EFFICIENT STREAM/ARRAY PROCESSING
IN LOGIC PROGRAMMING LANGUAGES

Kazumori Ueda
C&C Systems Research Laboratories
NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki 213 Japan

and

Takashi Chikayama
ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

ABSTRACT

The Concurrent Prolog predicate for merging n input streams is investigated, and a compilation technique for getting its efficient code is presented. Using the technique, data are transferred with a delay independent of n. Furthermore, it is shown that the addition and the removal of an input stream can be done within an average time of O(1). The predicate for distributing data on an input stream to n output streams can also be realized as efficiently as an n-way merge. The compilation technique for the distribute predicate can further be applied to the implementation of mutable arrays that allow constant-time accessing and updating. Although the efficiency stated above could be achieved by a sophisticated compiler, the codes should be provided directly by the system to get rid of the bulk of source programs and the time required to compile them.

1 INTRODUCTION

When we implement a large-scale distributed system in a parallel logic programming language such as Concurrent Prolog (Shapiro 1983) and PARLOG (Clark and Gregory 1984), the performance of the system will be influenced significantly by how efficiently streams as interprocess communication channels can be merged and distributed. This paper deals with implementation techniques of the predicates that merge many input streams and those which distribute data on a single input stream into multiple output streams.

The language we choose for the following discussions is Concurrent Prolog. However, the results obtained are applicable also to PARLOG. For readers unfamiliar with Concurrent Prolog, an outline of Concurrent Prolog is described in Appendix I.

This paper focuses on implementation on conventional sequential computers. Of course, to demonstrate the viability of Concurrent Prolog on parallel computers, the scope of discussion cannot be limited to sequential computers. However, even on a parallel architecture, it would be very likely for each processor to deal with multiple processes for the following reasons. First, the number of processes a user can create should not be limited to the number of processors available. Second, even if a lot of processors are available, the best way to allocate two processes which communicate intensively with each other and have little portions executable in parallel may well be to allocate them on the same processor. In that event, the techniques presented here will be directly applicable to communication within each processor.

1.1 Importance of Streams in Concurrent Prolog

Parallelism or coroutineing in Concurrent Prolog is realized by expressing individual processes via predicate calls or goals which are executed in AND-parallel, and interprocess communication via shared variables appearing as arguments. The shared variables express lists of data or messages flowing among (usually two) predicates: As program execution proceeds, the values of the lists are gradually instantiated to the end. The definition of a predicate is the specification of (the relationship between) values that the shared variables as its arguments can take, and a goal can be regarded as a process which processes the sequences of data represented by shared variables from the top downwards through tail recursion. We use the term ‘stream’ to refer to shared variables which are used in this manner.

Note that in Concurrent Prolog, ‘process’ and ‘stream’ are nothing but pragmatic concepts.

As is clear from the above explanation, communication with other processes is accomplished not by specifying their process names, but by instantiating (in the case of sending) or by checking (in the case of receiving) the streams which have already been laid between processes. Therefore, the efficiency of stream operations—sending, receiving, merging, and distributing—are of crucial importance.

1.2 Necessity of Dynamic, Multway Stream Merging and Distribution

Streams need not be merged or distributed if several processes are linearly connected by shared variables to perform pipeline processing. However, if there is a process that needs to receive data or messages from many other processes—e.g., a process that manages shared resources—a merging process must be put at the front-end:

\[
\begin{align*}
&\text{\texttt{p}\{G_1\}, \texttt{p}\{G_2\}, \ldots, \texttt{p}\{G_n\},} \\
&\text{merge}\{G, C_1, C_2, \ldots, C_k\}, \text{\textit{shared.resource}(C')}. \\
\end{align*}
\]

In order to accept messages from an indefinite number
of processes, it must also be possible to dynamically vary the number of input streams to be merged. In other words, if a process needs to communicate with a shared process, it must issue a request to the front-end merging process (by using other input streams or a 'request' stream), and set up a new input stream. Alternatively, a new stream could be laid by attaching a binary merge to one of the existing input streams, but a delay proportional to the number of communicating processes will arise if this method is repeatedly used.

As for message distribution, if it is done as broadcast, each process need only share the broadcast stream. However, if a process wants to communicate with another process to which none of its streams directly leads, communication must be enabled via the manager of the destination process. The manager process must appropriately distribute messages according to the destinations attached to the messages.

Again, it must be possible to dynamically change the number of processes to be managed.

1.3 Previous Research

(Shapiro and Mierowsky 1984) deals with the problem of merging an indefinite number of streams (henceforth the number of input streams will be denoted by n). They demonstrated:

(1) a method to ensure n-bounded waiting and a maximum delay of \(O(n)\) by using an unbalanced tree consisting of binary mergers, and

(2) a method to ensure n-bounded waiting and a maximum delay of \(O(\log n)\) by using a 2-3 tree (Aho et al. 1974) consisting of binary and ternary mergers.

The term 'n-bounded waiting' was defined by them to mean that any message arriving at the merging process will be overtaken by no more than n input messages from other streams.

The delay of \(O(n)\) in Method (1) above is probably unacceptable when n is large enough and the traffic is heavy. This method may be practical, however, in cases of essentially costly communication such as interprocess communication in multi-processor environments.

Method (2) is a major improvement over (1) in terms of delay. In procedural languages, however, the delay of interprocess communication does not depend on the number of sensors as long as it is simulated on a sequential computer. Therefore, also in logic programming languages, it is desirable to achieve a constant-time delay.

(Kusalik 1984) also deals with bounded-wait merging of n streams. He showed a method to ensure bounded-wait merging without resort to operational characteristics of an underlying machine or interpreter. One of his solutions has \(O(\log n)\) delay, but the number of input streams cannot be changed. The other solutions can merge indefinite number of streams, but they are inefficient.

The above two papers concentrate on how to program n-ary merge having the desired properties. On the other hand, this paper is devoted to how to compile a rather intuitive n-ary merge program.

Geilerber (Geilerber 1984) discusses the suitability of Concurrent Prolog for the description of multi-process systems. He concludes that interprocess communication using merge networks are 'not only bulky but unduly constricting'. It should be noted, however, that this criticism is not from the viewpoint of descriptive power or efficiency.

2 OBJECTIVES

We have the following two objectives:

(1) When the number of input streams n is fixed, to realize on a sequential computer n-ary merge and distribute with a maximum delay of \(O(1)\).

(2) To extend the solution to (1) to the case where n varies dynamically.

It is clear that (1) cannot be accomplished through the combination of binary and ternary mergers or distributors. The predicates must process all messages directly at the top level:

\[
\text{merge}(X_1, X_2, \ldots, X_n) \\
\text{merge}(Y_1, Y_2, \ldots, Y_n) \\
\text{distribute}(X_1, X_2, \ldots, X_n) \\
\text{distribute}(Y_1, Y_2, \ldots, Y_n)
\]

If these predicates are interpreted, the time for tail recursion can be proportional to the size of each clause \(O(n)\). However, if compiled, these predicates promise to yield higher efficiency, as will be discussed in 3.1.

When considering the above n-ary merge and distribute, we cannot define 'delay' as the depth of a tree. So we will define the word 'delay' as:

* the time passed from the arrival of a message at a goal in an input-wait state until the original input-wait state is restored by tail-recursion, during which the message is transferred to output streams.

The delay is calculated by the number of primitive operations which can be accomplished within a unit time on a sequential computer.

2.1 Outline of Sequential Implementation of Concurrent Prolog

Examples of Concurrent Prolog implementation on a sequential computer include (Shapiro 1983) and (Niis 1984), but both are interpreters. Here, we assume the implementation of a compiler which follows the guidelines stated in (Shapiro 1983 (unpublished)). What follows is a brief explanation of the process management technique.

The descriptors of conjunctive goals make up a circular list called an AND-loop, and the descriptors of uncommitted clauses composing a predicate make up a circular list called an OR-loop (Figure 1).

Each element of an AND-loop is, until it is committed, the parent of an OR-loop comprising candidate clauses;
Figure 1. AND-loop and OR-loop

(The Program)
\[ (C0) := p. \]
\[ (c1) p := q11, q12 \mid b11, b12. \]
\[ (c2) p := q21, q22 \mid true. \]

Figure 2. Tree Structure Constructed by AND/OR-loop

Figure 3. Changes and Their Propagation of AND/OR loop
after commitment, it is replaced by a doubly-linked list representing goals of the body. If the body is empty, the element of the original AND-loop disappears. The parent of an AND-loop, having lost all elements, is considered a success. On the contrary, failure of any AND-loop element is the failure of the parent (Figure 3).

Each element of an OR-loop represents a candidate clause which has not been committed yet, and is the parent of the AND-loop whose elements represent goals of the guard. The success of an OR-loop element implies the commitment of the corresponding clause. On the contrary, when some element of an OR-loop fails, that element simply disappears. The parent of an OR-loop, having lost all elements, is considered a failure (Figure 3).

The system has a queue called Process Queue in which leaf elements of a tree composed of AND/OR-loops (i.e., elements which are not parents of other loops: see Figure 2) await scheduling. In unification, a suspended clause due to some read-only variable is added onto the waiting list attached to that read-only variable instead of waiting in Process Queue. This clause will be re-scheduled when the read-only variable is instantiated.

One possible optimization of the above method is to perform the unification of a clause head and the execution of simple goals in a guard as an indivisible operation. We call this immediate check. If an immediate check succeeds, we can avoid creating an OR-loop. In other cases, an OR-loop is created for those clauses which have suspended during immediate check and which have succeeded in immediate check but have complex guards, and they go into a wait state.

3 IMPLEMENTATION OF THE MERGE PREDICATE

3.1 Examination of n-ary merge

The n-ary merge can be expressed by n clauses of the following form if one ignores the base cases for termination which will be dealt with in Section 3.3.5.

- The kth clause:
  \( \text{merge}(X | Y^k|, X_1, \ldots, [X | X_h], \ldots, X_m) \Rightarrow \text{merge}(Y, X_1, \ldots, X_h, \ldots, X_m). \)

This predicate has the following characteristics.

1. To see if the kth clause is selectable, one need only test the unification of the kth and the (n-th) arguments (henceforth we number the arguments starting with 0).

2. Upon the tail recursion employing the kth clause, only the 0th and the kth arguments change compared with the original call. Therefore, the argument list of the tail-recursive call can be made by slightly modifying that of the original call.

3. When all clauses are in a wait state and one of the argument variables is instantiated, there is only one clause (or two, even including the base case) which needs to be re-examined.

Now we will consider tail recursion. The arguments which do not change by tail recursion have the general property that they do not alter the wait condition of each of the clauses. Suppose that a predicate is called, that its nth clause is not selected due to the suspension (or failure) of the unification of the nth argument, and that the kth clause is selected instead. In this case, even after the tail recursion, the unification of the kth argument of the nth clause should suspend (or fail) provided:

- the kth argument of the nth clause does not change by tail recursion, and
- the read-only variable that suspended the unification of the kth argument of the nth clause does not become instantiated by the unification of other arguments of the nth clause.

If we state this in terms of n-ary merge, we get the following:

4. Upon the tail recursion employing the kth clause, the following clauses become new candidates:
   (a) the kth clause itself
   (b) clauses which were candidates in a previous call but have not been examined
   (c) clauses which need no longer suspend as the result of the instantiation of read-only variables.

Possibility (c) does not exist under normal circumstances, so we can ignore it. Possibility (b) refers to the clauses that have been 'carried over', so that once they are examined, they will either become non-candidates (by suspension or failure) or they will be selected and again become candidates after tail recursion. Therefore, the average number of clauses to be checked after each tail recursion does not depend on the total number of clauses.

From the above considerations, we can conjecture that n-ary merge can process each message within a constant time. Note that the implementation technique of sequential Prolog that takes advantage of the characteristics (1) and (2) appears in (Warren 1980).

3.2 Implementation Technique for the Fixed-Arity Merge

To efficiently implement n-ary merge, the following are necessary.

1. Even if all clauses suspend, an OR-loop (having O(n) elements) is not created for them, and they are made to wait at the predicate-call level.

2. The argument list is re-utilized.

3. In order to prevent examination of clauses not worth examining, candidate clauses are managed within the process descriptor (descriptor of the goal).

The implementation technique of predicates that follows these guidelines is described below in Sections 3.2.1 to 3.2.3. Since the description is general, it is applicable to predicates other than merge as long as they have no guards. Hereafter, the number of clauses composing the predicate will be denoted by \( M \), and the number of arguments by \( N \).
3.2.1 Configuration of a Process Descriptor

A process descriptor has the following items.

(1) **AND Brothers**: Two pointers for constructing an AND-loop.

(2) **Process Queue Pointer**: A pointer for designating the next element in a Process Queue.

(3) **Candidate Queue**: A queue of candidate clauses of the call managed by the current process descriptor. M elements.

(4) **Clause States**: An array indicating whether each clause is in the candidate, suspend, or fail state. M elements.

(5) **Clause Backward Pointers**: An array of pointers for designating entries on the waiting list of read-only variables that suspended unification. M elements (one element for each clause). Each pointer is meaningful if and only if the corresponding Clause State is suspend.

(6) **Suspend/Fail Table**: The reasons why a particular clause was not selected can be attributed to some of the caller's arguments. Thus, if these arguments change upon tail recursion, that clause may become selectable. Therefore, a pair of variables (c, k), where c is the number of the suspended or failing clause and k is the number of the argument that may be the cause, is maintained. This table must enable efficient
- sequential retrieval of elements containing c, and
- deletion of elements containing k.

For example, the structure shown in Figure 4 fulfills this condition. The maximum number of elements depends on the process; in the case of `merge`, it is \(O(N + M) = O(n)\).

(7) **Fail Count**: The total number of classes that cannot be selected for the current call.

(8) **Program Code**: A pointer to the predicate's code.

(9) **Argument List**: N elements.

3.2.2 Operations

A. Creation of a Process Descriptor

When a predicate is newly called (i.e., not as a tail recursion), the area for the process descriptor is allocated and its entries are set up as follows:

- all clauses are entered in Candidate Queue (3),
- all Clause States (4) are set to candidate,
- all Clause Backward Pointers (5) are left undefined,
- Suspend/Fail Table (6) is cleared,
- Fail Count (7) is set to 0, and
- Program Code (8) and Argument List (9) are set up.

The completed process descriptor is entered in the AND-loop by appropriately modifying AND Brothers (1) of this and neighboring goals. It is also entered in Process Queue by making it designated by the last element's Process Queue Pointer (2).

B. Selection of a Clause

B-1. If Candidate Queue is not empty, instructions for unifying the arguments of the first candidate (say the cth clause) and the arguments of the caller (Argument List of the process descriptor) are executed. In the case of `merge`, only the instructions for the 0th and the cth arguments are executed.

- If this succeeds, the body is executed (see D).
- If this fails,
  (1) the generated binding is undone,
  (2) Fail Count is incremented by 1,
  (3) Clause State of the clause is set to fail,
  (4) Suspend/Fail Table is updated (cf. 3.2.3), and
  (5) other candidate clauses are tested.

- If this suspends,
  (1) the generated binding is undone,
  (2) Clause State of the clause is set to suspend,
  (3) Suspend/Fail Table is updated,
  (4) the pair \((p, c)\), where \(p\) is the pointer to the process descriptor and \(c\) is the number of the clause, is entered in the waiting list of the read-only variable that caused the suspension,
  (5) Clause Backward Pointer for the clause is made to point to the pair entered in (4), and
  (6) other candidate clauses are tested.

B-2. If Candidate Queue is empty and Fail Count is equal to \(M(=\text{the number of clauses})\), the goal ends with failure. Otherwise, execution of the current goal is suspended.

C. Instantiation of Read-Only Variables

When a read-only variable is instantiated, the following is done for each entry \((p, c)\) in the waiting list of that read-only variable.

- The following is done for the process descriptor designated by \(p\).
  (1) Clause State of the cth clause is set to candidate, and \(c\) is entered in Candidate Queue.
  (2) All elements of the form \((c, \_\_\_\_)\) ('\_' means 'don't care') are deleted from Suspend/Fail Table.
  (3) This process descriptor is entered in Process Queue.

D. Execution of the Body

If a recursive call is contained in the body of the committed clause (say the cth clause), the following tasks
are done.

(1) Assume that the arguments of the head and the arguments of the recursive call differ in the $k_1$th, $k_2$th, ..., $k_j$th arguments. For each $k_i$ (i=1, ..., j), the following are done.

- Elements of the form $(d, k_i)$ are searched from Suspend/Fail Table, and for each d, the following are done.
  - If Clause State of the $i$th clause is fail, Fail Count is decremented by 1. If it is suspend, the entry of the waiting list pointed to by Clause Backward Pointer is eliminated.
  - Clause State of the $i$th clause is set to candidate, and d is entered in Candidate Queue.
  - Elements of the form $(d, -)$ are deleted from Suspend/Fail Table.

- The $k_i$th element of Argument List is rewritten.

(2) The $i$th clause is entered in Candidate Queue.

(3) Clause selection (cf. B) takes place.

If there is no recursive call, the area for the original process descriptor can be released after the pointers from the waiting lists of read-only variables are eliminated. However, there are cases in which this area can be reutilized for optimization (cf. 3.4).

### 3.2.3 Management of Suspend/Fail Table

If the $i$th clause of n-ary merge is called as follows,

$$\text{un} \leftarrow \text{merge}(Y_1, \ldots, X_i, \ldots).$$

unification of the $i$th argument suspends. In this case, the cause of suspension lies only in the $i$th argument of the caller; even if another clause were selected and tail recursion took place, this would not remove the cause. However, we cannot always attribute the suspension or failure of the unification of the $i$th argument only to the $i$th argument. Consider the following example:

$$\text{un} \leftarrow \text{merge}(Y_1, [0 | Y_2], [0 | Z_1], \ldots, [2 | X_2], \ldots).$$

If unification is done from the left, unification of the $i$th argument fails, but we should attribute the cause also to the $i$th argument. Actually, if the first clause is selected and tail recursion takes place, the $i$th clause immediately becomes selectable.

To generalize, when the unification of the $k_i$th argument of the $i$th clause of suspends or fails, all arguments (numbered $k_1$, ..., $k_i$, ..., $k_j$) related to the $i$th argument in the $i$th clause should be entered in Suspend/Fail Table in the form $(c, k_i)$.

Here, the term $A$ is related to $(b)$ (henceforth denoted by $R_b$) the term $B$ if and only if there are variables within $A$ which are related to variables within $B$, and the variable $V_1$ is related to the variable $V_2$ means that $V_1$ and $V_2$ are related by the reflexive transitive closure of the following relation $R_v$.

- Relation $R_v$: both variables appear together in a goal of the guard (if the guard is empty, $R_v$ is the transversity of the variables).

**Example:** For the $i$th clause of n-ary merge, the quotient $A/R$ of the set of arguments $A$ by $R_i$ is

$$\{0, c\}, \{1\}, \ldots, \{c-1\}, \{c+1\}, \ldots, \{n\}.$$

For the clause

$$p(i, j, k, l, m) \leftarrow a(i, j), b(j, k), c(l, m), \text{true},$$

we get

$$\{0, 1, 2\}, \{3, 4\}.$$

However, to efficiently implement n-ary merge, the above rules for updating Suspend/Fail Table must be slightly modified. If $(0, c)$ is entered in Suspend/Fail Table when the $i$th clause suspends, the $i$th clause will be returned to Candidate Queue even by the tail recursion of another clause and the desired efficiency is not achieved. However, the $i$th argument usually does not cause suspension, and in this case, $(0, c)$ need not be entered in Suspend/Fail Table. Therefore, in cases of suspension where

(1) the $k_i$th argument of the caller is a read-only variable (viewed at execution time) and

(2) the $k_i$th argument of the head is a non-variable term (viewed at compile time),

only $(c, k_i)$ should be entered in Suspend/Fail Table. This is all right because the cause of suspension is clearly not in the other arguments related to the $i$th argument. The number of elements that are simultaneously entered in Suspend/Fail Table does not exceed

$$\sum_{\text{clauses}} \left(\text{maximum size of the elements of } \frac{\text{set of arguments} \cap R_i}{R_i} \right).$$

In the case of n-ary merge, this value is $O(n)$.

### 3.3 Properties of the Fixed-Arity Merge

We will now examine the properties of n-ary merge compiled using the technique presented in 3.2. The existence of base-case clauses will not be considered here. It will be discussed later in 3.3.5.

#### 3.3.1 Space Efficiency

The size of each item of a process descriptor other than Suspend/Fail Table is clearly no greater than $O(n)$, and the size of Suspend/Fail Table is $O(n)$, as indicated in 3.2.3. Therefore, the size of each process descriptor is $O(n)$. The size of the program code will be discussed in 3.3.4.

#### 3.3.2 Time Efficiency

A. The generation of process descriptors: $O(n)$, but this need only be done once at the beginning.
B. Unification: The time required for the unification of the head of each clause is \(O(1)\), because unification must be attempted for no more than two arguments. If a data structure such as the one shown in Figure 2 is assumed, the time required for the tasks accompanying suspended/failed unification (i.e., updating of Suspend/Fail Table and the waiting lists of read-only variables) is also \(O(1)\).

C. Instantiation of a read-only variable: \(O(1)\) for each task.

D. Tail recursion: When the \(c\)th clause is selected, the \(0\)th and the \(c\)th arguments change. However, as long as \(\text{merge}\) is used in a usual manner, the \(0\)th argument will not be the cause of wait or failure, and the only clause waiting at the \(c\)th argument is the \(c\)th clause itself. Consequently, the only new candidate is the \(c\)th clause. Furthermore, only two entries of Argument List used be rewritten. Therefore, the overall time required is \(O(1)\).

The above shows that the time required for processing a message reaching \(n\)-ary \(\text{merge}\) in an input-wait state does not depend on \(n\).

3.3.3 Order of Clause Checking

Individual clauses of \(n\)-ary \(\text{merge}\) are checked in the order they are entered in Candidate Queue. Since a selected clause is reinserted at the tail of the queue, \(n\)-bounded waiting is achieved. Moreover, suspended or failing clauses are not in Candidate Queue, so they do not influence the efficiency.

3.3.4 Program Size

The codes for operations \(A\) and \(C\) in 3.2.2 is common to all predicates and have the size of \(O(1)\). The size of the code for \(n\)-ary \(\text{merge}\) is \(O(n)\), because the code for each clause describes operations \(B\) and \(D\), whose size is \(O(1)\).

However, since the codes for individual clauses are almost the same, they can be parameterized with respect to the clause numbers. If this is done, the code size for the whole predicate is drastically reduced to \(O(1)\).

This parameterization could be accomplished by a sophisticated compiler capable of detecting similarities among the clauses. However, even if such a compiler were employed, it would not reduce the size of the source program \(O(n^3)\) and the time required for compilation. Furthermore, there may be only a few programs which can benefit from this optimization. Considering all these things, the most realistic approach is to let the system provide the code for \(n\)-ary \(\text{merge}\).

Now we have \(n\)-ary \(\text{merge}\) at a code size of \(O(1)\). This, however, is still unsatisfactory. The system has to provide \(n\)-ary \(\text{merge}\) for every \(n\). If these were to be provided individually, the amount of code would be \(O(n_{max}^3)\), \(n_{max}\) being the maximum value of \(n\).

However, here again, drastic optimization is possible. Because the code for \(n\)-ary \(\text{merge}\) remain almost the same if \(n\) changes, it can be parameterized with respect to \(n\). This being done, the amount of code for merging any number of inputs becomes \(O(1)\).

Note that it is mandatory that these codes be provided by the system, because the size of the corresponding source program is \(O(n_{max}^3)\).

3.3.5 Base Case

To terminate the call of \(\text{merge}\), a clause describing the base case or termination condition must be carefully supplied. The clause

\[
\text{merge}([], [], \ldots, []).
\]

is logically correct, but it cannot be efficiently processed by the above implementation technique—unification must be performed for every argument. An alternative solution uses \(\text{otherwise}\) construct:

\[
\text{merge}([], [], \ldots, []) \leftarrow \text{otherwise} \mid \text{true}.
\]

An \(\text{otherwise}\) goal in a guard succeeds if and when all other guards fail (Shapiro and Takeuchi 1983). Implementation of this construct is simple: a clause containing \(\text{otherwise}\) in its guard should be put into Candidate Queue only after \(\text{Fail Count}\) reaches the number of clauses not containing \(\text{otherwise}\). With \(\text{otherwise}\), the base-case clause retains the efficiency of the predicate.

3.4 Dynamic Change of the Number of Input Streams

A fixed-arity \(\text{merge}\) predicate is useful only when the number of inputs is statically known. We will now expand this to allow the addition of new streams and the removal of terminated streams. The program shown below has an additional (the \((-1)\)th) argument for accepting requests of new input streams.

- The \(4\)th clause (transform)
  \[
  \text{merge}([S, |X_1|, Y_1, \ldots, |X_n|, X_1, \ldots, X_n] :- \text{merge}([S, Y_1, X_1, \ldots, X_n, X_{n+1}]).
  \]

- The \(5\)th clause (addition)
  \[
  \text{merge}([X_{n+1} | S, Y_1, X_1, \ldots, X_n] :- \text{merge}([S, Y_1, X_1, \ldots, X_n, X_{n+1}]).
  \]

- The \((n-k)\)th clause (removal)
  \[
  \text{merge}([S, X_1, X_2, \ldots, X_{n-k}] :- \text{merge}([S, X_1, X_2, \ldots, X_{n-k}]).
  \]

- Base Case
  \[
  \text{merge}([], [], \ldots, []).
  \]

The clauses for stream addition and removal are not tail recursive. However, if process descriptors for the goals in the bodies can be constructed by slightly modifying the original ones, it will be much more efficient than to create ones from scratch.

In Concurrent Prolog, process descriptors must be managed by a general memory management technique, not by a simple stack scheme. Here we will assume that the Buddy system (Knuth 1968) is employed. The site of each partitioned area will then be a power of two, and each process descriptor is created in one of these areas. When it
is created, its fields must be placed according to the size of the area allocated so that the cost of relocation with the addition and removal of streams is minimal. Then, even if the number of inputs changes, most of the existing information need not be moved as long as the same area can accommodate the new descriptor.

Here we will show the operations to be performed when the (−n)th to 0th clauses are selected and the process descriptor can be reused. When considering the reuse of process descriptors, unused must be added as one of the possible states that Clause State can take, and when the area for Clause State is allocated, the unutilized portion should be filled with unused’s.

A. When the 0th Clause is Selected and a New Stream is Added

(1) (Operations accompanying the addition of the \((n+1)th\) clauses) If Clause States of the \((n+1)th\) and the \(-(n+1)th\) clauses are not candidate, they are set to candidate and those clauses are entered in Candidate Queue.

(2) The 0th clause is entered in Candidate Queue.

(3) The \(-(n+1)th\) argument of Argument List is updated.

(4) The program code is replaced (If the program is parameterized with respect to \(n\), only the parameter value is replaced).

B. When the \(-(n+1)th\) Clause (\(n>0\)) is Selected and an Empty Stream is Removed

(1) (Operations accompanying the change of the nth argument) Elements of the form \((e, c)\) (only \(c, c\) can exist, if any) are retrieved from Suspended/Paused Table. For each \(e\), the following is done.

• If Clause State of the \(c\)th clause is fail, Fail Count is decremented by 1. If it is susp, the entry in the waiting list pointed to by Clause Backward Pointer for the \(c\)th clause is deleted.

• Clause State of the \(c\)th clause is set to candidate, and \(c\) is entered in Candidate Queue.

• Elements of the form \((e', c)\) (only \(c, c\) can exist, if any) are deleted from Suspended/Paused Table.

(2) (Operations accompanying disappearance of the \(n\)th clauses)

• If Clause State of the \(n\)th clause is fail, Fail Count is decremented by 1. The same is done for the \(-(n-1)th\) clause.

• Elements of the form \((e, n)\) are deleted from Suspended/Paused Table.

• (Nothing is done with the \(\pm n\)th clauses in Candidate Queue. When they are dequeued, nothing is due other than to change their Clause States to undefined.)

(3) The \(-(n+1)th\) clause is entered in Candidate Queue.

(4) The \(c\)th argument of Argument List is updated.

(5) The program code is replaced. It is clear that both \(A\) and \(B\) can be accomplished within a constant time.

If the area for the current process descriptor cannot be reused to add a new stream, it is necessary to allocate a new area of twice the size and to move to that area. On the contrary, if it becomes possible to express the process descriptor with half the size of the current area (by the repeated removal of streams), the process descriptor can be packed and the unused area collected can be freed. These operations are shown below.

\(A'\). Addition of Streams Entailing Moving to a New Area

(1) An area twice the size of the current process descriptor area is allocated.

(2) All items of the original process descriptor are copied.

(3) The entries designated by all meaningful Clause Backward Pointers (i.e., one for suspended clauses) are made to point to the new area.

(4) The operations described above in \(A\) are done.

\(B'\). Deletion of Streams Entailing Compaction

(1) The operations described above in \(B\) are done.

(2) Candidate Queue is examined and the \(\pm n\)th clauses are deleted, if any.

(3) The original process descriptor is packed in the top half of the current area.

(4) The bottom half of the area is released.

We will now consider the time complexity of \(A'\) and \(B'\). If the time needed for memory allocation and release is ignored, both \(A'\) and \(B'\) can be done within a time proportional to \(n\). The time complexity of memory allocation and release by Buddy system is

\[O(\log(n))\]

This value, however, is determined only by the execution environment of the program, which is independent of \(n\). Therefore, if the execution environment is fixed, the time needed for \(A'\) and \(B'\) is \(O(n)\).

In order to add and remove streams within an average time of \(O(1)\), it must be guaranteed that the frequency of doing operation \(A'\) or \(B'\) is at most once every \(O(n)\) times. This is easily achieved by doing \(B'\) only when it becomes possible to represent the process descriptor with (for example) one-fourth of the current area.

4 IMPLEMENTATION OF THE DISTRIBUTE PREDICATE

For the implementation technique of the distribute predicate, only outlines will be presented here.

4.1 Distribution to a Fixed Number of Output Streams

The predicate distribute with \(n\) output streams is expressed by \(n+1\) clauses of the following form:

- The \(k\)th clause

\[
\text{distribute}([k, X] \mid X_s), Y_1, \ldots, [X \mid Y_h], Y_n) \\
\Rightarrow \text{distribute}(\quad X_s), Y_1, \ldots, Y_h, Y_n).
\]

- The \(0\)th clause

\[
\text{distribute}([k, 1], Y_1, \ldots, [1, Y_n].
\]
First, we will consider the situation where there is no wait. Random accessing of clauses must be implemented because, if the 1st to nth clauses were individually checked, the time complexity would be $O(n)$. The DEC-10 Prolog compiler (Warren 1977) generates a code that selects clauses using the hash value of the principal functor of the first argument. However, this is inadequate for stream-oriented programming. In the case of distribute, hashing by the tertiary functor (a functor of the third level) of the first argument is necessary to select a clause within a constant time.

Next, as we did with merge, we will consider how to achieve the code size of $O(1)$. Parameterization of the codes of each clause is of course necessary. In the case of distribute, we should further make use of the fact that clauses can be selected by simple indexing which does not involve hashing: a hash table requires an area of $O(n)$.

What if there is a wait? In usual situations, the cause of wait is the 0th argument. In this case, if the 1st to nth clauses all individually go into wait, the desired efficiency cannot be achieved. Those clauses should always be managed together: not only when indexing, but also while waiting. In other words, they should be entered in the waiting lists of read-only variables as a cluster of clauses. When their suspension is released, the appropriate clause should be selected by indexing.

4.2 Dynamic Change of the Number of Output Streams

As in the case of merge, dynamic change of the number of output streams is important. This can be implemented by adding the following clauses:

- Addition
  
  distribute([prol(Y_n+1) | X], Y_1, ..., Y_n)
  
  distribute( X, Y_1, ..., Y_n, Y_{n+1}).

- Deletion
  
  distribute([retract | X], Y_1, ..., Y_{n-1}, Y_n)
  
  distribute( X, Y_1, ..., Y_{n-1}).

In order to efficiently change the number of output streams, a method similar to the one described for merge in 3.4 can be applied.

5 APPLYING IMPLEMENTATION TECHNIQUE OF DISTRIBUTION PREDICATES TO MUTABLE ARRAYS

The lack of mutable arrays (arrays of rewrutable elements) is often mentioned as one of the problems of Prolog. Of course, arrays can be simulated by assert and retract, but such arrays are not logical arrays. One direction to realize logical arrays is to make a correspondence

- Arrays: data of the array type
- Operations on arrays: predicates having array arguments

and to gain efficiency by a dedicated data structure. However, it is also possible to make the following correspondence:

- Arrays: goals (processes)
- Operations on arrays: messages in streams by the program

array(n, S) :- array(S, X_1, ..., X_n).
array([read(k, X), S], X_1, ..., X_k, ..., X_n) :- array(S, X_1, ..., X_n).
(array[k = i, ..., n], for k = i, ..., n).
array([write(k, X), S], X_1, ..., X_k, ..., X_n) :- array(S, X_1, ..., X_n).
(array[k = i, ..., n], for k = i, ..., n).

This is a rather natural solution if we regard arrays as mutable objects. This program has properties very similar to distribute, and if the implementation technique for distribute is used, constant-time accessing and updating is realized. It is also possible to add clauses for inquiring and/or changing the number of elements. Note that all transactions with an array object are done through the argument S of the binary array predicate; a programmer need not have direct access to each element.

6 CONCLUSIONS AND FUTURE WORKS

The properties of n-ary merge written in Concurrent Prolog were investigated and an implementation which transfers each message with a delay independent of n was presented. Furthermore, it was shown that an input stream can be added and removed within an average time of $O(1)$. With respect to n-ary distribute also, outlines for an implementation as efficient as merge were presented. Mutable arrays that allow constant-time accessing and updating were shown to be realizable by the same implementation technique as that for distribute.

However, it was concluded that these predicates should be supported directly by the system. If the system provides them, merge and distribute for all arities can be realized with the constant-size code. On the other hand, it is unrealistic to obtain the code by compiling a source program provided by the user, not from the viewpoint of the efficiency of the code obtained, but from the viewpoint of the bulk of the source program and the time needed for compilation. Nevertheless, it is favorable in many respects (e.g., for the construction of programming systems) that the semantics of the system-supplied code is expressible as a Concurrent Prolog program.

The suggested technique for the implementation of n-ary merge has a problem that it does not work efficiently when a bounded buffer (Takeuchi and Furukawa 1983) is connected to the output stream. However, it is expected that this problem can be solved by improving clause wait and scheduling.

The most important future tasks are to describe large-scale systems in Concurrent Prolog, to estimate the cost of interprocess communication, and to confirm the usefulness of the suggested capabilities. It is also important to consider an efficient implementation of interprocess communication in parallel environments.
ACKNOWLEDGMENTS

The authors thank to Katayua Takosaki, Masahiro Yamamoto, Kaschio Fuchi, and Kouichi Furukawa for providing a stimulating place in which to work. Thanks are also due to Shud Shapiro for offering them hints to embark on this study, as well as to Akitaka Takeuchi for valuable suggestions.

APPENDIX I

The outline of Concurrent Prolog is given below by quoting (Shapiro and Takeuchi 1980).

A. Syntax

A Concurrent Prolog program is a finite set of guarded-clauses. A guarded-clause is a universally quantified logical axiom of the form

$$A : = G_1, G_2, \ldots, G_m | B_1, B_2, \ldots B_n,$$  \( m, n \geq 0 \)

where the \( G_i \)'s and the \( B_i \)'s are atomic formulas, also called unit goals. \( A \) is called the clause's head, the \( G_i \)'s are called its guard, and the \( B_i \)'s its body. When the guard is empty the commit operator \( '!' \) may be omitted. Clauses may contain variables marked read-only, such as \( X' \). We follow the Prolog 10 syntax conventions: constants begin with a lower-case letter, and variables with an upper-case letter. The special binary term \( [X | Y] \) is used to denote the list whose head (car) is \( X \) and tail (cdr) is \( Y \). The constant \( [] \) denotes the empty list.

B. Semantics

Concerning the declarative semantics of a guarded clause, the commit operator reads like a conjunction: \( A \) is implied by the \( G_i \)'s and the \( B_i \)'s. The read-only annotations can be ignored in the declarative reading.

Procedurally, a guarded-clause functions similar to an alternative in a guarded-command. To reduce a process \( A \) using a clause \( A_1 : = G | B \), unify \( A \) with \( A_1 \), and, if successful, recursively reduce \( G \) to the empty system, and, if successful, commit to that clause, and, if successful, reduce \( A \) to \( B \).

The reduction of a process may suspend or fail during almost any of these steps. The unification of the process against the head of the clause suspends if it requires the instantiation of variables occurring as read-only in \( A \). It fails if \( A \) and \( A_1 \) are not unifiable. The computation of the guard system \( G \) suspends if any of the processes in it suspends, and fails if any of them fails.

The commitment operation is the most delicate, and grasping it fully is not required for the understanding of the example programs in this paper. It suffices to say that partial results computed by the first two steps of the reduction—unifying the process against the head of the clause, and solving the guard—are not accessible to other processes in \( A \)'s system prior to the commitment, and that after commitment all the Or-parallel attempts to reduce \( A \) using other clauses are abandoned.

The reduction of all processes in a system can be attempted in parallel, and similarly the search for a clause to reduce a process. Two restrictions prevent an all-out parallelism. Regarding Or-parallelism, only the guards are executed in parallel. Once a guard system terminates, the computation of other Or-parallel guards are aborted. Regarding And-parallelism, read-only annotations can enforce rather severe constraints on the order and pace in which processes can be reduced.

REFERENCES


