Report 85-18
Stanford -- KSL

Scientific DataLink

PM: A Parallel Execution Model for Backward-Chaining Deductions.

Vineet Singh, Michael R. Genesereth,

Aug 1985

KSL Report
KSL-85-18

First version: May 1985
Last revision: August 1985

PM: A Parallel Execution Model for Backward-Chaining Deductions

Vineet Singh
Michael R. Genesereth

Logic Group
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford, California 94305
(415) 497-4368
Arpanet: vsingh@sumex-aim.arpa

Abstract

This paper describes *PM*, an execution model for automating backward-chaining deductions on multiple processors. The term *execution model* refers to the state, messages and procedures required to perform the computation *correctly*. The target multiprocessor is characterized by (1) a large number of small processors, (2) inter-processor communication via messages, and (3) a distributed database. The key distinguishing feature of *PM* is simultaneous exploitation of *and-parallelism*, *or-parallelism* and *pipelining* in this scenario.

Table of Contents

List of Figures

BLANK PAGE

1. Introduction

Future reasoning machines — the fifth generation of computer architectures — will exploit parallelism to obtain dramatic increases in speed [6]. Several difficult problems need to be solved to make that happen, however. One such problem, and the one addressed in this paper, is the design of execution models, or interpreters, that allow desirable kinds of parallelism to be exploited for certain types of computations. Execution models designed for sequential Von Neumann computer architectures are usually not appropriate for multiprocessors.

Execution models for multiprocessors need more than the specification of the state and procedures required to specify sequential execution models. The interaction between processors must be described as well. A complete definition of a parallel execution model includes the description of the state to be maintained, the messages to be used to interact with other processors, and the procedures to manipulate the state and send/receive messages.

The type of computation being considered in this paper is backward-chaining deduction [2]. This is the kind of deduction employed in most extant Logic Programming Languages. Prolog is a prime example. Side-effects are not allowed by the current definition of *PM*, the execution model presented here.

Logical deduction is particularly attractive as a starting point for exploiting parallelism because (1) It has a well understood semantics that is completely independent of any computer architecture, be it sequential or parallel, and (2) All the exploitable parallelism is completely implicit and directly available. It is not necessary for the programmer to be burdened with explicitly specifying the parallelism or to use complex techniques to uncover the parallelism. These two advantages lead to a third powerful advantage. The programmer can program *approximately* as he would with a sequential computer. We say *approximately* because optional *pragmas* may be given by the programmer to increase the efficiency of parallel execution. This is analogous to explicit storage allocation for some storage to preempt the regular garbage collector.

The multiple processor scenario that *PM* is designed for has the following characteristics. There is a large number of small processors. The processors communicate solely by message-passing; each processor has some local memory but there is no shared memory. In addition, the database of facts is distributed over the processors. Parts of the database may be replicated if that is beneficial for some reason. No assumption is made about the size of the individual processor memories except that the entire database must fit in the collection of memories of the processors in the system. Similarly, no assumption is made about the speed of the processors. Moreover, it is assumed that each processor

can do backward-chaining deductions.

Several important kinds of parallelism have been identified for backward-chaining deductions [3, 8]. The three that are exploited by *PM* are *and-parallelism*, *or-parallelism*, and *pipelining*. Unrestricted *and-parallelism* is usually not exploited because of its wasteful, combinatoric explosion. Various researchers have considered different methods of restricting *and-parallelism* [8, 3, 4, 10, 5]. The *and-parallelism* exploited by *PM* is of the type described by Conery [3]. Future mention of *and-parallelism* in this paper will refer to this restricted type unless mentioned otherwise. A previous execution model has exploited a combination of *or-parallelism* and *and-parallelism* [3]. Another combination that has been used is *or-parallelism* and *pipelining* [10, 5]. *PM* is unique in exploiting all three at once.

What *PM* does not include is resource allocation techniques. Resource allocation will determine (1) the actual distribution of the database over the processors and (2) the actual processor to use in the case of replication of certain parts. Clearly, this will strongly affect the efficiency of backward-chaining deductions. These issues are certainly important but they are not the subject of this paper. Resource allocation in the context of *PM* will be examined in a later paper.

This paper is organized as follows. First, the general approach towards exploiting parallelism is described in section 2. Next, *PM*, the parallel execution advocated by this paper is described in section 3. At this point, a few simplifications are used to convey the basic idea of the execution model without cluttering it up with details. Next, the extensions required to handle the extra complexities not handled in the basic execution model are described in section 4. Finally, section 5 discusses the larger body of the authors' research in the context of which *PM* was developed. This final section also discusses some related work done by others and presents the conclusions.

2. The Approach

2.1. Viewing Backward-Chaining Deductions as And-Or Tree Computations

Backward-chaining [2] is an inference mechanism for automated deduction. It is used here in the context of a database of horn-clauses. An example of a horn clause is

H :— T1,T2,...,Tn

where H and all Ti are atomic propositions involving predicates and arguments which are terms. A term is a constant, a variable or a function of a term. The meaning of the above horn clause is

H is true if all of T1,T2,...,Tn are true.

H is the called the head of the horn clause and T1,T2,...,Tn the body of the clause. Also, by definition, the clause is an assertion if n is zero and is a rule if $n > 0$.

An and-or tree is a problem reduction representation [1] and this can be used to represent the problem of proving a goal by backward-chaining as well. A goal is an arbitrary conjunct of atomic propositions. For example, G1,G2,...,Gm is a goal.

Figure 2-1 shows an example of a syntactic and-or tree used to represent a backward-chaining deduction. In this figure, square nodes denote or-nodes and octagonal nodes denote and-nodes. And-nodes are called so because the goal they represent is one conjunct in a conjunctive goal set. Similarly, or-nodes represent a disjunct in a disjunctive goal set. Nilsson [7] gives a more formal characterization of and-or trees. Arcs are marked with the number of the clause used for the reduction. Also, a cut through the arcs going from a node to its children indicates that the children are and-nodes. The leaf nodes cannot be reduced any more. Leaf nodes may be empty square boxes. These denote empty goals or successes. All leaf nodes in the example are empty goals.

We call the tree syntactic because certain subtrees may be instantiated multiple times during an actual execution. For example, if conjuncts are solved left to right, multiple solutions to "p(X)" will lead to as many instantiations of the subtree rooted at "q(Y)".

Some of the leaf nodes in the and-or tree end in failure and others end in success. The purpose of the computation is to find one or all nodes associated with success in the and-or tree. Therefore, the computation is essentially a search problem.

2.2. Sequential Interpretation

The most widely used sequential interpretation is perhaps the one used by Prolog. The search through the tree is a depth-first, left-to-right search. When the first answer is obtained, it is announced. If more solutions are demanded, the search continues.

2.3. Parallel Interpretation

Many different parallel interpretations of the and-or tree are possible. One could, of course, do everything in parallel. All or-nodes that are the children of an and-node can be solved in parallel (or-parallelism) and all and-nodes that are the children of an or-node can be solved in parallel (and-parallelism). For and-parallelism, this would mean running a process for each of the atomic

Top level goal		r(X,Y)
Database	C1	r(X,Y) :- p(X),q(Y),s(X,Y)
	C2	p(a)
	C3	p(b)
	C4	q(Y) :- m(X),n(X,Y)
	C5	m(a)
	C6	m(b)
	C7	n(b,a)
	C8	n(b,b)
	C9	s(a,b)
	C10	s(b,a)

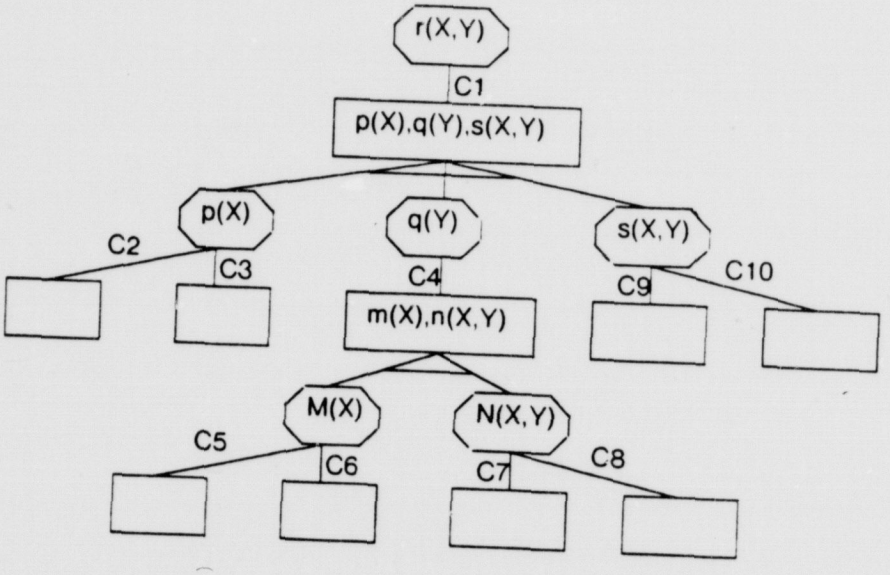


Figure 2-1: A syntactic and-or tree

propositions in the conjunct in parallel. This would generate many solutions, most of which might fail if there are shared variables in the conjunct that must be simultaneously satisfied. Therefore, in general, it is a good idea to avoid this highly combinatoric explosion.

The solution adopted here is to exploit all the *or-parallelism* but to take a more conservative position with respect to *and-parallelism*. Only those and-nodes are solved in parallel that do not share any common variables. Assume for now that an and-node solution binds all the variables in it to ground terms (i.e., terms with no variables). Once and-nodes bind certain variables, other and-nodes

may then not share variables and those can then be solved in parallel. One can think of the and-nodes as arranged in a directed, acyclic graph (DAG). Solutions from nodes flow to their downstream neighbors which can then be solved in parallel. Notice that each application of a rule in the database produces one such DAG. There is a one to one correspondence between the atomic propositions in the body of the rule and the nodes in the DAG. An example is shown in figure 2-2.

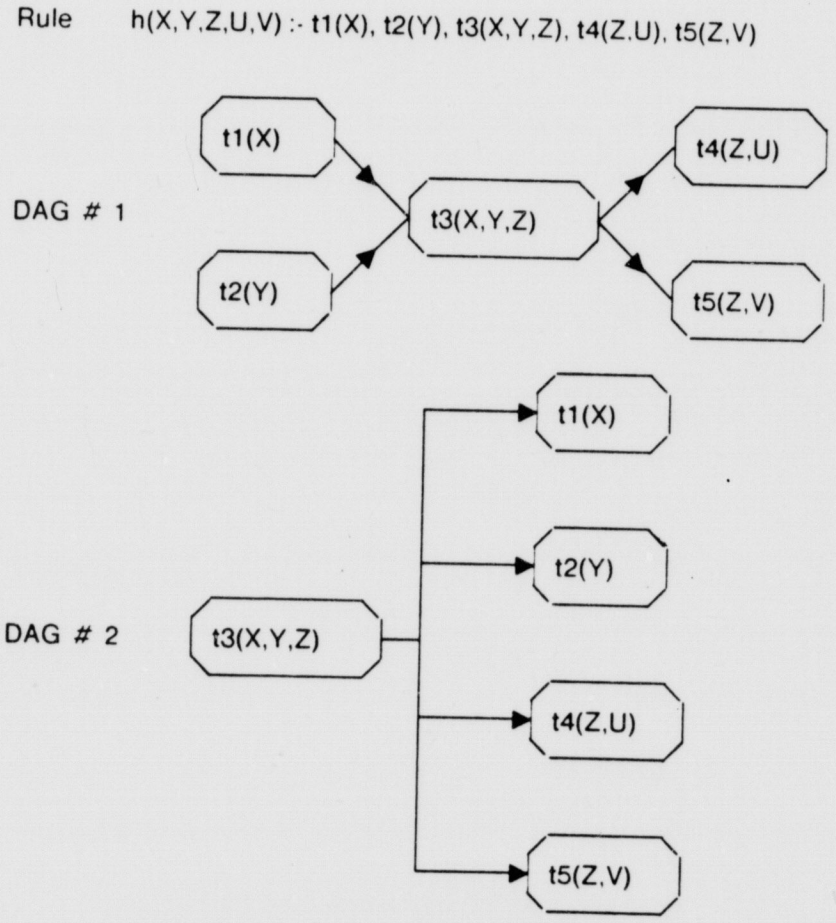


Figure 2-2: Example dataflow* graphs

The figure also illustrates that more than one DAG may be possible for a rule application. In general, some of these DAGs will be solved more efficiently than others. In fact, this problem is analogous to ordering conjuncts for efficient sequential interpretation [12]. This problem is important but is not the subject of this paper. For the purposes of this paper, we will assume that a suitable DAG gets chosen based on some efficiency considerations when a rule is actually applied. Also, note that the DAG that is chosen is, in general, different when different sets of variables get bound at rule application time. Section 4.2 has more to say about this.

This is also the kind of restricted *and-parallelism* chosen by Conery [3]. The difference here is that *pipelining* is exploited as well. *Pipelining* means passing solutions from and-nodes to downstream neighbors in a continuous stream. Conery's model does not exploit *pipelining* because only one solution of an and-node is passed to a downstream neighbor at a time. More solutions may be sent downstream because of "failures" but only at most one for each failure.

3. The Basic Execution Model

The basic execution model deals with a simplified view of the multiprocessor environment as well as of backward-chaining. The additional complexities are handled in the extensions to the basic execution model. The reason for presenting only the basic execution model first is that the basic execution model captures the essential ideas that are the basis of *PM*. Also, explaining the extensions right in the beginning would make it harder to get the most important ideas across.

The simplifications are the following: (1) It is assumed that the set of clauses pertinent to reducing any particular goal are in a single processor. For example, if facts are partitioned on the basis of predicate symbols, all facts with a certain predicate symbol are in a single processor. (2) It is assumed that once the database is distributed over the multiple processors, there is no shortage of temporary storage at individual processors during the computation. (3) Finally, it is assumed that all solutions to a goal bind all the variables in the goal to ground terms (i.e., terms not containing any variables).

3.1. Overall Description

The key concepts in the basic execution model are: processes, tasks, and dataflow* graphs. Intuitive descriptions appear in this section. Later sections present more detail.

A process is associated with a single atomic proposition. Its purpose is to solve the associated proposition when invoked.

A task is a 2-terminal DAG of processes and some bindings of the associated variables. The 2 terminals are special nodes called input node and output node. The DAG specifies the partial order in which the constituent atomic propositions must be solved. In general, one can think of a task DAG as a dataflow graph. For each solution produced by any node, the set of bindings is sent in a token to each of the downstream nodes. If there are multiple streams going to a node, the node must consider the cartesian product of the streams. This is obtained by finding all combinations of these tokens such that each combination has exactly one token from each of the streams. Each combination can

be considered to be one bigger virtual token by taking the union of the bindings in the individual tokens. The purpose of a process may now be restated more precisely to be to find all solutions of the associated atomic proposition for each virtual token sent to it. In general, a node in the task DAG may be considered to be the generator of all the variables in the associated atomic proposition minus the variables whose bindings it receives in the input virtual tokens. Finally, the tokens in the output stream from the output node of a task DAG correspond to solutions of the conjunct of the associated atomic propositions.

Subtasks (or child tasks) are created in the following way. To solve a process with a certain set of bindings corresponding to a virtual token, one uses all possible rules and assertions in the database that can be used for backward-chaining. We will assume that the processor containing the process also contains all possible clauses that might be applicable for backward-chaining from the associated proposition. Assertions may lead to solutions right away. For rules, each rule that can be applied (after checking for unification), creates a subtask corresponding to the body of the rule. This subtask will be a DAG of processes with one node for each atomic proposition in the body of the rule. In general, the DAG need not be a 2-terminal DAG. A dummy node is added to the input and the output to convert it to the standard 2-terminal DAG format. Solutions of these or-parallel subtasks are sent to the parent task in streams.

The overall picture is the following. One starts with a top level task — a 2-terminal DAG of processes. Processes create or-parallel subtasks to solve the associated atomic propositions for all input virtual tokens. This process continues recursively till there are no more rules to apply or enough solutions have been found for the top-level goal.

Finally, a dataflow* graph is simply the graph that is generated in the process of solving a top level goal. The graph is a dataflow graph in the conventional sense except for two non-dataflow features. First, it includes indeterminate merges of solutions from subtasks at parent tasks. Second, taking cartesian products of streams requires state to be maintained at the nodes. This is inconsistent with traditional dataflow graphs. One might note at this point that to be able to reclaim storage associated with streams whose cartesian product is being computed, it is necessary to know when the streams have terminated. Some kind of end of stream indicator is required. It will be seen in the detailed description that other state is kept as well at the nodes in the dataflow* graph. However, the other state is not essential to the basic idea. It is kept only to reduce communication cost.

A syntactic dataflow* graph is one where parts of the graph may, in an actual interpretation, be instantiated more than once. For example, each virtual token at a process creates an instantiation of

the subtask DAG associated with each applicable rule. The syntactic dataflow* graph represents all subtasks created by the application of a particular rule at a process with one generic task DAG. Just as a syntactic and-or tree is useful to get a static view of the decomposition of tasks, a syntactic dataflow* graph is useful to get a static view of the parallel execution of tasks.

In the detailed description that follows, it will be seen that one, in fact, does not just send bindings in the solution streams. One actually sends code pointers as well. As it turns out, the tokens are full-fledged tasks. However, the general idea is as described in this section.

The following notation is used for detailed description: <A,B,...,N> is used to denote a tuple with the fields A,B,...,N. {A,B,...,N} is used to denote a set with the elements A,B,...,N.

3.2. State

Each processor, process and task has a system-wide unique name. Typically, the names are of the form P_i, PS_i, and T_i respectively.

Each processor maintains the following state information on the tasks and processes for which it is responsible:

Work-Set: This is a set of tasks that the processor may work on.

Task: Each task is a 6-tuple of the form:

<*Task-Name, Task-Description, Subtasks, Spawning-Process-Name, Parent-Task-Name, Invocation-Binding*>

Task-Name is the system-wide unique name of the task. *Task-Description* is the description of the task (more later). *Subtasks* is a set of child tasks, if any. *Spawning-Process-Name* is the name of the process that spawned the task. *Parent-Task-Name* is the name of the parent task, if any. *Invocation-Binding* is only for tasks that have parents. It is a binding list that contains the bindings of the variables in the parent goal when it was reduced. Bindings to the child task are made at invocation time. Also, variables in the child task that are not bound are given system-wide unique names.

Task-Description: Each is a tuple of the form:

<*CG, BL*>

CG (or Conjunct Graph) is a 2-terminal DAG. *BL* is a binding list. The 2-terminal DAG has two special terminal nodes called input node and output node. The nodes in the graph are processes (as specified below).

Process: Each is a 9-tuple of the form:

<Process-Name, Literal, Processor-Name, Number-Inputs, Input-Queues, Outputs, Spawned-Task-Names, Type, Child-Task-Name>

Process-Name is the system-wide unique name of the process. *Literal* is the literal that the process is responsible for solving. *Processor-Name* is the name of the processor where the process resides. *Number-Inputs* is the number of inputs to the process. *Input-Queues* are the queues of messages waiting to be processed at the inputs to the process. These queues contain additional state to (1) indicate whether the end of stream marker has been received and (2) give the status of the cartesian product formation from the inputs. *Outputs* are a set of tuples specifying the inputs of other processes. Each tuple is of the form *<process-name, input-number>*. *Spawned-Task-Names* is a set of task names. The names correspond to tasks that are created by cartesian product. In case no cartesian product is necessary (when there is only one input), the unmodified task names from the inputs are directly included in *Spawned-Task-Names*. The *Type* of the process can be one of {Normal, Head, Tail}. Head and Tail processes correspond to input nodes and output nodes in child tasks. Both Head and Tail processes have null *Literal* fields. A Head process has no inputs and a Tail process has no outputs. Finally, a Tail process is the only kind of process that has a non-null *Child-Task-Name* field. This is the name of the child task whose output node is the Tail process.

A complete process specification as given above is not necessary for each node in the conjunct graph of a task specification. A partial specification as given below is sufficient. " - - - " indicates an unspecified field.

<Process-Name, Literal, Processor-Name, Number-Inputs, - - -, Outputs, - - -, Type, - - ->. The unspecified fields are *Input-Queues*, *Spawned-Task-Names* and *Child-Task-Name* from left to right.

3.3. Messages

Messages are 4-tuples of the form:

<*Message-Type, Source-Processor-Name, Destination-Processor-Name, Arguments*>

For now, only one message type is required. More types are required for the extensions to the basic execution model. The type needed now is *Input-Task*. For this, the *Arguments* field is a tuple of the form:

<*Destination-Process-Name, Destination-Input-Number, Task-Name, Task-Description*>

The fields have self-explanatory names. End of stream is indicated with "EOS" as the binding list in the *Task-Description*.

3.4. Procedures

The following procedures are needed:

Work-Loop()

Comment[Each processor is in this infinite loop.]
Do forever
    Process messages;
    Take next task from work-set;
    Do-Task(Task-Name);
End do forever;
Return from procedure.

Do-Task(Task-Name)

Set Current-Goal = Literal in input node of conjunct graph;
Set Current-Goal = Binding list of task applied to Current-Goal;
Get binding lists from unifying Current-Goal with assertions;
Compose binding list of task with each of these binding lists *and*
Send the composite binding lists to outputs of input node using
    Input-Task messages;
Create or-parallel children tasks, one for each applicable rule.
Comment[These are 2-T DAGs of nodes. Each has a Head-Process as
    the input node and a Tail-Process as the output node. The
    invocation-binding of each child task is set appropriately.];
Do **Peel-Off-Input(Task-Name)** for each of these tasks *and*
Send the resulting tasks to appropriate processors using Input-Task
    messages;
Return from procedure.

Peel-Off-Input(Task-Name)

Output = {};
Do for each node that corresponds to an output of the input node
    Set new-CG = The 2-T subgraph of the task conjunct graph that
    includes the node and all its successors;
    Add task description consisting of this new-CG and the
    original BL to Output;
End do for each;
Return from procedure with the resulting set of tasks in Output, after
giving each task a name.

Input-Task-Processing(Input-Task-Message)

Let Process1 = The Process to which the Input-Task-Message
is directed;
If the process does not exist, create it using the information
in the message;
Update state of process1 to reflect the receipt of a task
on the input specified in the message;
Incremental-Cartesian-Product(Process1);
Place each new task from cartesian product in work-set *unless*
process1 is a tail-process;
If process1 is a tail-process
then
Let task-name1 = child-task-name of process1;
Send binding list of task to parent-task of task-name1 using
the function **done-task**;
If there are no bindings but instead there is an EOS
and if there is an EOS on each input of the tail-process
and if the cartesian-product is complete up to the EOS from
each input
then
Send EOS to parent-task of task-name1 using
the function **done-task**;
Return from procedure;
Return from procedure.

Incremental-Cartesian-Product(Process)

Create as many new tasks from the cartesian product of the inputs as
possible, starting from when the last time this procedure was invoked
on this process;
Comment[When a task is created for the cartesian product out of
two input tasks, the binding list of the new task is set to the union
of the binding lists (treated as sets) of the two input tasks];
Save state with the process to indicate the extent to which
the cartesian-product has been formed;
Return from procedure.

Done-Task(Parent-Task Task-Name Solution)

Let Parent-Task = Task to which this solution is directed;
If the message contains an EOS
and if the process that spawned this task has received
EOS on each of its inputs
and if each of the spawned tasks has had all subtasks
created for it
and if each spawned task has received an EOS from each of
its subtasks
then
Let Output-Tasks = **Peel-Off-Input(Parent-Task)**;
Send EOS in input-task message to each task in
output-tasks;
Return from procedure;
Compose the binding list of Parent-Task with the binding list
received;
Let Output-Tasks = **Peel-Off-Input(Parent-Task)**;
Set binding list of each task of output-tasks to be the
composite binding list;
Send each task in output-tasks to appropriate
processor using input-task message;
Return from procedure.

3.5. An Example

Consider the example database shown in figure 3-1. The distribution of the database is also indicated in the figure.

The dataflow* graph associated with the database for the query R(X,Y) is shown in figure 3-2. Boxes indicate processes. The literals inside the boxes are the literals to be solved by the processes. The exceptions are the Head and Tail processes which have "Head" and "Tail" in the boxes respectively. Dotted lines around sets of boxes indicate that those processes reside in the same processor. The name of the processor is indicated as a name of the form P_i. Arcs that cross dotted lines indicate streams of *input-task* messages. Task names (of the form T_i) are written next to the arcs. "," indicates temporal sequencing. Arcs inside dotted lines indicate function calls within the same processor to set up child tasks (downward arcs) and to send solutions to parent tasks (upward arcs). The arcs used to indicate function calls to set up child tasks are marked with the name of the clause used (of the form C_i).

The top level task is T_1 at processor P_i. It turns out that it has only one literal "r(X,Y)" to solve. In general, there could be an arbitrary number.

Notice a couple of different dataflow* subgraphs for child tasks. The conjunct "p(X),q(Y),s(X,Y)" leads to the conjunct graph with "p(X)" and "q(Y)" solved independently followed by "s(X,Y)". In the case of the conjunct "m(X),n(X,Y)", the two literals have to be solved sequentially because they share the variable "Y".

Finally, figure 3-3 shows a simplified description of each task/solution The 2T-DAG (or conjunct graph) and binding list is shown for each task using a simple graphical notation. Each node represents a process. For the sake of simplicity, only the literal field of the process tuple (or the special symbol *Head* or *Tail*) is shown for each node. In addition, IB and PT denote the invocation binding and the parent task respectively.

It is very instructive to go through the entire example. The next best thing is to note the following interesting facts.

Each process is responsible for solving the input node in the conjunct graph of each task on its input streams. The tasks on the streams leaving the process incorporate the solutions of the input nodes. The outgoing tasks are, therefore, the incoming tasks with the input node "peeled off". For example, look at task T47 and task T49.

A child task such as T33 gets a variable renamed X101 uniquely because the unification between the relevant goal of the parent task and clause head left a variable in the clause unbound.

Bindings received by a process on its input tasks are composed with the bindings received from the solution of the child tasks. The composed binding is sent out with the outgoing tasks. For example, observe how the bindings in input task T50 are composed with the bindings received from child task T61 to send out the bindings of output task T52.

Cartesian product of multiple input streams at an process creates new tasks with new names. T37, T38, T39, and T40 are created out of the cartesian product of T18, T19 and T21, T22.

C1	r(X,Y) :- p(X),q(Y),s(X,Y)	At processor P1
C2	p(a)	At processor P2
C3	p(b)	
C4	q(Y) :- m(X),n(X,Y)	At processor P3
C5	m(a)	At processor P4
C6	m(b)	
C7	n(b,a)	At processor P5
C8	n(b,b)	
C9	s(a,b)	At processor P6
C10	s(b,a)	

Figure 3-1: An Example Database

4. Extensions to Basic Model

4.1. Handling Storage Constraints

4.1.1. Large Databases

As mentioned before, the basic execution deals with the case where all clauses that can be used to reduce any particular atomic proposition goal reside in a single processor. One can achieve this if, for example, one partitions the database on the basis of the predicate symbol of the head of the clause. Each partition is mapped onto a single processor.

One should note here that it is not necessary to partition the database on this basis. Any other syntactic basis for partitioning will do as well provided any given goal's relevant clauses are a subset of a single partition.

Of course, it is possible that a particular partition may not fit in a single processor due to memory constraints. In addition, one may want to spread a partition over many processors so that one may exploit the parallelism in a single backward-chaining step also. The goal proposition may be unified in parallel with the heads of the relevant clauses and subtasks may be created in parallel.

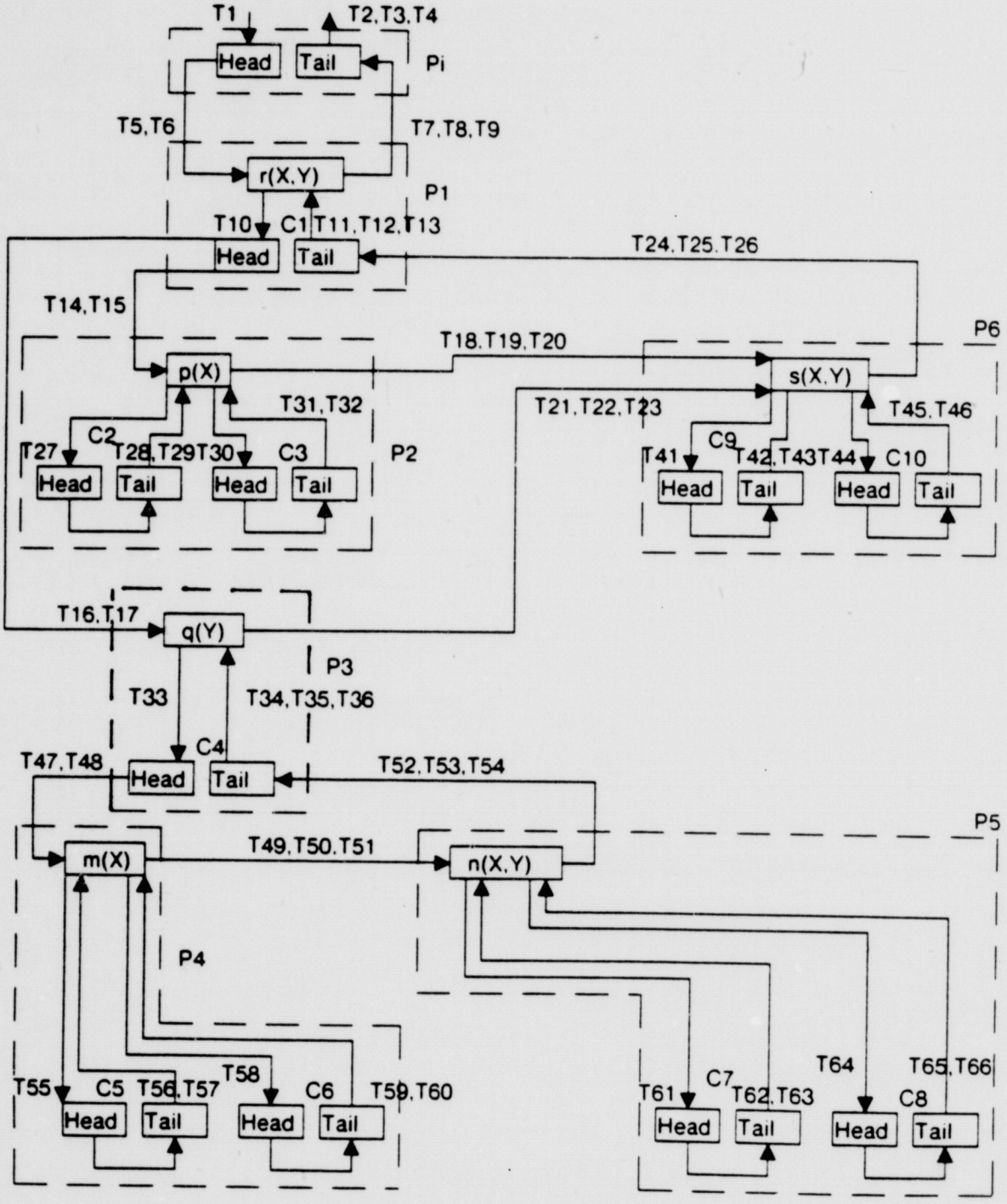


Figure 3-2: Dataflow* Graph for Example

$\dot{T}_1$ = [Head → R(X, Y) → Tail , ((t.t))] { IB = ((t.t)) PT = none

T_2 = [nil , ((X.a)(X101.b)(Y.b)(t.t))] PT = none

T_3 = [nil , ((X.b)(X101.b)(Y.a)(t.t))] PT = none

T_4 = [nil, EOS] PT = none

T_5 = [R(X, Y) → Tail , ((t.t))]

T_6 = [R(X, Y) → Tail , EOS]

T_7 = [Tail , ((X.a)(X101.b)(Y.b)(t.t))]

T_8 = [Tail , ((X.b)(X101.b)(Y.a)(t.t))]

T_9 = [Tail , EOS]

T_{10} = [Head → P(X) → S(X, Y) → Tail , ((t.t))] { IB = ((t.t)) PT = T5
Head → Q(Y) → S(X, Y) → Tail

T_{11} = [nil, ((X.a)(X101.b)(Y.b)(t.t))] PT = T5

T_{12} = [nil , ((X.b)(X101.b)(Y.a)(t.t))] PT = T5

T_{13} = [nil, EOS]

T_{14} = [P(X) → S(X, Y) → Tail , ((t.t))]

T_{15} = [P(X) → S(X, Y) → Tail , EOS]

T_{16} = [Q(Y) → S(X, Y) → Tail , ((t t))]

T_{17} = [Q(Y) → S(X, Y) → Tail , EOS]

T_{18} = [S(X, Y) → Tail , ((X.a)(t.t))]

T_{19} = [S(X, Y) → Tail , ((X.b)(t.t))]

T_{20} = [S(X, Y) → Tail , EOS]

T_{21} = [S(X, Y) → Tail , ((X101.b)(Y.a)(t.t))]

T_{22} = [S(X, Y) → Tail , ((X101.b)(Y.b)(t.t))]

T_{23} = [S(X, Y) → Tail , EOS]

T_{24} = [Tail , ((X.a)(X101.b)(Y.b)(t.t))]

T_{25} = [Tail , ((X.b)(X101.b)(Y.a)(t.t))]

T_{26} = [Tail , EOS]

Figure 3-3: Task Descriptions

$T_{27} = [\text{Head} \rightarrow \text{Tail}, ((t.t))]$ $\begin{cases} IB = ((X.a)(t.t)) \\ PT = T_{14} \end{cases}$

$T_{28} = [nil, ((X.a)(t.t))]$ $PT = T_{14}$

$T_{29} = [nil, EOS]$

$T_{30} = [\text{Head} \rightarrow \text{Tail}, ((t.t))]$ $IB = ((X.b)(t.t))$ $PT = T_{14}$

$T_{31} = [nil, ((X.b)(t.t))]$ $PT = T_{14}$

$T_{32} = [nil, EOS]$ $PT = T_{14}$

$T_{33} = [\text{Head} \rightarrow M(X101) \rightarrow N(X101, Y) \rightarrow \text{Tail}, ((t.t))]$ $\begin{cases} IB = ((t.t)) \\ PT = T_{16} \end{cases}$

$T_{34} = [nil, ((X_{101}.b)(Y.a)(t.t))]$ $PT = T_{16}$

$T_{35} = [nil, ((X_{101}.b)(Y.b)(t.t))]$ $PT = T_{16}$

$T_{36} = [nil, EOS]$

$T_{37} = [S(X,Y) \rightarrow \text{Tail}, ((X.a)(X_{101}.b)(Y.a)(t.t))]$

$T_{38} = [S(X,Y) \rightarrow \text{Tail}, ((X.a)(X_{101}.b)(Y.b)(t.t))]$

$T_{39} = [S(X,Y) \rightarrow \text{Tail}, ((X.b)(X_{101}.b)(Y.a)(t.t))]$

$T_{40} = [S(X,Y) \rightarrow \text{Tail}, ((X.b)(X_{101}.b)(Y.b)(t.t))]$

} CARTESIAN PRODUCT OF T_{18}, T_{19} AND T_{21}, T_{22}

$T_{41} = [\text{Head} \rightarrow \text{Tail}, ((t.t))]$ $\begin{cases} IB = ((t.t)) \\ PT = T_{38} \end{cases}$

$T_{42} = [nil, ((t.t))]$ $PT = T_{38}$

$T_{43} = [nil, EOS]$

$T_{44} = [\text{Head} \rightarrow \text{Tail}, ((t.t))]$ $\begin{cases} IB = ((t.t)) \\ PT = T_{39} \end{cases}$

$T_{45} = [nil, ((t.t))]$ $PT = T_{39}$

$T_{46} = [nil, EOS]$ $PT = T_{39}$

$T_{47} = [M(X101) \rightarrow N(X101, Y) \rightarrow \text{Tail}, ((t.t))]$

$T_{48} = [M(X101) \rightarrow N(X101, Y) \rightarrow \text{Tail}, EOS]$

$T_{49} = [N(X101, Y) \rightarrow \text{Tail}, ((X_{101}.a)(t.t))]$

$T_{50} = [N(X101, Y) \rightarrow \text{Tail}, ((X_{101}.b)(t.t))]$

$T_{51} = [N(X,Y) \rightarrow \text{Tail}, EOS]$

$T_{52} = [\text{Tail}, ((X_{101}.b)(Y.a)(t.t))]$

$T_{53} = [\text{Tail}, ((X_{101}.b)(Y.b)(t.t))]$

$T_{54} = [\text{Tail}, EOS]$

$T_{55} = [\text{Head} \rightarrow \text{Tail}, ((t.t))]$ $IB = ((X_{101}.a)(t.t))$ $PT = T_{47}$

$T_{56} = [nil, ((X_{101}.a)(t.t))]$ $PT = T_{47}$

$T_{57} = [nil, EOS]$ $PT = T_{47}$

Figure 3-3, continued

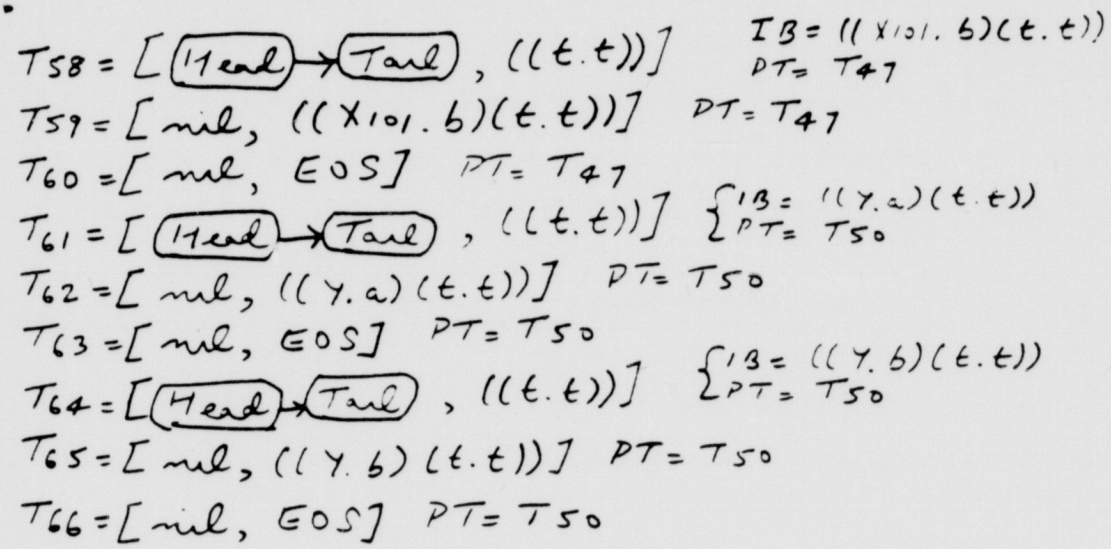


Figure 3-3, concluded

The solution is to maintain a single processor as being responsible for each partition (as before). However, instead of the clauses in a partition physically residing in the responsible processor, the clauses are distributed over a certain neighborhood of the processor. One could, for example, distribute the partition over all processors within some number of message hops away from the responsible processor.

Two extra message types are required now to make the subtask creation and solution propagation possible. The message types are **Do-Task** and **Done-Task**.

The *Arguments* field of the **Do-Task** message type is of the form:

<*Task-Name Task-Description Sending-Processor-Name*>

Task-Name is the name of the task that needs to be worked on. *Task-Description* is its description. *Sending-Processor-Name* is the identity of the processor that is sending the message.

The *Arguments* field of the **Done-Task** message type is of the form:

<*Parent-Task-Name Task-Name Sending-Processor-Name Solution*>

Parent-Task is the name of the parent task of Task-Name. *Sending-Processor-Name* is the name of the processor where the message originated and also the location of *Task-Name*. *Solution* is the

binding list being reported as the answer to *Task-Name*. Again, end of stream may be indicated by an EOS binding list.

The messages are used in the following way. Processes still reside at the processor responsible for the relevant partition. The relevant partition is the one that is relevant to solving the atomic proposition associated with the process. When a process receives an input-task message, it finds the incremental-cartesian product as before. The new tasks are, however, not solvable locally. They are sent to the neighborhood associated with the relevant partition around the processor using **Do-Task** messages (i.e., each processor in the neighborhood receives a copy of the **Do-Task** message). These processors in the neighborhood create subtasks just as the single responsible processor would in the basic execution model. The difference now is that solutions must be communicated back to the responsible processor using **Done-Task** messages. This is analogous to invoking the function **Done-Task** locally in the basic execution model. End of stream is indicated as before with the *Solution* argument set to *EOS*. The difference here is that each processor in the neighborhood, including those that cannot create any solutions or subtasks, must report to the responsible processor when all subtasks have been generated using the **Done-Task** message. This is done by setting the *Task-Name* argument of the message to *nil* and the *Solution* argument to *EOS*. In the basic execution model, since all clauses that could be used to create a subtask were in a single responsible processor, the responsible processor knew locally when all possible subtasks had been created. The new mechanism is necessary to replace knowledge that no longer resides locally.[1]

In addition, one needs to maintain a flag at the parent-task to indicate whether all possible subtasks have been found. This flag is *false* when a task is first created using cartesian product at a process. After a **Do-Task** message is sent out to the appropriate neighborhood and after the responsible processor has received an indication from each processor that all subtasks have been generated, then the flag can be set to *true*. This is the flag that will be checked in the extended execution model analog of the **Done-Task** function of the basic execution model when the function needs to check that "each of the spawned tasks has had all the subtasks created for it".

---

[1] A more efficient solution to propagating **Do-Task** and **Done-Task** messages to/from the neighborhood is possible but the idea here is merely to show that a *satisfactory* solution to the problem exists.

4.1.2. Long Streams

Only the sketch of a solution will be presented here.

To recapitulate, the problem is the following. Processes may have multiple input streams that contain various solutions to other goals. A cartesian product of the streams has to be computed. To create this cartesian product, essentially the process has to store complete streams till the entire cartesian product has been obtained. Since it may be hard to accurately predict the lengths of these streams ahead of time, it is possible that the processor responsible for the process may not have the requisite storage.

The solution is to sequentialize the dataflow* graph upstream from the process up to the previous subtask/parent-task boundary. As much sequentialization is done as is necessary to remove the memory problem. In the worst case, the sequentialization may lead to a linear sequence of processes requiring absolutely no cross-products of streams. Figure 4-1 shows an example of this process. In the example, the node corresponding to t3(X,Y,Z) is the one that gets into a memory constraint situation.

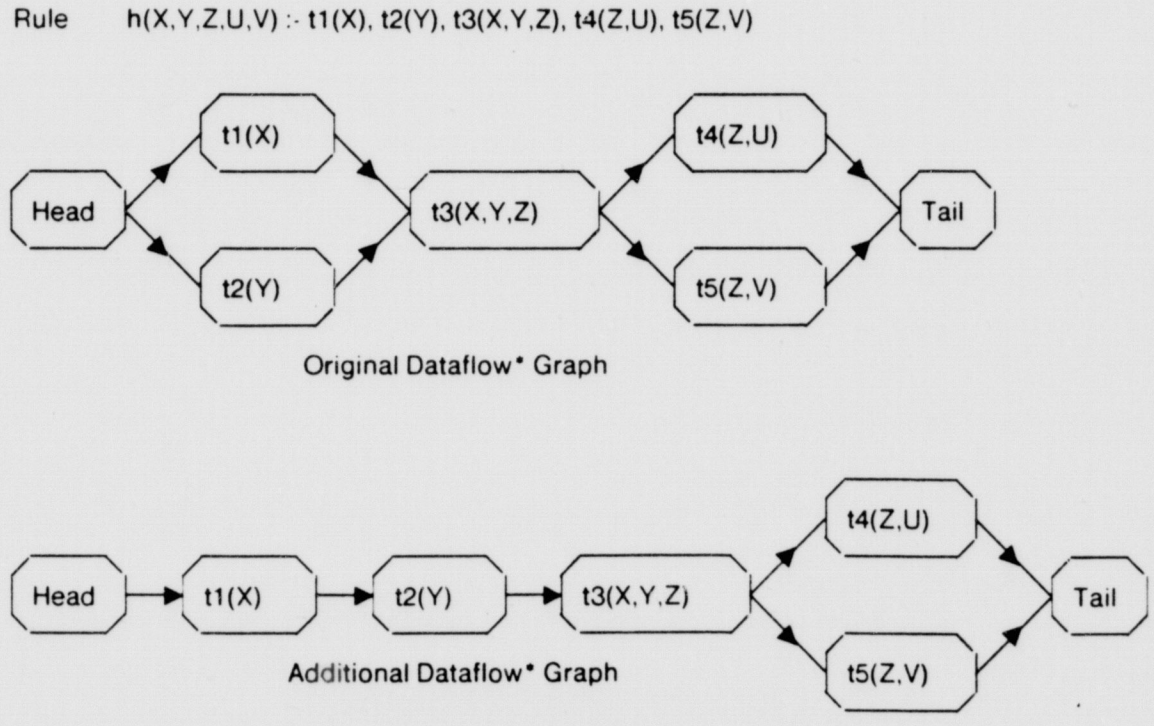


Figure 4-1: Handling Long Streams

Notice that the new dataflow* graph must exist independently along with the old dataflow* graph. This is necessary because tasks/solutions may be still in the pipeline in the original dataflow* graph when the new graph is introduced. Therefore, more than one tail process may exist for a certain task. When solutions flow out of the tail processes, an indeterminate merge of these streams must happen. Also, an EOS is sent from the collection of tail processes only when they have all produced an EOS.

4.2. Handling Non-Ground Bindings

Again, only the sketch of a solution will be presented here.

The problem is that if a process produces non-ground bindings for the atomic proposition associated with it, then some downstream processes that work in parallel may not be able to do so any more. Processes should execute in parallel only if the bindings they are expected to produce are not for any common variables. A non-ground binding from a preceding process may remove this necessary condition.

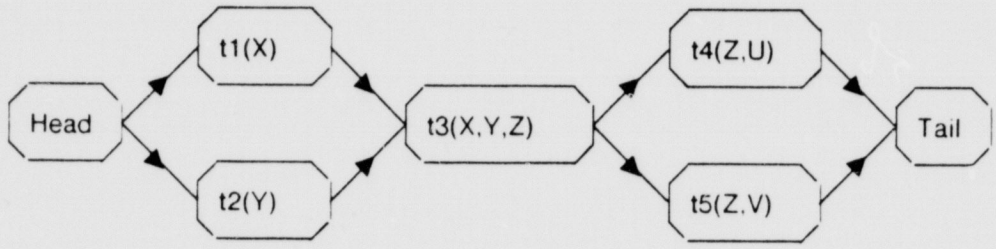
The solution is more or less complementary to the solution for the long stream problem. The dataflow* graph *downstream* from the process in question is sequentialized as much as necessary in order to avoid the problem. Figure 4-2 shows an example of this process. In the example, the node t3(X,Y,Z) is expected to produce a ground binding for the variable Z but does not. Similar to the long stream case, the multiple dataflow* graphs coexist independently. Multiple tail processes are handled as before.

5. Discussion

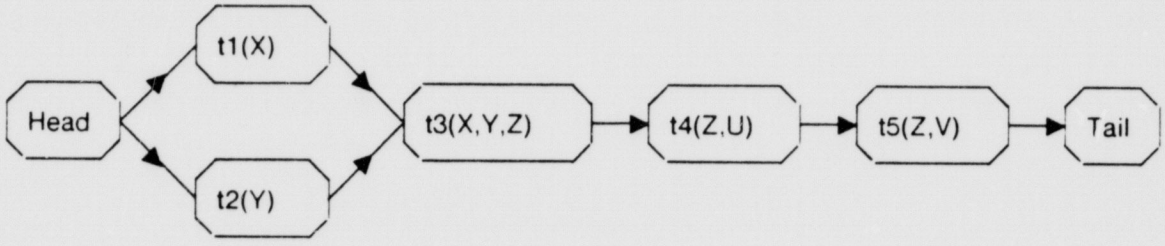
5.1. The Larger Picture

The work presented in this paper has been done as part of a dissertation in progress by Vineet Singh — "Distributing Deductions to Multiple Processors" [11]. The dissertation will show that controlling communication cost is a necessary and achievable condition for the successful use of multiprocessing for backward-chaining deductions. Two important general principles of controlling the cost of communication will be used: (1) The design of special purpose protocols and procedures to reduce communication cost outright. (2) Utilization of the tradeoff between parallelism and communication cost. These general principles will be shown to work for two fairly different case studies. The first case study deals with dynamic allocation of deductions for fairly large processors with replicated databases on a broadcast network [10]. The second case study deals with static allocation for large numbers of small processors with a distributed database and local connections to

Rule h(X,Y,Z,U,V) :- t1(X), t2(Y), t3(X,Y,Z), t4(Z,U), t5(Z,V)



Original Dataflow* Graph



Additional Dataflow* Graph

Figure 4-2: Handling Non-Ground Bindings

neighbors in a 2-D plane. *PM* was designed in the context of this second case study. Further work on the second case study will involve designing resource allocation techniques and an implementation to provide empirical validation of results.

5.2. Related Work

The research presented in this paper builds on two important sets of ideas. One is the exploitation of and-parallelism as described by Conery in his dissertation [3]. The other is the exploitation of or-parallelism and pipelining as described independently by Singh et al. in [9, 10] and Lindstrom et al. in [5]. The connections of *PM* with these two sets of ideas are described below.

Conery's execution model exploited a restricted sort of and-parallelism. This restriction is exactly the one used in *PM*. The difference is that *PM* exploits an additional form of parallelism — pipelining. It does this by streaming the solutions through the dataflow graphs of the conjuncts in conjunctive goals. Conery's execution model, on the other hand, sends one solution at a time along dataflow arcs. Further solutions are sent on the prodding of another level of control analogous to backtracking in sequential Prolog.

Singh et al. and Lindstrom et al. showed how or-parallelism and pipelining could be exploited together. in both of these pieces of research, conjunctive goals were solved from left to right in sequence. *PM* exploits and-parallelism also by using the idea of streaming for pipelining but allows the total order of conjuncts to be changed to a partial order. Or-parallelism is exploited as before. However, the cost of exploiting the additional parallelism is that a dataflow solution (modulo indeterminate merge) has a non-dataflow feature, cartesian product of streams, added to it. Although cartesian product does require state to be maintained, the good news is that it is only local state. No global state is maintained.

5.3. Conclusions

This paper has described *PM*, a parallel execution model for backward-chaining deductions. The most important contribution of this paper is that *PM* can simultaneously exploit *or-parallelism*, *and-parallelism*, and *pipelining*. This is more parallelism than can be exploited by some execution models described in the recent literature. The extra parallelism can be an important advantage in a situation where large numbers of processors are available. *PM* can work in an environment where processors may communicate by message-passing only and for which the database may be distributed.

References

[1] Barr, Avron and Edward A. Feigenbaum (Eds.).
Search.
William Kauffman, Inc., Los Altos, California, 1981, pages 39, chapter 2.

[2] Barr, Avron and Edward A. Feigenbaum (Eds.).
Automatic Deduction.
William Kauffman, Inc., Los Altos, California, 1982, pages 80, chapter 12.

[3] Conery, John Simpson.
The And/Or Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, University of California, Irvine, 1983.

[4] DeGroot, Doug.
Restricted And-Parallelism.
In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471-478. ICOT, Japan, 1984.

[5] Lindstrom, Gary and Panangaden, Prakash.
Stream-Based Execution of Logic Programs.
In *IEEE Logic Programming Conference*, pages 168-176. IEEE, February, 1984.

[6] Moto-oka, Tohru.
Fifth Generation Computer Systems: A Japanese Project.
Computer :6-13, March, 1984.

[7] Nilsson, N.J.
Principles of Artificial Intelligence.
Tioga Publishing Company, 1980.

[8] Shapiro, Ehud Y.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, ICOT, Japan, January, 1983.

[9] Singh, Vineet and Michael R. Genesereth.
A Variable Supply Model for Distributing Deductions.
Technical Report HPP-84-14, Heuristic Programming Project, Computer Science Department, Stanford University, May, 1984.
A version of this will appear in IJCAI 85.

[10] Singh, Vineet and Michael R. Genesereth.
A Variable Supply Model for Distributing Deductions.
In *Proceedings of IJCAI-85*. Morgan Kaufmann Publishers Inc., August, 1985.

[11] Singh, Vineet.
Distributing Deduction to Multiple Processors.
PhD thesis, Stanford University, December, 1985.

[12] Smith, Dave and Michael R. Genesereth.
Ordering Conjunctive Queries.
Artificial Intelligence , October, 1984.