A MODEL FOR EXECUTION OF PARLOG ON A DISTRIBUTED PROCESSOR NETWORK

Barton Schaefer
Peter Borgwardt

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 87-007
June, 1987

A Model for Execution of PARLOG on a Distributed Processor Network

Barton Schaefer
Peter Borgwardt

Technical Report No. CS/E 87-007
June 1987

A Model for Execution of PARLOG on a Distributed Processor Network

Barton Schaefer, Oregon Graduate Center

Peter Borgwardt, Imaging Research Laboratory, Tektronix, Inc.

June 29, 1987

ABSTRACT

Message-passing multiprocessors hold the promise of low cost per node and a large upper bound on the number of nodes, and logic languages provide inherent parallelism to exploit these multiprocessors. PARLOG is a very attractive language in this regard, because it is designed for use in parallel systems programming. This paper presents an execution model and implementation strategy for PARLOG, based on the work of Crammond but extending the AND/OR-tree model to a distributed processor network. Difficulties in adapting the execution model to a distributed system are discussed, some minor problems with the Crammond implementation are addressed, and examples of compiled programs and of execution are presented. The paper concludes with a brief discussion of the application of the model and execution strategy to a transputer PARLOG system.

This research was supported in part by National Science Foundation Grant No. ECS-8400758 at the Oregon Graduate Center, where the second author is an adjunct professor.

1. Introduction

There has been a great deal of recent interest in message-passing multiprocessors, and there have even been some commercial introductions (*e.g.*, the Intel hypercube and the N-cube hypercube). The primary attractions of an architecture based on message passing over one based on shared memory are that a multiprocessor can be built at a lower cost per processing node and that the practical upper bound on the number of processing nodes is much larger. However, the programming of such networks is still at a primitive level. In order to apply the parallelism of the network to a problem, the programmer must first divide the given problem into a number of parallel communicating processes, and then must map these processes onto the network of processors. The use of logic programming languages holds out the promise of moving parallel programming to a much higher level; these languages have inherent parallelism that can be detected by the system and dynamically spread across the processing network, with reliance on load balancing to keep all the processors reasonably active [PaB87].

PARLOG, designed for easy exploitation of parallelism and also for systems programming, is a strong candidate for this application. Guarded clauses are used in PARLOG to achieve the committed-choice non-deterministic solution of logic queries [ClG85, ClG86]. The guards allow each alternate clause of a predicate to be tried in parallel; only a clause whose guard succeeds will commit to further execution of the body of its clause. If more than one guard succeeds, then the system will non-deterministically choose only one clause to continue and will kill the rest.

PARLOG also exploits stream AND-parallelism by executing the goals of a clause body as parallel processes. Stream communication is accomplished via the sharing of the logical variables (*ala* producer–consumer relations in CSP). This is signalled to the compiler by supplying mode declarations with each clause, declaring each argument either input or output. If a goal attempts to unify a term with an unbound input argument, that goal suspends execution until some other goal provides a binding for the variable. Output arguments must come into the goal unbound, so that the goal can bind these if a clause commits.

A further advantage of PARLOG is its use of a specialized form of unification based on term matching and tests for syntactic identity [Gre85]. This is simpler, more efficient, and easier to implement in a parallel environment than full unification. This has the side effect of eliminating "multiple mode" programs, *e.g.*, predicates that can be used both to construct and break down data structures, but such programs can nearly always be replaced by a few more specialized predicates.

The model presented here starts with Crammond's work [CrM84, Cra86] on executing PARLOG (and other committed-choice non-deterministic languages) on shared-memory multiprocessors, which in turn is based on the AND/OR-tree model of execution [CoK81, Con83]. To eliminate reliance on shared memory, the concept of *ownership* of variables has been added. Bindings are distributed throughout the network, and the setting and questioning of bindings is done via messages; the routing of such messages is related to the structure of the PARLOG program that is executing. Also, information about PARLOG processes must be maintained in a distributed fashion in the network. For this reason, a *grandparent pointer* has been added to the information recorded for each process; this aids in the *promotion* of child processes. Finally, some miscellaneous problems in adapting Crammond's model to a message-passing architecture are dealt with.

2. Execution Model

As has been noted, the execution model given here is based on one proposed by Crammond [CrM84, Cra86]. The essential difference is that this model assumes no global state; all transfers of information are made explicit. Crammond's model is therefore a subset of this one, suitable for a shared memory multiprocessor; this model is intended to be suitable for either a shared or distributed memory system.

The concept of *ownership* of variables is an important addition. With no global state, each process must become responsible for any variables that it originates, *i.e.*, that are local to that process and its descendants. A process is said to *own* its local variables, and other processes will supply and/or request bindings for these variables by communicating with the owner via messages, as described below. The owner also maintains a *suspension list* for each unbound variable, listing processes that have

requested a binding.

Types of Processes

As in Crammond's model, there are two process types, *goal* and *clause*. It is important to note that these are *logical* classifications; no record of the type of a particular process is kept at run time. The compiler must generate for each predicate one subprogram that acts as the goal process and one or more subprograms to act as clause processes. The predicate is invoked by calling the goal subprogram, which contains instructions to call the appropriate clause subprograms and wait for their results.

Goal (OR process)

Goal processes are responsible for creation of new processes to solve each of the alternative clauses matching the goal, and may be promoted to replace earlier goals of which they are known to be the only remaining subgoals.

Clause (AND process)

Clause processes are created by goal processes, one for each alternative clause matching the goal. They are responsible for creation of new goal processes for subgoals in the right hand side of the clause.

Process States

The process states correspond to those of Crammond's model, with the addition of a state for processes which have terminated.

Runnable, Executing (RE)

Processes in this state are (obviously) those which are currently executing.

Runnable, Queued (RQ)

Processes in this state are ready but waiting; this is the initial state of all processes.

Suspended on Variable (SV)

Processes in this state are waiting for an input variable to be instantiated.

Suspended on Children (SC)

Goal processes waiting for a clause to commit and clause processes waiting for subgoals to finish will be in this state.

Dead (D)

The process has terminated, either as a result of an instruction or as a result of a message, but its process control block has not yet been deallocated. Processes which have terminated but are waiting for all of their children to terminate will be in this state.

Process Control Messages

Process control messages are those that can cause the recipient process to change state or to send process control messages to other processes, as well as provoking other actions. All processes must respond to process control messages in the same ways at all times, because there is no differentiation of process types at run time and because the messages are asynchronously received. The messages described here are essentially identical to Crammond's process signals, with the addition of the COMMIT message.

DONE (goal process succeeded / clause process failed)

Sent from child to parent, this message causes the parent to remove this child from its list of children. If the parent's state is SC and this is the last child, the parent's state changes to RQ.

QUIT (goal process failed / clause process succeeded)

Sent from child to parent, this message causes the parent to send KILL to all its other children, send DONE to its parent, and change state to D.

KILL

Sent from parent to child, this message causes the child to send KILL to all its children and

change state to D.

COMMIT

Sent from child to parent, this message causes the parent to send KILL to all its other children (but not to the child from which the COMMIT was received). The parent ignores all subsequent COMMIT messages.

Information Messages

Information messages are those that transfer information from one process to another, or that request that information be transferred. These messages do not directly cause the recipient process to change state, although the information carried by a BIND message may awaken a suspended process. These have been added to Crammond's model to handle interprocess communication in a distributed memory environment.

NEED(V)

This message is sent by a process that requires the binding of variable V to the process that owns V. Possible responses of the receiver:

(1) V is unbound – add the sender to the suspension list for V.

(2) V is bound – forward the binding to the sender.

BIND(V, T)

This message signals that a term T is to be bound to the variable V, and is sent in two cases:

(1) From a process that has found a binding T for variable V, to the owner of V. The owner will ignore the BIND if V is already bound, and will delay the BIND if no COMMIT has yet been received.

(2) From the owner of V to a process that has sent a NEED(V) message. When an owner receives BIND(V, T), it immediately forwards the BIND to all processes on V's suspension list.

Process Execution

Goal Processes

The primary function of goal processes is the creation of clause processes, although some preliminary testing of arguments can be done at the goal level. A clause process is created for each clause of the goal, and the process then enters state SC. Possible outcomes are:

(a) All children send DONE (failure) – the process sends QUIT to its parent and terminates.

(b) Any child sends COMMIT – the process sends KILL to its other children and resumes waiting. If the last child sends DONE, the process sends QUIT to its parent; otherwise (when the committed child sends QUIT), the process sends DONE to its parent. In either case, the process terminates.

(c) Any child sends QUIT – the process sends KILL to any remaining children, sends DONE to its parent, and terminates.

Clause Processes

Four stages of clause process evaluation are outlined by Crammond [Cra86]: head unification, solving of guard goals, committing, and solving of body goals. In PARLOG, predicates are generally compiled to a *standard form* [Gre85], so that head unification and guard solution can be done in parallel. However, exploiting parallelism in this manner may require the creation of processes whose only purpose is to perform the unification; it may be more efficient to perform some head unifications sequentially in the clause process, before creating new processes to complete the head unifications and solve the guard. For this reason, and for reasons of clarity, head unification will be considered a separate stage in this model, as it is in Crammond's. However, the use of the COMMIT message and the parent's response to it make committing trivial, so it is considered part of the solution of the guard. Execution of clause processes thus involves three stages.

(1) Unification

An attempt is made to unify arguments in the goal with arguments in the clause head. Possible outcomes are:

(a) Unification fails – the process sends DONE and terminates.

(b) Unification suspends – the process is switched to state SV, an appropriate NEED message is sent, and the process resumes here when awakened.

(c) Unification succeeds – the process continues immediately.

(2) Solving the Guard

The process spawns goal processes to solve the guard, then enters state SC. Possible outcomes are:

(a) QUIT is received from any child – the process sends KILL to its other children, sends DONE to its parent, and changes state to D.

(b) DONE is received from every child – the process sends COMMIT to its parent and continues.

(3) Solving the Body

In order to hold down the number of active processes and to shorten the process trees, goals are reduced whenever possible to the system of body goals in the committed clause. As in Crammond's model [CrM84, Cra86] this is accomplished by *promoting* the body goals so that they become the children of this process's grandparent and the siblings of the parent goal (see Figure 1). A process which is created in this manner is called a ***promoted process***. There are two possible cases when the clause body is solved:

(a) The body has no sequential conjunction – the process spawns promoted goal processes to solve the body, sends QUIT to its parent, and terminates.

(b) The body has a sequential conjunction – the process spawns the first sequential part without promoting, then enters state SC. When DONE has been received

from all goals in the first part, this case analysis is repeated for the rest of the body. Complex combinations of parallel and sequential conjunctions, such as

p <– (q & r) , (s & t).

must be resolved by the compiler, as will be explained later.

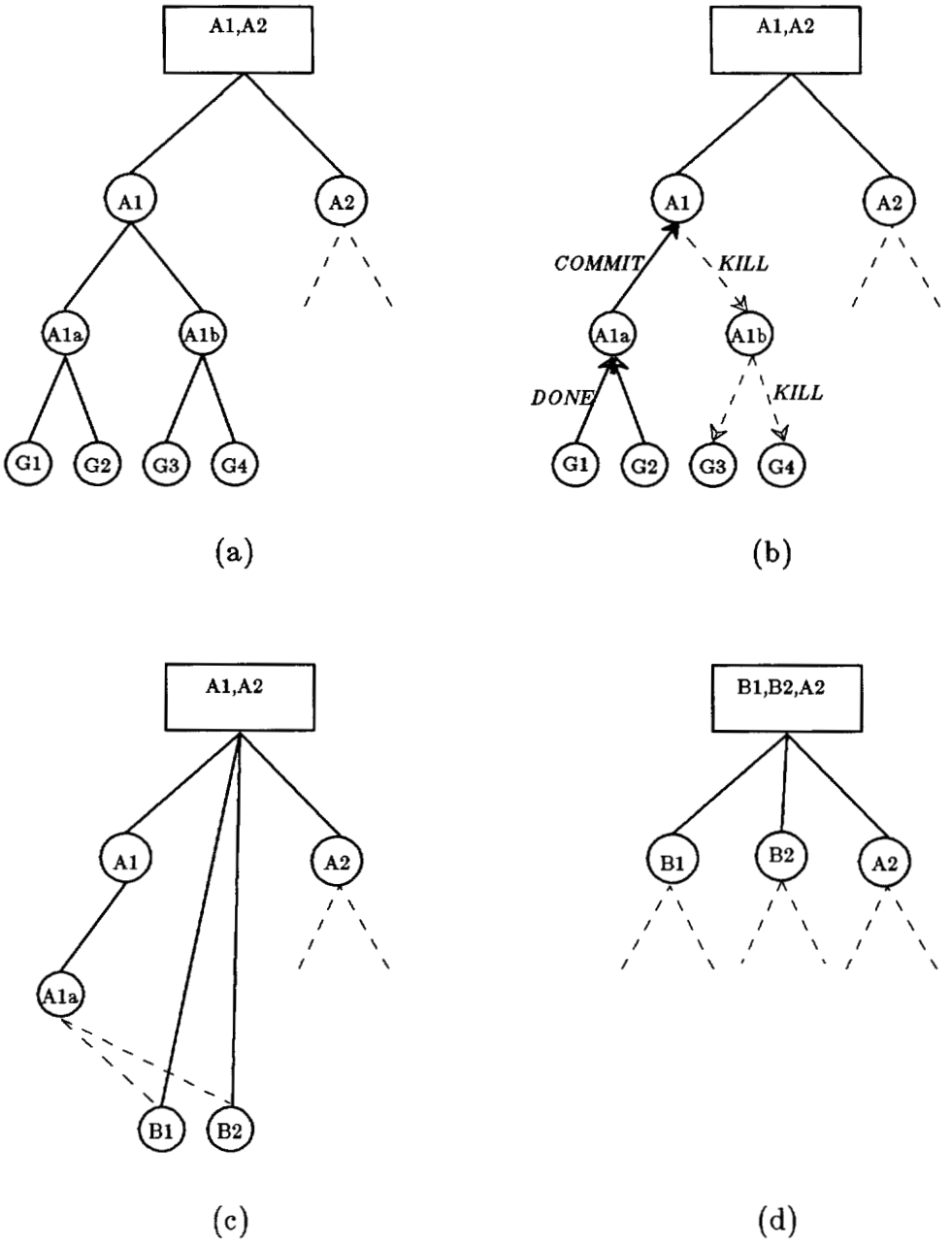
3. Complications of Distributed Execution

In addition to the basic changes that must be made to the execution model, there are two more serious complications that arise if it is to be used with a fully distributed system. The first, process promotion, applies to any committed-choice non-deterministic language; the second involves communication of large structures, and is complicated by PARLOG's implementation of stream communications.

3.1. Promoted Processes

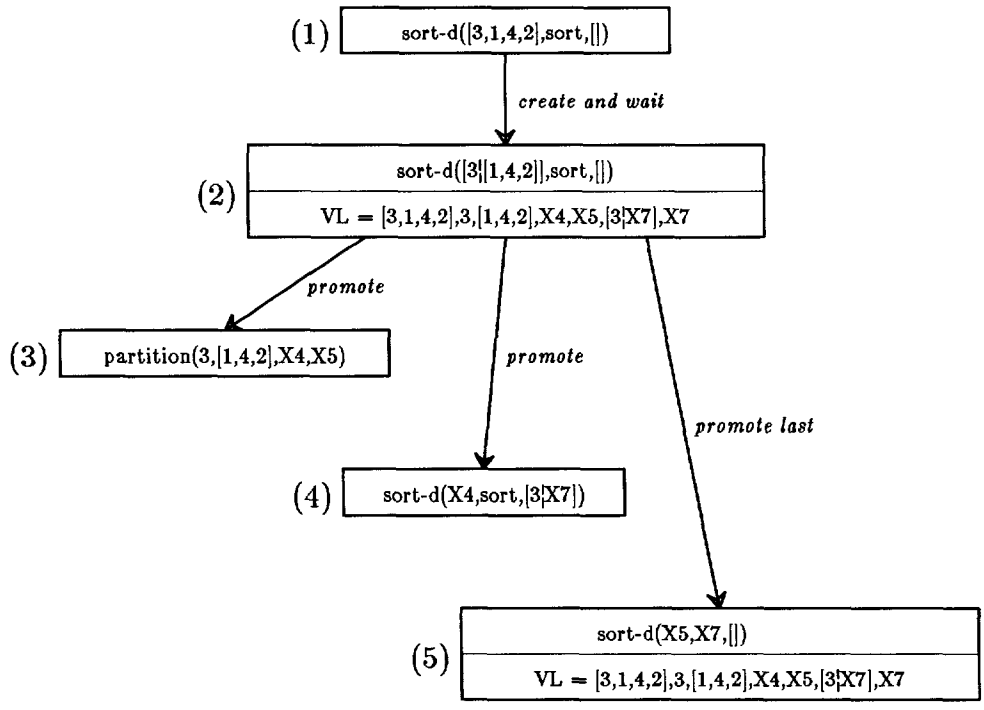
Promoted processes are a special problem for the use of variable ownership. The most significant problem arises when the parent must create new variables, which will then be passed to more than one promoted child process. The spawner will be terminating, so it cannot continue to own the new variables. This problem can be easily solved by turning over ownership of these variables to the last promoted process spawned; this will be called *inheritance* of variables (see Figure 2). A consequence of this is that the last promoted process must remain in existence until all of its siblings (and their descendants) have succeeded. However, because the promoted processes are goal processes, if any of the siblings fails the inheriting process can be terminated without further delay.

The mechanics of promoting a process also cause some trouble in a distributed system. Every clause process must either know the identity of its grandparent or be able to query its parent for this information, so that the promoted processes can be told about their new parent and so the grandparent can be notified of the existence of new children. The costs of maintaining more information about each process and transferring more information when a process is spawned must be compared to the cost of the communications necessary to establish the new parent-child relationship. If both of these costs are



(A1a) A1 <– G1 , G2 : B1 , B2.
(A1b) A1 <– G3 , G4 : B3 , B4.

Figure 1 – Process trees for the query "A1 , A2" at stages (a) solution of guards, before a clause for A1 has committed; (b) committing of clause A1a; (c) promotion of body goals B1 and B2; and (d) after termination of A1.




```
sort-d([],list,list).
sort-d([hd|tail],sorted,rem) <-
    partition(hd,tail,list1,list2),
    sort-d(list1,sorted,[hd|rem1]),
    sort-d(list2,rem1,rem).
```

Figure 2 – Inheritance of variables in the *sort-d* relation. (This relation is from [ClG84].) The last promoted *sort-d* goal (5) inherits the variable list of its parent clause (2); the parent clause can then terminate. Variables whose bindings are known have been replaced by their bindings in the variable list (VL) to show the relationship of the processes.

very high, it will be necessary to consider whether the benefits of process promotion (fewer active processes, shorter process trees) are worth the extra expense involved.

It is important to note that promotion can be done with only slight additional expense in the case of tail-recursive processes, even if it is not implemented in general. The tail call must be the last new process created, and can simply replace its parent by assuming its parent's identity, with the parent's variables inherited as described. This change of parent is invisible to any previously created children, and the only difference in the tail call process is that it must differentiate between messages from its

own children and messages from its siblings. The latter is necessary because this limited form of promotion introduces a case where a single process must play the roles of both clause (the original parent) and goal (the tail call) processes.

3.2. Large Structures and Stream Communication

In any distributed-memory system, it may become necessary that access to a large data structure be provided to a process executing on another node of the network. This is especially true in PARLOG, where processes are spawned to examine structures such as potentially infinite lists in parallel with their generation. Whether for reasons of efficiency, because of limited message sizes, or because the entire structure has not yet been generated, it may not be possible to transmit an entire structure in a single message. Two possible solutions to this communication problem are remote references and channels.

3.2.1. Remote References

Remote references provide a compact means for transferring information about data stored on one node to a process running on another node. One form of remote reference, the logical variable, has already been addressed; information about the binding of the variable is maintained by an *owner* process, and other processes must communicate with the owner to obtain or modify this information. This scheme can be extended to terms in general, using the same NEED and BIND messages for communication. Of course, remote references provide no savings in the case of small ground terms such as atoms and numeric values, so in practice it is only necessary to use remote references for structures. Furthermore, the *suspended on variable* process state can be generalized to *suspended on reference.*

The only information needed in a remote reference is the *type* of the term (*e.g.*, variable, list, etc.) and its *remote address.* The remote address includes the node on which the data is stored and an address (which need not be the actual physical address) of the term on that node. The type of the remote term is needed for efficiency; many decisions can be made based only on the type of a term without needing its actual value. For example, an attempt to unify an integer with a remotely referenced list can be completed (failed) without transmitting the components of the remote list.

Remote references, including logical variables, introduce the need to keep an accurate record of which terms must be retained in memory because their value may be required by another process, and which terms may be discarded and their storage re-used. Watson [Wat86] has proposed *remote reference rights* as a means to handle reference counting on a distributed system; a variation of this scheme is well suited to the distributed PARLOG model, and is best explained by an example.†

Let process A be the owner of a term a. When a reference to a is to be given to some process B, the reference count for a is incremented by a predefined amount. This amount is sent to B along with the reference to a, and represents the number of remote references to a that B has the *right* to hold and/or pass on.

Only A can increment the reference count for a; if B wishes to pass along a remote reference for a to a third process C, B must reduce its own rights by the number of rights given to C. B can give away all of its rights, but is thereafter not allowed to make any request for the value of a. When B releases its remote references to a (*e.g.*, when B terminates), it sends a message to A with the current rights value, and A decrements a's reference count by that amount. Rights are *not* released when passed along to another process.

If B wants to give away more rights than it currently holds, it creates an *indirect remote reference*, and then gives away rights to the indirect reference. However, an indirect remote reference may *not* be created if B currently holds *no* rights for a. Indirect references eliminate any need to send a message to A to obtain additional rights, and prevent the possibility of such messages "crossing in the mail" with decrement messages, *i.e.*, there is no chance that A will mistakenly think that a's reference count has gone to zero.

In the PARLOG model, remote references can be employed in two cases:

(1) When processes are spawned, structure-valued arguments may be passed as remote references or may have remote references as components.

† Example adapted from Foster [Fos87].

(2) When a BIND message is sent, some components of structure-valued bindings may be remote references. Note that it is never sensible to send a remote reference as the only value of a BIND message, since the fact that a BIND is being sent indicates that more detail about the binding is needed.

Use of remote references in these cases will reduce the size of process creation messages and delay transmission of structures until some process is actually ready to examine the contents. This may in turn reduce the number of messages, since communication will be more direct between the owner of the term and the process that will examine it. Indirect references could increase the number of messages, but this can be held to a minimum by adjusting the number of remote rights allocated.

Remote references are also released in two ways:

(1) When a BIND is sent in response to a NEED, the remote references held by the "needy" process are released, because that process now has a copy of the referenced term.

(2) When a process terminates, all remote reference rights it holds are released. This does not occur if the process is spawning promoted children; instead, the last promoted child inherits the parent's remote rights and indirect remote references, as well as the parent's variables.

Releasing remote references requires a few changes in the model. NEED messages can carry the current reference rights for the term they request, which are then deducted from the count for that term when the BIND is sent in reply. A new message, DECREMENT, is needed to handle rights released on termination; before a process completes, its list of terms must be searched and DECREMENT messages sent for any remote references it holds.

For the remainder of this paper, it is assumed that the remote reference rights technique is used to reference count logical variables, regardless of whether remote references are used in the general case.

3.2.2. Streams and Channels

One of the most important features of PARLOG is its use of stream communication between processes executing in parallel. Whenever a producer and a consumer goal share a common variable

which is eventually bound to an incrementally generated structure, a stream is created† between the producer and the consumer (see Figure 3). Each time the producer recursively spawns a new process to generate another element of the structure, and each time the consumer recursively spawns a new process to consume that element, the stream is extended to these new processes.

In fact, multiple consumers may be created for each new element, so the streams can branch arbitrarily; and the elements themselves may be variables whose bindings will be incrementally generated by some descendant of the consumer, requiring that new streams be established in the opposite direction. In a shared memory environment this is not a serious problem, for the streams can be logical "channels", with the bindings actually stored in memory and all access done by pointer traversal. In a distributed environment, however, the problem is a considerable one. There are two possible solutions: channels can actually be created, and extended as necessary; or true streaming can be abandoned, in which case all communications take place strictly at the level of NEED and BIND messages.

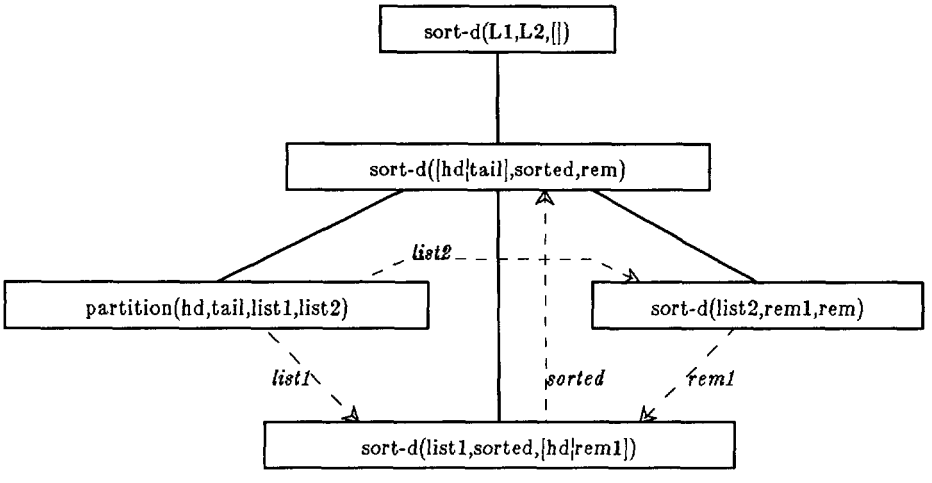


Figure 3 – Streaming in the solution of the *sort-d* relation.

† In [ClG84], streams were restricted to list structures; this restriction has since been removed. Note that streams are *not* created for variables bound to ground terms.

If true channels are to be established, variables can carry a tag labeling them as *owned* or *channeled.* If the binding of an *owned* variable is required, a NEED message must be sent to the owner of that variable; but to get the binding of a *channeled* variable it is only necessary to wait for another value to appear on the appropriate channel.

There are two types of channels, *incoming* and *outgoing.* A process can have at most *one* incoming channel per variable, except under the following conditions:

(1) The variable is owned by the process that has the multiple incoming channels;

(2) The incoming channels are all from children of the process; and

(3) Neither COMMIT nor QUIT has been received from any child.

There may be any number of outgoing channels for each variable. A process is responsible for copying to each of its variables' outgoing channels any bindings that it computes or that are received on the variables' incoming channels or in BIND messages.

Channeled variables, therefore, have a *channel list* rather than a suspension list; as incremental bindings for each variable are received, they are copied to all outgoing channels on that variable's list. Any process that spawns multiple processes to some non-ground component of the binding of a channeled variable, such as the tail of a list, becomes the *local owner* of that component, and is responsible for maintaining a local channel list for purposes of forwarding multiple copies of further bindings. Otherwise, channels are simply extended to new processes that will use the tail.

There are thus three ways in which channels and channeled variables can be created:

(1) If a process spawns multiple children to use a non-ground component of a channeled variable, that component is considered a new channeled variable and channels for it are established between the parent and the children.

(2) If a BIND for an owned variable is received and carries a partially bound structure (*i.e.*, a structure containing logical variables), a channel is established between the receiver and the sender, and the type of the variable is changed to channeled in both the sender and the receiver.

(3) If NEED messages are received for a variable known by the owner to be channeled, or if an owned variable with a non-empty suspension list is changed to a channeled variable, channels are established between the owner and all processes that have requested bindings, and the type of the variable is changed to channeled in the requesting processes.

Owned variables that are passed on to child processes are also tagged as owned in the child, so no channels are established until bindings are requested or (partially) supplied. When a process terminates, the closing of its incoming channels must be communicated to the processes at the other ends so that no further bindings will be output on those channels.

The decision of whether to create dynamic channels or to use messages for all communications must be based on the capabilities of the underlying system and the relative costs of creating and extending channels *vs.* sending one or more messages for every element in a list. Channel creation and extension may add considerable overhead to the spawning of new processes and to the implementation of NEED and BIND messages. However, if the cost of sending a message is high, that overhead is probably preferable to the large number of messages that must be passed in the absence of channels.

4. Implementation Strategy

As with the execution model, this implementation strategy is derived from that of Crammond [Cra86]. The changes that have been made are primarily those necessary to eliminate reliance on a global address space. It should also be noted that this strategy is directed solely at developing a PARLOG implementation; Crammond's strategy was intended to support a variety of committed-choice non-deterministic languages.

4.1. Problems with Crammond's Implementation

The primary difficulty with Crammond's model has already been mentioned, *i.e.*, its implicit reliance on shared (or, at least, globally addressable) memory. Although this allows many of the signals and state changes to be done very efficiently, it is not applicable to a fully distributed execution model. In addition, an attempt is made to address some other minor problems with Crammond's

implementation.

4.1.1. Performing KILL

Crammond advocates an indirect implementation of the KILL message, wherein the status and commit flags of a process are examined by its children or grandchildren under specific circumstances. The descendants then terminate if they discover the process to be dead or to be committed to a different goal. This is not only impossible in a distributed implementation, since no process can directly examine the control block of another, but it also has a serious drawback in that some infinitely recursive processes that should have been terminated could become "runaways."

In the indirect implementation, processes check their parents' status whenever they intend to commit, and check their grandparents' status whenever they intend to spawn a promoted child. This covers most cases where a process should be terminated, but does not always handle termination of siblings when a goal process has failed. For example, consider the following set of clauses:

```
p <- q , r.
q <- q & s.
etc.
```

In the clause *q*, the recursive call is made in the first part of a sequential conjunction; these recurrent subgoals are therefore *not* promoted. If goal *r* fails during the evaluation of *p*, *q* should be terminated, but until the (possibly infinite) recurrence completes, *q* will never receive the indirect KILL. For this reason, it may be necessary to consider use of a direct KILL even in implementations where it is not required.

4.1.2. Sequential Conjunctions

Crammond's model has difficulty with the sequential conjunction operator of PARLOG when parentheses are used to group sequential conjunctions. This is easily shown by a simple example. In the PARLOG clause

```
p <– (q & r) , (s & t).
```

the sequential evaluation of goals *q* and *r* should run in parallel with the sequential evaluation of goals *s* and *t*. Goal *r* should not have to wait for *s*, nor should *t* have to wait for *q*. However, the best that the model's two-step execution of clauses with sequential conjunctions can do is to run *q* and *s* in parallel, wait for both to finish, and then run *r* and *t* in parallel. This occurs because a grouping like "(q & r)" is really a single subgoal, but it is not possible to directly generate goal and clause processes for this construction.

The best solution to this problem is to require the compiler to eliminate such groupings by generating new predicates. For example, the clause above would be compiled into a set of clauses:

```
p <– seq1 , seq2.
seq1 <– q & r.
seq2 <– s & t.
```

It is then possible to generate goal and clause processes for the new predicates. The only complication of this scheme is that the compiler must make an analysis of the input and output arguments to the replaced subgoals, and generate a consistent set in the new goals. This should not prove to be particularly difficult, given the *mode* declarations of PARLOG.

4.2. Modified Strategy

The following discussion presents a modified implementation strategy suitable for the distributed execution model. In this discussion, it is assumed for simplicity that all communication takes place via messages, *i.e.*, that channels are not established, and that remote references are used to implement shared logical variables. The variations necessary to employ channels should be fairly obvious.

In order to easily describe the exchange of messages, the convention of giving each process a *postal address* has been adopted. Each processor can be considered a "post office," to which all messages are routed that are intended for delivery to one of the processes executing on that node; the specific process is selected once the message reaches its processor. All references to some process by another process are

via this postal address, even if both processes reside on the same processor. Of course, some optimization of the delivery could be done in the latter case.

Process States

As noted in the discussion of remote references, the *Suspended on Variable* state is replaced by *Suspended on Reference* (SR). Processes suspend when it becomes necessary to resolve a remote reference, in the same way that they suspend when waiting for an unbound variable to become bound. A process will become suspended at most once for each remote reference, because the reference will be replaced by the actual term when it is received.

Process Creation Messages

These messages are used to create new processes in the distributed system. They are listed here rather than in the execution model because they are specific to this implementation of process spawning.

CREATE

This is the only message that does not have another process as recipient. It is sent when a process is to be spawned, and carries all the information necessary for the creation of the new process. The CREATE message moves through the network until a processor decides to accept it, at which time the new process is created on that processor.

READY

This message is sent immediately by a newly created process to its parent process. If the receiver is dead (state D), or has previously received a COMMIT from some other process, a KILL is sent to the sender of the READY message; otherwise, the sender of the READY is added to the receiver's list of children.

Process Control Block

To support the fully distributed execution model, a number of changes have been made to the process control block as outlined by Crammond. These changes involve all fields whose use implies access to the control record of one process by another process. The most significant changes are the additions of the Grandparent Pointer and of explicit Variable List and Argument List fields. The Reference Count field has been replaced by a Reference List field to record active child processes.

Parent Pointer (PP)

The postal address of the parent of this process. When a new process is started, a READY message is sent to this address.

Grandparent Pointer (GP)

The postal address of the grandparent of this process, included to reduce the number of messages that must be sent when spawning a promoted process. This field is not needed by goal processes, and is left undefined when a promoted goal process is spawned.

Code Pointer (CP, or Instruction Pointer, IP)

If the state of this process is RE, this is a pointer to the instruction being executed.

Return Pointer (RP)

A pointer to the instruction at which this process will resume when awakened from suspension. (This pointer can be overlaid with CP; it is separated here for clarity.)

Reference List (RL)

A list of the postal addresses of all the children of this process. As long as this list is nonempty, this process control block cannot be deallocated.

Status Word (SW)

This field records the current state of the process, and has five possible values corresponding to the states RE, RQ, SR, SC, and D. Crammond's model has a simple "dead-or-alive" status flag, which is written by child processes to signal QUIT and read by child processes as an indirect implementation of KILL.

Commit Flag (CF)

This flag is set when the first COMMIT is received from any child; it is not used by clause processes. Once this flag is set, this process will ignore subsequent COMMIT messages and will respond to READY messages by sending KILL messages back.

Variable List (VL)

A list of the variables owned by this process. Each variable may have an associated suspension list, recording the postal addresses of processes that have requested a binding for the variable.

Argument List (AL)

This is a list of the argument terms on which this process was invoked.

Every process maintains its own process control block, making updates as instructions are executed and messages are received. No process can directly modify the control block of another process. Instructions are interrupted only by QUIT and KILL messages, which cause termination of the process, or by a suspension inherent to the instruction; hence no locking mechanism is required.

Instruction Set

Some significant changes have been made to Crammond's instruction set, with the goal of making the instructions directly executable. The instructions have also been modified to make them more suitable for use in a PARLOG system. The primary changes are the elimination of the *setup_args* instruction in favor of *spawn* instructions with variable arguments, the replacement of the two unification instructions with three more specific unifications, and the addition of instructions for list and structure manipulation. Also assumed, but not listed, are basic arithmetic operations.

The cases in which a process could become suspended on a nonvariable term are too numerous to list here, but the possibility of such a suspension should be kept in mind. Suspension

on variable terms is detailed because this applies regardless of the use of remote references.

spawn(IP, A_1, A_2, ..., A_n)

Used in goal processes to create new clause processes, and in clause processes to create goal processes that will not be promoted (*e.g.*, within guards). IP is the instruction pointer where the new process should begin, and A_1 through A_n are the arguments to the new process, if any. A CREATE message is broadcast, with the spawner's address given as the new process's Parent Pointer and the spawner's PP as the new process's Grandparent Pointer.

spawnl(IP, A_1, A_2, ..., A_n)

Spawn last. This is equivalent to spawn followed by waitc, *i.e.*, the spawner enters state SC. The process resumes at the next instruction when awakened.

spawnp(IP, A_1, A_2, ..., A_n)

Spawn promoted. Used in clause processes to create new goal processes. This instruction is equivalent to spawn, except that the spawner's Grandparent Pointer is given as the Parent Pointer of the new process, and the GP of the new process is undefined. The promotion is completed when the new process sends READY to the spawner's grandparent.

spawnpl(IP, A_1, A_2, ..., A_n)

Spawn last promoted. Instead of broadcasting a CREATE message, the spawner is transformed into the new process, in a manner similar to UNIX's[†] *execve*. QUIT is sent to the spawner's parent, the process's Grandparent Pointer is copied to its Parent Pointer, and the new process begins execution by sending READY to its new parent. Any variables owned by the parent are inherited by the new process, and the new process keeps the same postal address as the parent.

waitc

Puts the process into state SC if any children exist. The process resumes at the next instruction when awakened.

waitv(V)

If no binding for V is known, puts the process into state SR, and sends NEED(V) to the owner of V. The process resumes at the next instruction when awakened (by a BIND(V, T) message). The process continues immediately if a binding for V is known.

done

Terminates the process and sends DONE to its parent.

quit

Terminates the process and sends QUIT to its parent.

lunify(T_1, T_2)

Leftward unification; can only bind variables in T_1. If success would require a binding in T_2, the process is put into state SR and NEED is sent to the owner of the variable that would need to be bound. The process resumes at this instruction when awakened. If variables in T_1 are bound, BIND messages are sent to the owners of those variables.

tunify(T_1, T_2)

Test unify; cannot bind any terms in either argument. If success would require a binding, the process is put into state SR and NEED is sent to the owner of the variable that would need to be bound. The process resumes at this instruction when awakened (by a BIND message for the variable).

assign(V, T)

Assignment unification; binds the term T to the variable V and sends BIND(V, T) to the owner of V. The build and access instructions can also be used as special cases of assign.

build(V, functor, T_1, T_2, ..., T_n)

Constructs a new term and binds it to the variable V. The value of *functor* is a special code if the term to be constructed is a list or pair, otherwise it is the name of the structure to be

† UNIX is a trademark of Bell Labs.

built. T_1 through T_n are the component terms of the new term.

access(V, T, N)

The N*th* component of the term T is assigned to the variable V. If T is a list or a pair, N can be no greater than 2. The 0*th* component of a term is its functor.

switch(T, V_i, C_i, S_i, L_i)

Executes V_i if T is an unbound variable, C_i if T is a constant, S_i if T is a structure, or L_i if T is a list.

switchv(T, V_i, N_i)

A less general form of switch, added because many tests in PARLOG depend only on whether or not a term is a variable. Executes V_i if T is a variable, N_i in all other cases.

commit

Used in clause processes to signal success of all guard subgoals. Sends COMMIT to the parent of this process.

If any of the unification instructions fails, including *assign*, *build*, and *access*, the result is the same as if the process had executed a *done* instruction. For this reason, all unifications must take place in clause processes.

4.3. Examples

The following three examples correspond to those given by Crammond, for comparison with his implementation. It is assumed that the compile-time *preunification* optimization suggested by Crammond has been employed in generating the instructions for each predicate, *i.e.*, that any test for a non-variable argument which must occur for all clauses has been moved to the goal process, instead of being repeated in every clause process. This technique answers one of the objections to Crammond's implementation that was raised by Gregory [Gre85], *i.e.*, that the sequential unification of arguments could lead to unexpected behavior depending on which input arguments were bound and which were unbound. The unification here is still sequential, but by applying this test to the input arguments in the

goal process, consistent behavior is guaranteed: a predicate will always ensure that, before it succeeds or fails, input arguments whose structures are required are bound.

In all examples, goal arguments are referred to by A*n*, where *n* is an index into the process's Argument List, and owned (local) variables are referred to by X*n*, where *n* is an index into the process's Variable List. The first example is a simple *append* relation:

```
mode append(?, ?, ^).
append([a|b], c, [a|d]) <- append(b, c, d).
append([], x, x).
```

The code for the goal process is then

```
append:waitv(A1)
       switch(A1,_,spawnl(app2,A1,A2,A3),quit,spawnl(app1,A1,A2,A3))
       quit
```

where the underscore in the *switch* indicates a no-op for a case which cannot occur. Note that if the selected clause process succeeds, it will send QUIT to the goal process, which will then immediately terminate with DONE (success, for a goal); only if the clause fails will the *quit* (failure) instruction be executed.

The code for the two clause processes is

```
app1:  build(X3,list,X1,X2)
       lunify(X3,A1)
       commit
       build(A3,list,X1,X4)
       spawnpl(append,X2,A2,X4)
app2:  tunify(A1,[])
       commit
       assign(A3,A2)
       quit
```

Note the use of *build* as a special case of assignment in *app1*.

The *on_tree* relation, originally from Gregory [Gre85], demonstrates two important points. First, this predicate uses sequential clause selection, which requires special treatment in the goal process. Second, it has "bodiless" clauses, doing all their work in the guard; but input variables cannot be bound

until the guard has succeeded, so the clause processes also require special compilation.

```
mode on_tree(?, ^, ?).

on_tree(key, value, Tree(left, Node(key, value), right)) ;
on_tree(key, value, Tree(left, Node(key, value), right)) <-
      on_tree(key, value, left) : .
on_tree(key, value, Tree(left, Node(key, value), right)) <-
      on_tree(key, value, right) : .
```

The compilation of this relation is (in part):

```
on_tree: waitv(A3)
         switch(A3,_,quit,spawnl(on1,A1,A2,A3),quit)
         spawn(on2,A1,A2,A3)
         spawnl(on3,A1,A2,A3)
         quit
on1:     build(X1,Node,X2,X3)
         build(X4,Tree,X5,X1,X6)
         lunify(X4,A3)
         lunify(X2,A1)
         commit
         assign(A2,X3)
         quit
on2:     access(X1,A3,0)
         tunify(X1,Tree)
         access(X2,A3,1)
         switch(X2,spawnl(on_tree,A1,X3,X2),done,spawnl(on_tree,A1,X3,X2),done)
         commit
         assign(A2,X3)
         quit
```

The code for *on3* is identical to *on2* except for the index in the second *access* instruction. Note the use of *spawnl* in the *switch* in the *on_tree* goal process to handle the sequential selection of the second and third clause processes only after the first has failed. If the first should succeed, the execution of the goal would be terminated with success immediately after the *switch*.

The unification in *on1* of the third argument and a structure of the form *Tree(_,Node(_,_),_)* could be carried out in two ways. The way shown uses two *build* instructions to assemble a structure of the appropriate form, and a single *lunify* to make the test. Alternately, as demonstrated in part in *on2*, *access* could have been used to decompose the argument structure, and each functor then tested with *tunify* against a constant term. The advantages of the code used in *on1* are in the instructions that follow the unification, where judicious choice of variables in the *build*s allows the remaining unification

and the final assignment to be done without further structure accesses. If only the unification were to be done, the alternative code has two advantages: first, it aborts without doing unnecessary work if the outer functor is not *Tree*, and second, three accesses and two atomic tests are presumably faster than two constructions and a structure unification.

In both *on1* and *on2*, note that local variables have been used in place of the second argument until after the *commit*, at which time assignment to the second argument is done. Also, in *on2*, the test that the second element of the *Tree* is actually a *Node* has been omitted.

As a final example, this partial compilation of a sorting relation demonstrates the handling of a sequential conjunction within a clause. This relation is the same as the *sort-d* relation seen earlier, except for the sequential conjunction after the *partition* call.

```
mode sort(?, ^, ?)

sort([], list, list).
sort([hd|tail], sorted, rem) <-
      partition(hd, tail, list1, list2) &
      ( sort(list1, sorted, [hd|rem1]) ,
        sort(list2, rem1, rem) ).
```

The compilation of the second clause is of interest:

```
sort2:build(X1,list,X2,X3)
      lunify(X1,A1)
      commit
      spawnl(partition,X2,X3,X4,X5)
      build(X6,list,X2,X7)
      spawnp(sort,X4,A2,X6)
      spawnpl(sort,X5,X7,A3)
      quit
```

The use of *spawnl* in creating the *partition* process allows the first part of the sequential conjunction to complete before the recursive *sort* processes are spawned.

5. Conclusion, Current and Future Research

This paper has presented a model of execution for PARLOG suitable for use on a message-passing multiprocessor with no shared memory. Crammond's model for shared-memory multiprocessors has been

enhanced by adding the concept of variable ownership, the grandparent pointer to help the promotion of processes, and the distribution of process information.

Currently, this model is being used to develop a PARLOG implementation targeted for a network of INMOS transputers [INM86]. The transputer consists of a single chip containing a 32-bit microprocessor, a memory interface, and four asynchronous serial channels (links) that can send messages at 20 Mbits per second. These point-to-point links allow transputers to be connected together to form large networks at low cost.

PARLOG is particularly attractive for implementation on a transputer network because interprocess communication in PARLOG is similar to that in occam† (the base language of the transputer), which is based on the CSP model of communicating sequential processes [Hoa78,Hoa85]. However, PARLOG is at a higher level than occam in that occam handles only a static number of processes [Pou86], whereas PARLOG handles a dynamic number. For this reason, it has been decided that for the first version of the system, all communication among PARLOG processes will be via messages, rather than attempting to simulate dynamic channels.

The instruction set interpreter and message facilities are to be written in C,‡ with parts of the message interface in occam. Each transputer will have two transputer processes running in alternation: the post office, which handles messages, and the interpreter, which executes the instructions of the PARLOG processes running on this transputer. The interpreter and the post office share lists of runnable, suspended, and terminated PARLOG processes on the transputer. The interpreter is responsible for moving PARLOG processes from the active to suspended lists, and the post office is to return them to the active list when the message that "awakens" them is processed.

Future work needs to focus on problems of spreading and balancing the load among the processors in the system as well as the effect of network topology on the efficiency of the PARLOG system. Work on these for traditional OR-parallel Prolog [PaB87] may also have application for this PARLOG model.

---

† occam is a trademark of the INMOS Group of Companies.

‡ The entire system can be converted to C when a full compiler becomes available; the current compiler provides access to parallel communications (ALT) only through occam.

Also remaining is the implementation of the PARLOG *set* and *subset* primitives for OR-parallel and OR-sequential execution. However, as noted by Clark [ClG85], these can be duplicated by AND-parallel relations if a few additional primitives are available.

References

[ClG84] Clark, K. and Gregory, S., "PARLOG: Parallel Programming in Logic," Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, London, April 1984.

[ClG85] Clark, K. and Gregory, S., "Notes on the Implementation of PARLOG," *The Journal of Logic Programming*, 1985, pp. 17-42.

[ClG86] Clark, K. and Gregory, S., "PARLOG: Parallel Programing in Logic," *ACM Transactions on Programming Languages and Systems*, vol. 8, 1 (January 1986), pp. 1-49.

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

[Con83] Conery, J. S., "The AND/OR Process Model for Parallel Interpretation of Logic Programs," Ph.D. Thesis (Technical Report 204), University of California, Irvine, June 1983.

[CrM84] Crammond, J. A. and Miller, C. D. F., "An Architecture for Parallel Logic Languages," *Procedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pp. 183-194.

[Cra86] Crammond, J., "An Execution Model for Committed-Choice Non-Deterministic Languages," *Proceedings of the Third Symposium on Logic Programming*, Salt Lake City, September 1986, pp. 148-158.

[Fos87] Foster, M., "Parallel Graph Reduction Runtime Process Organization," PGR research group internal document, Oregon Graduate Center, Beaverton, Oregon, February 19, 1987.

[Gre85] Gregory, S., "Design, Application and Implementation of a Parallel Logic Programming Language," Ph.D. Thesis, Department of Computing, Imperial College of Science and Technology, London, September 1985.

[Hoa78] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, 8 (1978), pp. 666-677.

[Hoa85] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[INM86] INMOS, *Transputer Reference Manual*, INMOS Ltd., Bristol, UK, October 1986.

[PaB87] Pase, D. and Borgwardt, P., "Load Balancing Heuristics and Network Topologies for Distributed Evaluation of Prolog," Technical Report CS/E 87-005, Oregon Graduate Center, Beaverton, Oregon, April 1987.

[Pou86] Pountain, D., *A Tutorial Introduction to occam Programming*, INMOS Ltd., Bristol, UK, March, 1986.

[Wat86] Watson, I., "Reference Counting for Distributed Virtual Memory," talk at *Graph Reduction Workshop*, Santa Fe, NM, September 29 - October 1, 1986. Proceedings to be published in 1987 by Springer-Verlag.