *1984 International Symposium on Logic Programming, Atlantic City*, Silver Spring MD, February 1984.

[Bor86]
Borgwardt, P., Distributed Semi-Intelligent Backtracking for a Stack-based AND-parallel Prolog, *1986 International Symposium on Logic Programming, Salt Lake City*, Silver Spring MD, 1986.

[BrP81]
Bruynooghe, M. and L.M. Pereira, *Revision of Top-Down Logical Reasoning through Intelligent Backtracking*, Departement Computerwetenschappen, Katholieke Universiteit Leuven Belgium, 1981.

[CDD85]
Chang, J.H., A. Despain and D. DeGroot, AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis, *IEEE Compcon*, February 1985.

[ChD85]
Chang, J. H. and A. M. Despain, Semi-Intelligent Backtracking of Prolog Based on Static Dependency Analysis, *IEEE Symposium on Logic Programming*, July 1985.

[CiH84]
Ciepielewski, A. and S. Haridi, Control of Activities in the Or-Parallel Token Machine, *1984 International Symposium on Logic Programming, Atlantic City*, Silver Spring MD, February 1984, 49-58.

[ClM79]
Clark, K. L. and G. McCabe, The Control Facilities of IC-Prolog, in *Expert Systems in the Micro Electronic Age*, D. Michie (ed.), Edinburgh University Press, 1979.

[ClM84]
Clocksin, W.F. and C.S. Mellish, in *Programming in Prolog*, Springer-Verlag, New York, 1984.

[CoK81]
Conery, J. and D. Kibler, Parallel Interpretation of Logic Programs, *Proceedings of the Conference on Functional Languages and Computer Architecture*, October 1981, 163-170.

[Con83]
Conery, J.S., The AND/OR Process Model for Parallel Interpretation of Logic Programs, Technical Report 204, PhD Thesis, The University of California at Irvine, 1983.

[CoK85]
Conery, J. and D. Kibler, AND Parallelism and Nondeterminism in Logic Programs, *New Generation Computing 3*(1985), 43-70.

[CoP81]
Cox, P. and T. Pietrzykowski, Deduction Plans: A Basis for Intelligent Backtracking, *IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI 3*, 1981.

[DeG84]
DeGroot, D., Restricted AND-Parallelism, *Proceedings of the International Conference on Fifth Generation Computer Systems*, November 1984.

[DeG85]
DeGroot, D., Alternate Graph Expressions for Restricted AND-Parallelism, *IEEE Compcon*, February 1985.

[DKM84]
Dwork, C., P. C. Kanellakis and J. C. Mitchell, On the Sequential Nature of Unification, *The Journal of Logic Programming*, June 1984.

[EKM82]
Eisinger, N., S. Kasif and J. Minker, Logic Programming: A Parallel Approach, *Proceedings of the First International Programming Conference*, Marseille, September 1982, 1-8.

[EmL81]
Emden, M.H. van and G. J. de Lucena, Predicate Logic as a Language for Parallel Programming, in *Logic Programming*, K. L. Clark and S. A. Tarnlund (ed.), Academic Press, New York, 1981.

[Her86]
Hermenegildo, Manual V., *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, PhD Thesis, The University of Texas at Austin, 1986.

[KTM86]
Kahn, Kenneth, Eric D. Tribble, Mark S. Miller and Daniel G. Bobrow, Objects in Concurrent Logic Programming Languages, *OOPSLA 1986 Conference Proceedings 21*(November 1986), 242-257, ACM.

[Kow74]
Kowalski, R. A., Predicate Logic as a Programming Language, *Proc. IFIPS*, 1974.

[LiP84]
Lindstrom, G. and P. Panangden, Stream-Based Execution of Logic Programming, *1984 International Symposium on Logic Programming, Atlantic City*, Silver Spring MD, February 1984, 168-177.

[MaU86]
Mannila, H. and E. Ukkonen, On the Complexity of Unification Sequences, *Proceedings of the Third International Conference on Logic Programming*, 1986,

122-134.

[MaM82]
Martelli, A. and U. Montanari, An Efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems 4*(April 1982), 258-282.

[OGL85]
Overbeek, R. A., J. Gabriel, T. Lindholm and E.L. Lusk, Prolog on Multiprocessors, Technical Report, Argonne National Laboratory, Argonne Ill. 60439, 1985.

[PeP81]
Pereira, L. M. and A. Porto, An Interpreter of Logic Programs Using Selective Backtracking, Report 3-80, Departmento de Informatica , Universidade Nova de Lisboa, July 1981.

[Sha83]
Shapiro, E. Y., A Subset of Concurrent Prolog and Its Interpreter, *ICOT Technical Report Tech. Rep.-003*, Tokoyo Japan, 1983.

[TaF86]
Takeuchi, A. and K. Furukawa, Parallel Logic Programming Languages, *ICOT Technical Report*, Tokoyo Japan, 1986.

[TLJ84]
Taylor, S., A. Lowry, G. Q. McGuire Jr. and S. J. Stolfo, Logic Programming Using Parallel Associative Operations, *1984 International Symposium on Logic Programming, Atlantic City*, Silver Spring MD, February 1984, 12-21.

[TiW84]
Tick, E. and D. H. D. Warren, Towards a Pipelined Prolog Processor, *1984 International Symposium on Logic Programming, Atlantic City*, Silver Spring MD, February 1984, 29-42.

[War77]
Warren, D. H. D., Implementing Prolog-- compiling predicate logic programs, Dept. of AI Research Reports No. 39 & 40, University of Eidenburgh, May 1977.

[War83]
Warren, D. H. D., An abstract Prolog instruction set, Technical Note 309, Artifical Intelligence Center SRI, Menlo Park CA, October 1983.

[WAD84]
Warren, D. S., M. Ahamad, S. K. Debray and L. V. Kale, Executing Distributed Prolog Pograms on a Broadcast Network, *1984 International Symposium on Logic Programming, Atlantic City*, Silver Spring MD, February 1984, 58-68.

APPENDIX A

Opcode Instructions

The intermediate-code instructions executed by PAPI that are not included in D.H.D. Warren's instruction set [War77] are described below. These are the intermediate-code instructions for the execution graph expressions: `ipar`, `gpar`, `par`, `seq`, and `end`. As stated earlier, these instructions result from the addition of AND-parallelism to the sequential opcode-interpreter.

`ipar(A,label(end1),G)`

ARGUMENTS:

The `A` argument is an integer value specifying the number of arguments that `ipar` must check for dependencies, `label(end1)` is the address of the `end` that corresponds to the `ipar`, and `G` is an integer value for the number of goals that are within `ipar`'s scope.

USE:

The `ipar` intermediate-code instruction is used for the `ipar` execution graph expression in the Prolog source code. The `N` arguments are tested for dependencies during runtime. If dependencies exist among any of the `N` arguments, the corresponding `end` instruction at `L` is marked for sequential execution and the `G` goals are executed sequentially. Otherwise, the `end` instruction is marked for parallelism and the `G` goals are executed in parallel.

`gpar(A,label(end1),G)`

ARGUMENTS:

The `A` argument is an integer value specifying the number of arguments that `ipar` must check for dependencies, `label(end1)` is the address of the `end` that corresponds to the `gpar`, and `G` is an integer value for the number of goals that are within `gpar`'s scope.

USE:

The `gpar` intermediate-code instruction is used for the `gpar` execution graph expression in the Prolog source code. The `N` arguments' argument types are

checked during runtime. If any one type is not *ground*, the corresponding end instruction at L is marked for sequential execution and the G goals are executed sequentially. Otherwise, all N are type *ground*, the end instruction is marked for parallelism, and the G goals are executed in parallel.

```
par(label(end1),G)
```

ARGUMENTS:

Label(end1) is the address of the end that corresponds to the par and G is an integer value for the number of goals that are within par's scope.

USE:

The par intermediate-code instruction is used for the par execution graph expression in the Prolog source code. Parallel execution for the G goals is specified through this instruction. At runtime, the end instruction is marked for parallelism and the G goals are executed in parallel.

```
seq(label(end1))
```

ARGUMENTS:

Label(end1) is the address of the end that corresponds to the seq.

USE:

The seq intermediate-code instruction is used for the seq execution graph expression in the Prolog source code. The seq instruction is a no-op instruction since sequential execution of goals is the default. The corresponding end instruction is marked for sequential execution and the goals that follow this instruction are executed sequentially. It also serves as a parenthesis for sequential segments within a par, thus allowing the par to statement to correctly find the goals or sements of goals that will be executed in parallel.

```
end(G,label(end1),G)
```

ARGUMENTS:

Label(end1) is the address of the next end that follows this instruction. The label is anything if this is the last end instruction in the program. G is an integer value for the number of goals that are within the ipar, gpar, par, or seq's scope to which this end belongs.

USE:

The `end` instruction prevents execution from continuing until all G goals in the scope of the corresponding `ipar`, `gpar`, or `par` are finished. If the `end` instruction is marked for sequential execution, it is a no-op, but if marked for parallel execution, it makes processes `steal()` or `getwork()` before the next goal beyond the scope of parallelism may be executed.

APPENDIX B

Source, Intermediate-Code, and Opcode Files

The Prolog source code for the `fibonacci` program is given below.

```
query(X,Y):- fibo(4,Y).
fibo(0,1).
fibo(1,1).
fibo(X,Y):- X >= 2,
          gpar([X],
           seq(X1 is X-1,
               fibo(X1,Y1)
                   ),
           seq(X2 is X-2,
               fibo(X2,Y2)
                   )
            ),
            Y is Y1 + Y2.
```

The `fibonacci` program is compiled by the compiler which generates the intermediate-code instructions:

```
proc(query).
      enter(2).
      trylast(clause(1)).

clause(1).
      uvar(1,0).
      neck(1).
      call(2,fibo).
      int(4).
      local(0).
deflabel(outlabel).
      foot(2).

proc(fibo).
      enter(2).
      try(clause(1)).
      try(clause(2)).
      trylast(clause(3)).

clause(1).
      uint(0,0).
      uint(1,1).
      neckfoot(2).
clause(2).
      uint(0,1).
      uint(1,1).
      neckfoot(2).
clause(3).
      uvar(0,0).
      uvar(1,1).
      locinit(2,9).
      neck(9).
      call(2,>=).
      local(0).
      int(2).
      gpar(1,label(1),2).
      local(0).
      seq(label(2)).
      call(2,is).
      local(2).
      label(3).
```

```
        call(2,fibo).
        local(2).
        local(3).
deflabel(2).
        end(7,label(1)).
        seq(label(4)).
        call(2,is).
        local(4).
        label(5).
        call(2,fibo).
        local(4).
        local(5).
deflabel(4).
        end(8,label(1)).
deflabel(1).
        end(6,label(outlabel)).
        call(2,is).
        local(1).
        label(6).
deflabel(outlabel).
        foot(2).
deflabel(3).
        fn(-).
        ref(0).
        int(1).
deflabel(5).
        fn(-).
        ref(0).
        int(2).
deflabel(6).
        fn(+).
        ref(3).
        ref(5).
```

The intermediate-code instructions are then translated by the assembler into the opcode instructions that are interpreted by PAPI. The opcode instructions for the `fibonacci` program are:

```
Q 1
I 26 0 2
I 100 0 0
I 13 0 1
I 29 1 0
C 25 2 fibo
I 44 4 0
I 42 0 0
I 99 1 2

P fibo
I 28 2 0
I 26 0 3
I 26 0 5
I 27 0 7
I 7 0 0
I 7 1 1
I 34 2 0
I 7 0 1
I 7 1 1
I 34 2 0
I 1 0 0
I 1 1 1
I 13 2 9
I 29 9 0
I 75 2 5
I 42 0 0
I 44 2 0
M 61 1 18 2
I 42 0 0
I 63 0 7
I 75 2 7
I 42 2 0
I 47 0 18
C 25 2 fibo
I 42 2 0
I 42 3 0
I 64 7 9
I 63 0 7
I 75 2 7
```

```
I 42 4 0
I 47 0 13
C 25 2 fibo
I 42 4 0
I 42 5 0
I 64 8 1
I 64 6 4
I 75 2 7
I 42 1 0
I 47 0 8
I 32 2 0
B 46 2 -
I 48 0 0
I 44 1 0
B 46 2 -
I 48 0 0
I 44 2 0
B 46 2 +
I 48 3 0
I 48 5 0
```

APPENDIX C

Source Code for Benchmark Tests

C.1. Mapcolor

The mapcolor program solves a map-coloring problem for a map consisting of five regions as discussed in Chapter 2.

```
query(V, W, X, Y, Z).

query(A, B, C, D, E):- mapcolor(A, B, C, D, E).

mapcolor(A, B, C, D, E):- gpar([A, B, C, D],
                              next(A, B),
                              next(C, D)
                             ),
                          par(
                              next(B, C),
                              next(A, C),
                              next(A, D),
                              seq(
                                  next(B, E),
                                  par(
                                      next(C, E),
                                      next(D, E)
                                     )
                                 )
                             ).

next(red, blue).
next(blue, red).
next(yellow, red).
next(red, yellow).
next(blue, yellow).
next(yellow, blue).
```

C.2. Fibonacci

The `fibonacci` program determines and returns the number in the 15th position of the Fibonacci sequence.

```
query(Y):- fibo(15, Y).

fibo(0, 1).
fibo(1, 1).
fibo(X, Y):- X >= 2,
             par(
                 seq(
                     X1 is X-1,
                     fibo(X1, Y1)
                    ),
                 seq(
                     X2 is X-2,
                     fibo(X2, Y2)
                    )
                ),
             Y is Y1 + Y2.
```

C.3. Quicksort

The `quicksort` program sorts the list of 100 elements, the first argument in `sort`, and returns the sorted 100 element list as `B`, `sort`'s second argument.

```
query(B):- sort([79,59,34,10,98,61,49,20,67,16,99,77,12,
            50,0,80,41,30,8,68,3,78,24,52,1,48,91,
            71,25,6,23,51,89,66,15,96,43,76,95,69,
            39,40,63,81,11,55,45,88,18,9,70,93,38,2,
            60,58,82,62,42,26,75,36,5,29,17,32,85,74,
            4,73,53,87,44,7,94,13,35,97,14,21,64,54,
            83,27,90,46,31,57,19,33,28,86,47,84,22,72,
            65,37,92,56],B).

sort([H|T], S):- split(H, T, U1, U2),
                 par(
                     sort(U1, V1),
                     sort(U2, V2)
                    ),
                 append(V1, [H|V2], S).
sort([], []).

append([], L, L).
append([H|T], L, [H|U]):- append(T, L, U).

split(H, [H1|T1], [H1|U1], U2):- H1=<H,
                                 split(H, T1, U1, U2).
split(H, [H1|T1], U1, [H1|U2]):- H1>H,
                                 split(H, T1, U1, U2).
split(D, [], [], []).
```

C.4. Partiming

The partiming programs below were used by Hermenegildo as "efficiency" benchmarks because the amount of parallelism available in these programs is fixed and is known "a priori". The clause p(X) is a loop that is executed as many times as times(X)'s argument indicates and does not create new goals to be executed in parallel. Once a p(X) goal is stolen, the stealing processor finishes the goal and looks for work. Only the p(X) goals in the query clause are available for parallel execution throughout the entire program.

```
partiming2
query:- times2(X), gpar([X],
                        p(X), p(X)
                       ).

p(0).
p(X):- Y is (X-1), p(Y).

times2(64).


partiming4
query:- times4(X), gpar([X],
                        p(X), p(X), p(X), p(X)
                       ).

p(0).
p(X):- Y is (X-1), p(Y).

times4(32).
```

```
partiming8
query:- times8(X),gpar([X],
                       p(X), p(X), p(X), p(X),
                       p(X), p(X), p(X), p(X)
                      ).

p(0).
p(X):- Y is (X-1), p(Y).

times8(16).


partiming16
query:- times16(X),gpar([X],
                        p(X), p(X), p(X), p(X),
                        p(X), p(X), p(X), p(X),
                        p(X), p(X), p(X), p(X),
                        p(X), p(X), p(X), p(X)
                       ).

p(0).
p(X):- Y is (X-1), p(Y).

times16(8).
```

C.5. Deriv

The `deriv` program is another of Hermenegildo's benchmarks. This program calculates the derivative of the specified expression, `X`, defined in `expression`, with respect to `x` and returns the result as `Y`. The `d()` clauses are simple definitions of symbolic derivation, the `expression` clause contains the expression to be evaluated in a simplified form, and the `value` clause is the expanded version of the components in the `expression` clause. The expression is presented in two clauses for simplicity.

```
query:- expression(X), d(X, x, Y).

d(U+V, X, DU+DV):- gpar([X], d(U, X, DU), d(V, X, DV)).
d(U-V, X, DU-DV):- gpar([X], d(U, X, DU),d(V, X, DV)).
d(U*V, X, DU*V+U*DV):- gpar([X], d(U, X, DU), d(V, X, DV)).
d(U/V, X, (DU*V-U*DV)/pow(V, 2)) :- gpar([X], d(U, X, DU),
                                    d(V, X, DV)).
d(pow(U, N), X, DU*N*pow(U, N1)):- integer(N),
                          N1 is N - 1,d(U, X, DU).
d(-U, X, -DU):- d(U, X, DU).
d(exp(U), X, exp(U)*DU):- d(U, X, DU).
d(log(U), X, DU/U):- d(U, X, DU).
d(X, X, 1).
d(C, X, 0).

value(((3*x + (4*exp(pow(x, 3))*log(pow(x, 2)) - 2)) /
      ( -(3*x) + 5/(exp(pow(x, 4))+2)))).

expression( Exp + Exp - Exp*Exp / Exp*Exp / Exp):- value(Exp).
```

APPENDIX D

Calculation of Avail_par Curves

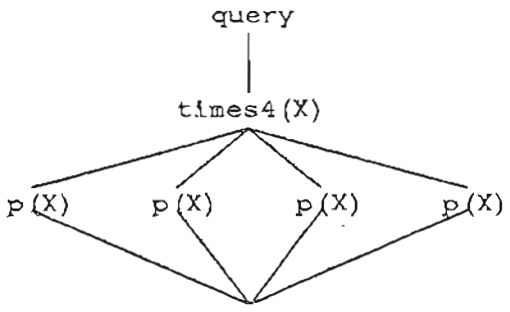
As discussed in Chapter 3, specific goals in a clause may require sequential execution due to dependencies among their variables. This sequential execution reduces the amount of parallelism that can be derived from the program. Hence, in order to make a fair estimate of PAPI's expected performance as more processors are added to the execution of a benchmark program, the *avail_par* curve is created. The calculation of the *avail_par* curve is by no means scientific or rigid. Rather, it is an estimate of the optimal speedup that PAPI can derive from a program, based on an educated evaluation of the amount of parallelism available in the program. The following statements are assumed while calculating the *avail_par* curve for a program:

1) all processors work (if work is available)
2) work is distributed equally among all processors
3) processors that finish their work steal more
4) idle processors find work quickly
5) stealing goals takes no time
6) programs do not backtrack
7) clause heads take roughly equal time to unify with a goal

Reference to Appendix C, which contains the source code for each of the benchmark programs, will greatly facilitate the reader's understanding of the remaining discussion.

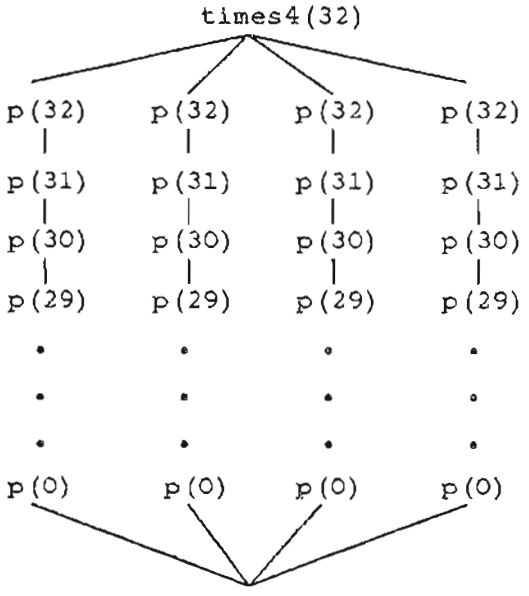
D.1. Example Calculation of an Avail_par Curve

Calculation of a program's *avail_par* curve begins with the creation of the program's *essential* execution graph. An essential execution graph (EEG) is an execution graph stripped of its "insignificant" goals and all remaining goals' variables are instantiated. Only goals requiring a significant amount of computation (i.e., `fibonacci(14)` in the `fibonacci(15)` program) compose the EEG. For example, the execution graph for the benchmark `partiming4` is:



`Partiming4`'s Execution Graph
Figure D.1

which becomes:



Partiming4's EEG
Figure D.2

Once created, the EEG is examined and analyzed to estimate the number of goals that would be executed by each processor if from 1 to 15 processors executed the program under the assumptions made above. Since `partiming4` is a relatively small program, the number of significant goals in the program is easily determined from Figure D.2 to be 133.

Sequential execution of `partiming4` would result in one processor executing all 133 goals one after another, creating a "goal-string" of 133 goals. (The goal-string measures the *maximum* number of goals executed by one processor.) Two

processors executing partiming4 would split the goals such that proc0 executed the partiming4(32) goal and two of the p(32) streams, while proc1 executed the other two p(32) streams. Thus, the goal-string for two processors is 67 goals (33 + 33 + 1). The addition of a third processor, proc2, would result in: proc0 executing partiming4(32) and one p(32) stream; proc1 executing one p(32) stream; proc2 executing one p(32) stream; and the first process to finish steals and executes the remaining p(32) goal stream. Hence, one processor must execute two p(32) streams, creating a goal-string of 66 goals for three processors. If four processors execute partiming4, one processor will execute the partiming4(32) goal, and each of the four processors will execute one p(32) stream. Thus, the goal-string for four processors is 34 goals.

As more processors are added to the execution of partiming4, the goal-string remains the same. Due to the sequentiality of the p(32) streams, partiming4 has enough work for 4 processors only, all additional processors will remain idle throughout execution.

After the goal-string value for 1 to 15 processors is established, the most speedup that the program offers is determined from these values. The speedup is calculated using goal-string values in place of timing values and plotted as the *avail_par* curve. See Figure 6.2 for partiming4's *avail_par* curve. The *avail_par* curves for the other benchmark programs are calculated similarly.

D.2. Avail_par Curve Calculations

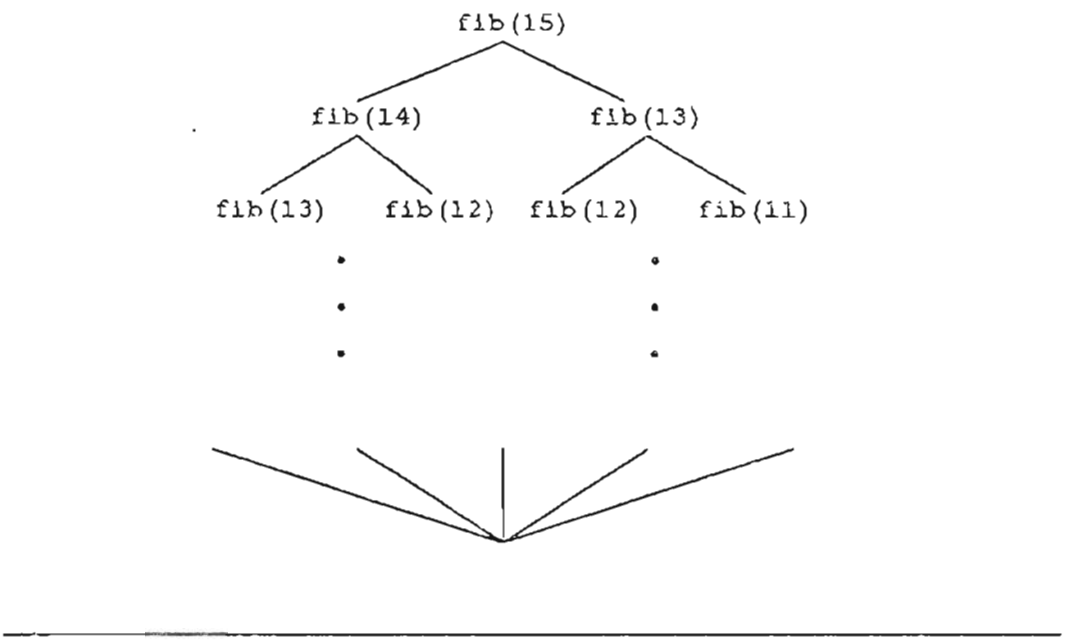
Below are the calculations of the *avail_par* curves for the benchmark programs.

D.2.1. Mapcolor

Due to its backward execution, the calculation of the EEG and *avail_par* curve for `mapcolor` is very difficult. Processors cannot steal during backward execution and processors must often wait for other processors before continuing their own execution. Thus, synchronization of processors must be considered, which is beyond the *avail_par* curve calculation scheme devised here. Therefore, an *avail_par* curve is not created for `mapcolor`.

D.2.2. Fibonacci

`Fibonacci`'s execution graph is stripped and `fibonacci` is shortened to `fib` to arrive at the EEG below:



fibonacci's EEG
Figure D.3

The exact number of goals in fibonacci's EEG is approximately 3 * fib(N). Due to the sequential segments in the fibonacci program, the *avail_par* curve may be formed using just fib(N). That is, the number of goals in fibonacci's EEG is 987, the value of fib(15). The same is true for all fib() goals in the EEG, fib(14) is 610, fib(13) is 377, and so on. Thus, the goal-string for one processor is 987. Execution with two processors would begin with proc0 would execute the goal, fib(15), and the tree of fib(14), while proc1 would execute the tree of fib(14). Thus, proc0 executes 1 + 610 goals and proc1 executes 377 goals. But, proc1 would finish its execution before proc0 and, due to the assumptions made,

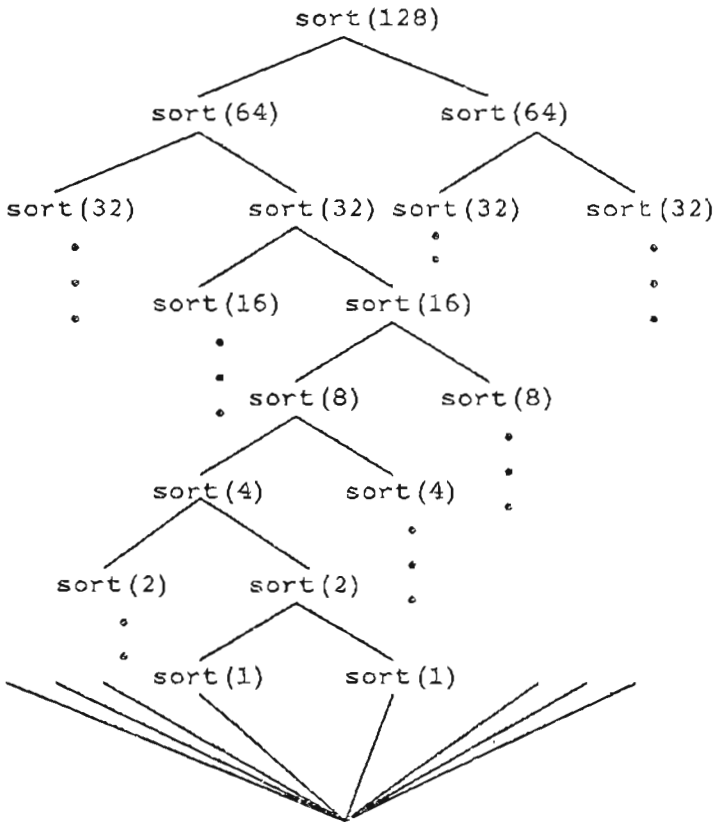
steal some of proc0's goals such that the number of goals executed by each processor would most likely split evenly. As a result, the goal-string for two processors would be close to 500.

As more processors are added to `fibonacci's` execution, work is expected to be divided evenly among the processors and, due to the large amount of parallelism in the `fibonacci` program, the *avail_par* curve is expected to have a slope of 1, as illustrated in Figure 6.6.

D.2.3. Quicksort

For `quicksort`, the `split` and `append` goals generate a sequential list of goals equal to the length of the list to be sorted. Thus, by symmetry, only the lengths of the lists need to be considered when determining the number of goals in `quicksort`'s EEG.

`Quicksort`'s EEG in Figure D.4 contains the `sort` goals with the size of the list that will be sorted by the goal as its argument, rather than the actual list. It is assumed in this discussion that the `split` clause splits its argument list evenly and returns two lists one half the length of the argument list. In addition, the execution of each `sort(LENGTH)` goal creates a goal-string of `LENGTH` since the sort consists of `LENGTH` calls to `sort()`. Goal strings are calculated for the execution of `quicksort` by 1, 2, 4, 8, and 16 processors, as these are the natural breaks in the program's EEG. Later, the speedup based on goal-strings will be interpolated for the missing goal-string values.



Quicksort's EEG
Figure D.4

Sequential execution of `quicksort` creates a goal-string of 1024 goals (8 * 128 for the eight levels in the EEG that sort 128 goals on each level). Execution by two processors would split the EEG goals such that proc0 executes `sort(128)` and the `sort(64)` subtree, while proc1 executes the other `sort(64)` subtree. The goal-string for two processors has 576 goals (128 + 7 * 64). Four processors would

execute the EEG such that: proc0 executes goal `sort(128), sort(64),` and the `sort(32)` tree; proc1 executes the `sort(32)` tree next to that executed by proc0; and proc2 executes the remaining `sort(64)` tree. Thus, the goal-string for four processors has 384 goals (128 + 64 + 6*32). The EEG is further analyzed for the addition of processors until goal-string values are determined for execution of `quicksort` with 1, 2, 4, 8, and 16 processors. From these values, the speedup is calculated and graphed and speedup for 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, and 15 processors is interpolated, resulting in the *avail_par* curve in Figure 6.7 (which is essentially a logarithamic curve).

D.2.4. Partiming

The `partiming` programs are easier to evaluate since they are the smallest of the benchmark programs. In each of these programs, the amount of parallelism available is large, but due to the loop of sequential execution in the `p(X)` clause, speedup is not always linear. For example, `partiming16` has 16 calls to `p(X)` and `p(X)` is a sequential loop of 8 calls to itself. If this benchmark is executed by 8 to 15 processors, the best speedup that can be obtained is eight. That is, all 8 to 15 processors will execute one `p(X)` loop in parallel, but there are still 1 to 8 branches of `p(X)` left to be executed after the processors finish. Since these are sequential segments, processors must finish their own segments before starting another. As a result, at least one processor must execute two sequential `p(X)` segments, creating a goal-string one eighth of that created by one processor executing `partiming16`. Hence, the best speedup possible for up to 15 processors is 8, and the best speedup

for 16 processors is 16, since each processor executes only one p(X) segment.

The evaluation of the *avail_par* curves for the partiming2, partiming4, partiming8, and partiming16 benchmarks follow the same pattern presented in partiming4's *avail_par* calculation example above. Due to the simplicity of these benchmarks, the EEGs and evaluation of the *avail_par* curves is not discussed here, but the reader is referenced to the graphs in Figures 6.4 through 6.8.

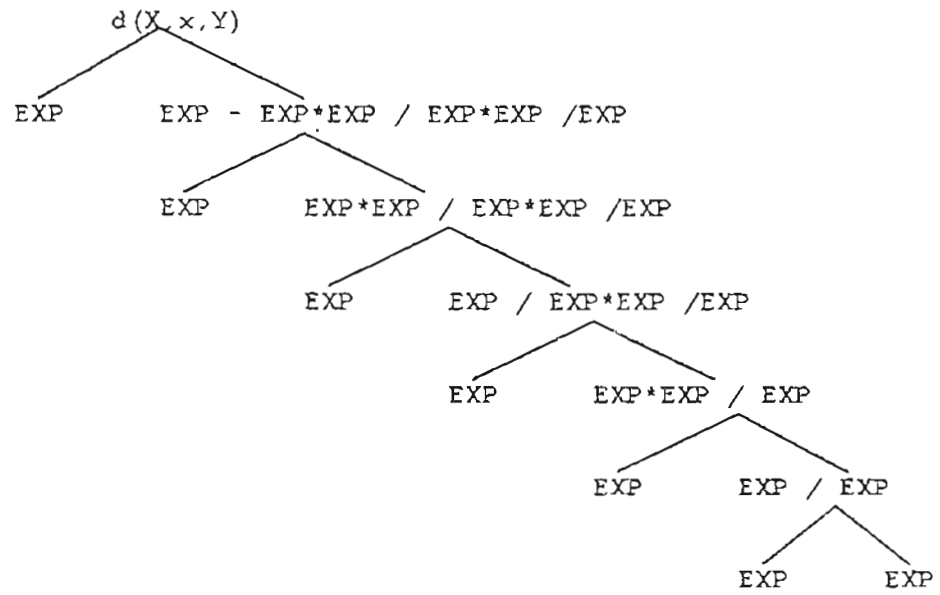
D.2.5. Deriv

An estimate of the amount of parallelism in the deriv benchmark is difficult to determine, due to the large expression evaluated in the query. As a result, this is most likely the least accurate of the *avail_par* curves.

Deriv begins evaluation of the large expression by examining the expression in terms of its larger components:

```
EXP + EXP - EXP*EXP / EXP*EXP / EXP
```

Thus, the EEG for Deriv is:



Deriv's EEG
Figure D.5

where each EXP represents the large term in the fact `value`. After the overall expression is broken and distributed, the individual EXP terms are evaluated. Thus, the `deriv` of EXP appears to be rich in parallelism.

Based on the EEG in Figure D.5 and the size of the terms, it is estimated that there may be enough work to keep up to 7 processors busy at all times. Further examination of the `deriv` of EXP indicates that there is plenty of work to share among 15 processors at all times. Thus, a linear curve is presented for `deriv`'s *avail_par* curve in Figure 6.8.

BIOGRAPHICAL NOTE

The author was born 27 December 1961 in Santa Monica, California. In 1967 she moved to New Haven, Connecticut and 2 years later, to Walnut Creek, California where she graduated from Northgate High School in June 1979.

The August following graduation, the author began her studies at Reed College in Portland, Oregon. She attended the University of Uppsala in Uppsala, Sweden during the period from July 1981 to August 1982. The author returned to Reed College where in May of 1984, she received a Bachelor of Arts Degree in Physics. During her four years at Reed, she was awarded two Commendation for Excellence Awards.

In September 1984, the author began her studies at Oregon Graduate Center where she completed the requirements for the degree of Master of Science in August 1987.

After completing her degree at Oregon Graduate Center, the author began her employment at Floating Point Systems Inc. in Beaverton, Oregon as an Engineering Scientist. Her interests are primarily in the areas of logic programming, parallel programming, artificial intelligence, and knowledge engineering.