TM-1125

# Implementing Streams on Parallel Machines with Distributed Memory

by
K. Konishi, T. Maruyama, A. Konagaya (NEC),
K.Yoshida & T. Chikayama

October, 1991



ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

# Implementing Streams on Parallel Machines with Distributed Memory

Koichi Konishi Tsutomu Maruyama Akihiko Konagaya
NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki, Kanagawa 216, Japan
{konishi, maruyama, konagaya}@csl.cl.nec.co.jp

Kaoru Yoshida Takashi Chikayama
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, Japan
{yoshida, chikayama}@icot.or.jp

**Abstract**

Stream-based concurrent object-oriented programming languages (SCOOL) to date have been typically implemented in concurrent logic programming languages (CLL). However, CLLs have two drawbacks when used to implement message streams on parallel machines with distributed memory. One is the lack of restriction on the number of readers of a shared variable. The other is a cascaded buffer representation of streams. These require many interprocessor communications, which can be avoided by language systems designed specially for SCOOLs. The authors have been developing such a language system named A'UM-90 for A'UM, a SCOOL with highly abstract stream communication. This paper presents the optimized method used in A'UM-90 to implement streams on distributed memory. A stream is represented by a message queue, which migrates to its reader's processor after the processor becomes known. The improvement from using this method is estimated in terms of the number of required interprocessor communication, and is demonstrated by the result of a preliminary evaluation.

## 1 Introduction

One natural use of concurrent logic programming languages(CLLs) is to implement the Actor or object-oriented programming models. In a CLL, it is easy to specify objects running concurrently, communicating with one another by messages sent in streams[5]. Message streams in CLLs are especially useful, as they provide flexibility and modularity, and facilitates the exploitation of parallelism; they allow dynamic re-configuration of communication channels, while each object knows little about the partners with whom it is communicating.

To support this style of programming, a number of languages have been proposed ([1] [2] [7] [4]). We call these languages stream-based concurrent object-oriented languages(SCOOL).

Most research on SCOOLs to date has been focused on providing excellent expressibility. While SCOOLs have been implemented in CLLs, to our knowledge, no language system dedicated for SCOOLs has been implemented.

A dedicated system for SCOOL can be much more efficient than those implemented in CLLs when the abstraction and other information in programs are fully exploited. The authors have been developing such a dedicated system for a kind of SCOOL, A'UM. The system is named A'UM-90, and is targeted for multiprocessor systems with distributed memory.

In this paper, some drawbacks of CLLs as implementation languages for stream communications are discussed, then it is shown how A'UM's well-regulated abstract streams can be efficiently implemented. A brief description of such an implementation is given, its improvement over a CLL implementation is estimated, and the results of a preliminary evaluation are given.

The next section describes the implementation of objects and stream communication in CLLs. Section 3 introduces SCOOLs as natural descendants of CLLs. Section 4 explains why CLLs are inadequate for implementing streams. Section 5 describes A'UM and A'UM-90 briefly. Section 6 describes the implementation of stream communication in A'UM-90 and its costs. Section 7 shows some results of evaluation. The last section gives conclusion.

# 2 Objects in CLL

Stream-based concurrent object-oriented programming languages have evolved from efforts to embody the Actor or object-oriented programming models in CLLs[5]. This style of programming has the virtues of object-oriented programming such as modularity and natural parallelism in an extended way[3]. For example, an object implemented in a CLL may have multiple input ports, and communication ports can be transferred between processes. Moreover, it can send messages before the destination is determined. In this chapter, an implementation of object-oriented programming in a CLL

```
object([message(Arguments) | In], State) :-
        true |
        method(Arguments, State, NewState),
        object(In, NewState).
```

Figure 1: A clause representing an object

is briefly described.

Many CLLs (FCP, FGHC, Fleng, Oc, Strand, etc.) have been proposed to date. We use FGHC[6] in the following explanation.

Figure 1 shows a typical example of representing an object in FGHC. The behavior of an object is defined by a number of clauses similar to the one above. Given these clauses, a goal named `object` represents the state of an object at a certain moment. The first argument is a shared variable used as a communication port, from which the object receives messages. The second argument is the internal state of the object.

When another goal sharing the variable with the first goal assigns a term `[message(Actuals) | Rest]` to the variable, the above clause can be selected, and `Rest` becomes shared by the two goals. `Actuals` are bound to `Arguments`, and the body of the clause is executed.

A goal named `method` performs most of the actual work, creating new states and assigning it to `NewState`. A new `object` goal is created with `Rest` as the first argument and `NewState` the second. Thus, an object, or a process, is represented by the recurring creation of goals with altered states.

Communication ports are represented by variables shared by two goals. One goal emits a message by assigning a structure containing a message and a new variable. When the other goal receives the message by successfully matching itself with a head of a clause, the new variable becomes shared, to be used as a new port. By repeating this procedure, these goals can communicate as many messages as required, one after another. The connection is closed when a structure containing no variable is assigned. Communication in this style is called *stream communication*.

Basically, stream communication is one-to-one as described above. However, several streams of messages can easily be merged into one by a simple process. A merger should

have several ports representing the input streams to be merged and one more for the output. It receives a message from one of its input ports and forwards it to the output port.

Many types of mergers with varying policies can be devised. A merger of one type might receive from an arbitrary port, utilizing the non-determinism in clause selection of the CLL. A merger of another type might concentrate on one port until the connection through it is closed, then it might move on to another port. We call the former type a *merger*, and the latter an *appender*, because it effectively appends streams one after another.

# 3 SCOOL

Programming objects in a CLL has several obvious drawbacks.

First of all, the implementation of stream communication is explicitly described in the program. Streams are explicitly formed using messages and a variable, and many to one communications are implemented with merger processes. Programmers must make sure that the same conventions are used throughout their programs. Secondly, contentions are apt to happen, due to the lack of restriction on multiple writers to a variable. Lastly, the verbosity, in particular manipulation of internal states, is excessive. It is cumbersome to provide all the details of communication.

Many SCOOLs have been proposed to remove these drawbacks ([1] [2] [7] [4]). These languages have a form for class definition, introduced to make a concise description of object behavior possible. Stream communication is denoted by dedicated expressions, with its implementation removed from programs.

To our knowledge, all SCOOLs have been implemented in CLLs. It is natural and efficient to use CLLs for this purpose, but is problematic with respect to the resulting system's performance. CLL systems can not provide a thoroughly object-oriented view efficiently, such as integers operated on by messages. Another problem is implementing stream communication on a multiprocessor system with distributed memory. We focus on the latter problem, and explain the inadequacies of CLLs in the next section.

# 4 Problems in implementing streams in CLLs

Stream communication, and more generally asynchronous communication, uses message buffers to store pending messages. In distributed memory multiprocessor systems, accessing a message buffer requires inter-processor communications(IPC), unless both the accessing process and the buffer are on the same processor.

While a single IPC suffices to write a message into a buffer on a remote processor, reading a message requires two: a request and a reply. Placing the buffer on the reader's processor, one IPC can be saved for each message communicated through the buffer.

However, it's difficult for CLL systems to place the buffer on the reader's processor. CLL systems use a shared variable as a message buffer, and they can't tell the readers of a variable from the writers. In addition, there may be multiple readers for a variable. In that case, there is a relatively small advantage in saving IPCs for only one reader among many.

Moreover, the number of IPC's required would not be reduced even if the buffer is placed on the reader's processor. In a CLL, streams are represented as a sequence of message buffers, and the writer only knows the last one. When it becomes full, a new buffer is appended to the sequence, and if it is created on the reader's processor, the address must be propagated to the writer. This costs an additional IPC for every message sent.

Since CLL systems may not place shared variables on the reader's processor, implementing these streams in CLLs results in costly remote reads, repeated for every buffer.

The argument so far prompts the development of a dedicated system for SCOOLs. A'UM-90 is such a system for A'UM, a SCOOL that thoroughly integrates streams into its specification. The next section describes A'UM and gives an overview of A'UM-90.

# 5 A'UM and A'UM-90

## 5.1 Behavior of Objects

All A'UM objects run concurrently. They keep internal states called *slots*, and execute methods according to the messages they receive.

The class an object belongs to defines its behavior. A class definition has the following form, which includes the declaration of the class name, the classes it inherits from, slot names (local state) and definitions of its methods.

```
class class_name.
        super_class_decl
        slot_decl
        method_defs
end.
```

An object receives messages from only one stream, called its *interface*. An object is referenced by connecting a stream to its interface. Streams connected to the object later on will be merged into the interface.

A method is defined by the following form.

*selector* -> *actions*.

where *selector* is the method's name, and *actions* specify the operations it performs.

The only operations methods are allowed to perform are connecting a stream to another, creating an object, and sending a message to a stream.

## 5.2 Streams in A'UM

Stream communication in A'UM is highly abstract, providing safe communications and the notion of channels. Directed variables prevent contentions for a stream. The semantics of variables are enhanced so that they denote a set of confluent streams called a channel, a more general concept than a stream.

All variables in A'UM have a stream as their value. The role of streams in A'UM is similar to pointers in Lisp; streams are the sole way of referencing objects.

### 5.2.1 Operations on Streams

A stream is a sequence of messages, directed to a certain receiver. A message sent to a stream is placed at the end of the stream. Sending is expressed simply by juxtaposing a stream and a message, as follows.

*stream message*

Connection of two streams are denoted by the following syntax.

*receiver* = *stream*

This means that all messages sent to *stream* flow into *receiver*.

Closing a stream indicates that no more messages will be sent through it. Closing is always performed automatically, when a stream is discarded.

In addition, messages arriving at an object's interface stream are consumed exclusively by that object. This operation is also performed automatically.

### 5.2.2 Directed Streams

Stream connection is asymmetric; a stream may only be connected to another stream once, but many other streams may be connected to it. In order to assure at compile-time that streams are connected only once, references to a stream are classified into two types, called *directions*. An *inlet* is a reference to a stream from which messages flow; an *outlet* is another kind of reference in which messages are sent[1]. The single connection of a stream is assured by the restrictions requiring that a stream has only one inlet and that the right hand value of a *connect* expression be an inlet.

Inlets and outlets are distinguished syntactically. Variables referencing inlets are denoted with a variable name with ^ prepended to it, e.g. `^X`. Slots holding inlets and outlets are written as slot names preceded by @ and by !, respectively. Expressions have a value whose direction is determined according to their kind. Messages are distinguished by the directions of their arguments as well as their number, and the message's name.

[1] They are named from an object's point of view.

```
class account.
    out balance.
:init ->        0 = !balance.
:deposit(^Amount) ->    !balance + Amount = !balance.
:withdraw(^Amount, ^Ack) ->
        (Amount < !balance) ? (
        :'true ->       !balance - Amount = !balance.
        :'false ->      Ack :overdrawn(!balance).
        ).
:balance(!balance) -> .
end.
```

Figure 2: Bank account

### 5.2.3 Channel Abstraction

Two types of stream confluence, namely mergers and appenders have special support in the language. As mentioned earlier, a merger performs non-deterministic merging, and an appender connects streams one after another in a specified order.

A channel is a tree formed of these confluences of streams. Variables represent a channel of a particular form, consisting of an appender and an arbitrary number of mergers. All outputs of the mergers are connected to inputs of the appender.

For a variable named `Foo`, `^Foo` is an inlet of the root stream of the channel. `Foo$1`, `Foo$2`, `Foo$3`, and so on, are leaf streams. `Foo` is equivalent to `Foo$1`. They are appended into the root in the order of their number. When there are many expressions having the same number, the streams they denote are merged before being appended.

Using channels reduces the description of mergers and appenders in programs, which would be indecipherable otherwise.

## 5.3 An Example Program

Figure 2 is an example A'UM program defining a class for a bank account.

Arguments in a message are connected with values of the expressions in the selector corresponding to the message. For example, `:deposit` receives an outlet and connects `^Amount` to it. `:balance` receives an inlet and connects it to the value of `!balance`.

A binary expression is a macro form. It expands into a *send* expression, which sends to the left hand value a message with two arguments, the right hand value and an inlet of a new stream. The name of the message is determined according to the operator. A macro form evaluates into an outlet of the new stream. Thus, `!balance + Amount` are expanded into `!balance :add(Amount, ^Result)`, with `Result` as its value.

*exp* `?` `( ... )` is an anonymous class definition, which is used to represent a conditional behavior. Either of the methods `:'true` or `:'false` is executed by the instance of the anonymous class, according to the result of `Amount < !balance`.

### 5.4 An outline of A'UM-90

A'UM-90 is an A'UM language system, independent of any CLL. It provides efficient stream communication on a distributed memory multiprocessor system. Moving stream data structures to their reader's processor saves many IPCs, which are otherwise required in stream communication.

A'UM-90 manages coarse-grained processes. Specifically, a process executes an instance of a user-defined class.

An A'UM-90 system consists of a compiler and an emulator. The compiler generates code for an abstract-machine designed for the system, and the emulator executes the code.

Two different types of platform have been used. One is a Sequent Symmetry with 16 processors, and the other is a number of Sun Sparc Stations communicating by Ethernet. Although a Symmetry has shared memory, we used it as a distributed memory machine. We used a small part of the memory to implement message communication, and divided the rest among processors.

## 6 Implementation of Streams in A'UM-90

The implementation described here fully utilizes information on stream abstraction and message flow direction available in A'UM programs. Although the delivery of the first message is somewhat delayed, the number of IPCs required is significantly reduced, when many messages are sent through a long cascade of streams. Moreover,

the delay is eliminated in many cases by various subtle optimization methods.

## 6.1 Streams

A stream is represented by a structure consisting of a message queue, a pointer to its receiver, and a reference count. The reference count is necessary for detecting closed streams and for implementing the appenders correctly. The structure is named *M node*, where M stands for merging. A merger is simply represented as an M node having more than one pointer referring to it. An appender is represented by a structure consisting of an M node and a pointer to the following stream. The structure is named *A node*.

With these structures, implementing operations on streams within a processor is straightforward. Sending a message is simply queuing it. Connecting a stream to a receiver is making the pointer in the stream point to the receiver and increasing the reference count of the receiver. When a stream is closed, its reference count is decreased. Receiving a message is just dequeuing it.

## 6.2 Location of Streams

As argued in a previous section, a stream should be placed on its receiver's processor in order to decrease the number of IPCs. However, when a stream is created, its receiver is still unknown. So we place it on the processor local to its creater at its creation, and let it migrate later to the receiver's processor.

Since it is always an object that ultimately receives messages sent to a stream, the stream migrates to the object's processor. When the stream is directly connected to the object, it migrates immediately. If it is connected to an intermediate stream, it waits until the intermediate stream migrates.

Suppose that an address of a stream in a processor is announced to an object in another processor and that the stream has not yet migrated. If the object sends messages to the stream, two series of IPCs occur, one for sending them to the stream, and another for the migration process of the stream. We eliminate the former series by putting the messages into a new stream created on the same processor as the sending object and connecting the new stream to the original.

With the strategy described so far, and assuming that objects don't migrate, all messages, except those used for implementing the strategy, are transferred between

processors at most once. In the next section, a more detailed description of the stream migration is given.

## 6.3 Migration Procedure

In the following description, all streams are supposed to reside in different processors until they move. Operations within a processor are trivial, and are assumed to cost much less than ones involving IPCs. It is also supposed that streams are connected in a processor other than that of the receiving object. Otherwise, the migration procedure is so simple to become identical with an ordinary sending without migration.

1. A stream is placed on the same processor as its creator object.

2. When the stream is connected, a control message named *where* is sent to the specified receiver. The control message has a pointer to the stream and a tag showing the type of the stream, i.e., either an M node or an A node.

3. The *where* causes the following actions according to the type of the receiver:

   **a stream before its migration** handles the control message as if it is an ordinary message. That is, it is put into the receiver's queue. It will be transferred again when the receiver eventually migrates, and will be forwarded to another receiver, which should cause the following case.

   **an object or a stream after its migration** creates a new node of the type indicated by the tag in the control message, and reports the address of the new node by a control message named *here* to the stream waiting for the reply. When the type of the immigrant and the receiver is the same, the receiver creates no new node, and reports its own address.

4. When the stream receives the *here*, it migrates to the specified new residence, in one of the following manners according to its type:

   **M node** It sends all messages in its queue to the new residence. If it hasn't been closed yet, it leaves in the former residence a pointer forwarding to the new location. The original residence will be reclaimed when it is closed.

**A node** In addition to the procedure for the M node, the stream to be appended to the migrating one is connected to the same receiver at the moment when this A node is closed. That is, a new *where* with a pointer to the stream is sent to the receiver.

## 6.4 Migration Cost

Each stream creates a *where*. It is transferred between processors twice, once when the stream is connected, and once when its receiver migrates. The second transfer doesn't happen if the receiver is an already moved stream or an object. Suppose a channel connected to an object consists of $n$ streams, and of which $n_d$ are connected directly to the object, then the number of IPCs for *where* is $n + (n - n_d)$.

A *here* is created in correspondence with a *where*, and is transferred between processors once. For all *here*'s , $n$ IPC's occur.

Migration brings about no transfer of control messages, so the number of IPCs required for migration is:

$$n + (n - n_d) + n = 3n - n_d$$

Closing a stream requires another kind of control message. We call it *close*. Each stream sends its reader one *close* when closed. This adds up to $n$ *close*'s requiring $n$ IPC's.

Ordinary messages are transferred between processors always once. If there are $m$ ordinary messages to be sent, then, in total,

$$(3n - n_d) + m + n$$

transfers between processors occur.

How many IPCs occur for stream communication if streams don't move? Neither of *where* and *here* are created. A *close* is still created for a stream. The number of times ordinary messages and *close*'s are transferred depends on the structure of the channel.

A channel is a tree having streams as its nodes. Suppose the i-th node receives $m_i$ messages, and its depth is $d_i$, where a depth of a node is number of streams in the

path from the leaf to the root. For example, the depth of a leaf directly connected to an object is 2. Then messages sent to the i-th leaf is transferred $d_i - 1$ times, and the total number of transfers will be:

$$\sum_{i=1}^{n}(d_i - 1)(m_i + 1)$$

The condition when it requires less IPCs to implement stream communication with migrating streams than without them is:

$$\sum_{i=1}^{n}(d_i - 1)(m_i + 1) > (3n - n_d) + m + n$$

This can be rewritten as:

$$\sum_{i=1}^{n}(d_i - 2)(m_i + 1) > 3n - n_d$$

Since $d_i$ can not be smaller than 2, $d_i - 2$ never becomes negative. The next term $m_i + 1$ is the number of messages sent from a node, including a *close*. The last term $3n - n_d$ is the number of control messages used to move all streams.

The above condition says that if the channel has some intermediate nodes between the root and leaves, and more than a certain number of messages are sent through them, then stream migration is beneficial. Conversely, if all streams in a channel are directly connected to an object, or too few messages are sent, streams should not be moved. The next section discusses some optimization based on detecting those cases.

## 6.5 Further Optimization

The left-hand side of the above condition becomes zero when all streams are directly connected to an object. When connecting a stream, it is detected at run-time that the receiver is an object; pointers are tagged to indicate the type of the pointed structure. By not moving those streams, the right-hand side is also decreased to zero when the left-hand becomes zero.

When less than two messages are sent through a stream, the stream does not migrate, i.e. it does not send out a *where*. More detailed analysis shows that *two* is the least number to make stream migration beneficial. The current emulator in A'UM-90 determines at run-time whether a stream receives less than two messages. Finding

| *create* | *here* | ordinary | *close* |
|---|---|---|---|
| 303 | 303 | 47572 | 303 |

Table 1: numbers of messages sent in *PRIMES*

all such streams is difficult for a compiler because it requires global analysis. Local analysis within a method, however, may be able to catch most of them. For example, for a stream created and closed within a method, the number of messages sent to it is easy to count.

In addition, various minor optimization methods are applied to reduce the delay of the first message's delivery. For example, the first message is sent with a *where*, packed together in one IPC, if it is available when the *where* is sent out. When a *where* is received by a stream that only bridges two other streams, receiving no ordinary messages, it immediately forwards this *where* instead of sending out a new one. Such a stream can be distinguished by checking its reference count when it receives a *where*.

# 7 Evaluation

Table 1 shows the numbers of various messages sent in a execution of a program generating prime numbers under 2000 by Eratosthenes's sieving(*PRIMES*). 303 objects are created, and the same number of streams are created and are directly connected to these objects respectively. No *where* is sent, since every connection in this program takes place in its receiver's processor.

If this program is executed on the system implemented in a CLL, there would occur almost twice as many IPC's as the above, i.e. more than forty thousand extra IPC's.

The number of IPC's used in a CLL implementation can be decreased by sending many messages at a time using a large buffer. Of course, the size of the buffer must be adjusted appropriately, as too large a buffer degrades efficiency of memory usage. Such adjustments are practical only after you acquire some knowledge about actual number of messages sent in a program by executing it. Our implementation does not need such an elaboration.

Figure 3 is a graph presenting speedups in accordance with numbers of processors

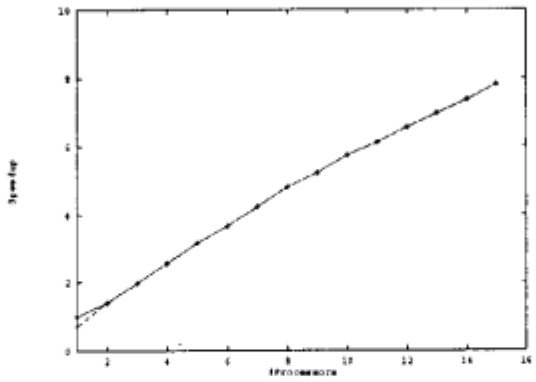

Figure 3: speedups of *PRIMES*

utilized. The result was measured on a Symmetry. A speedup of eight times is attained using fifteen processors.

The speedups presented in the graph are relative to the performance of a system made specifically for sequent execution, which is free from any parallel processing overhead. The current emulator employs a simple-minded load distribution method of always placing a newly created object on a different processor than that of its parent. Considering these conditions, this result can be considered fairly good.

# 8 Conclusion

Streams in CLLs are difficult to implement efficiently for two reasons:

1. Message buffers are not always placed on their readers' processor, because an arbitrary number of readers are allowed for a buffer. Therefore, interprocessor reading from the buffer takes place with two IPCs, instead of one required for writing into it.

2. A stream is represented by cascaded message buffers, which CLLs don't treat as a single body. Consequently, even if these buffers are placed on their reader's processor, their address has to be repeatedly sent to their writer.

This is not the case for A'UM. A'UM has abstract stream communication, whose implementation is left as the language systems' responsibility. In addition, every stream is restricted to have only one reader. So streams in A'UM can be more efficiently implemented than ones in CLLs. Moving a stream to its reader's processor saves about half of the IPC's required in CLLs, and, in spite of the migration, the first message through the stream is delivered without delay for most of the cases.

While the optimization method given in this paper tries to reduce the number of IPC's for a given distribution of objects, it is also important to find the best distribution of objects. Of course, those methods have to balance the amount of IPC's and the parallelism exploitation.

## Acknowledgments

We thank Shinji Yanagida and Toshio Tange of NEC Scientific Information System Development for developing the A'UM-90 abstract-machine emulator.

## References

[1] K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki, K. Ueda, *Mandala: A Logic Based Knowledge Programming System*, Proc. FGCS'84, November 1984.

[2] K. Kahn, E. D. Tribble, M. S. Miller, D. G. Bobrow, *Objects in Concurrent Logic Programming Languages*, Proc. OOPSLA'86, September, 1986.

[3] K. Kahn, *Objects - a fresh look*, Proc. Third European Conf. on Object-Oriented Programming, Cambridge University Press, July 1989.

[4] V. A. Saraswat, K. Kahn, J. Levy, *Janus: A step towards distributed constraint programming*, North American Logic Programming Conference, October 1990.

[5] E. Shapiro, A. Takeuchi, *Object-oriented Programming in Concurrent Prolog*, New Generation Computing, 1, 1983.

[6] K. Ueda, *Guarded Horn Clauses*, Technical Report TR-103, ICOT, June 1985.

[7] K. Yoshida, T. Chikayama, *A'UM: A Stream-Based Object-Oriented Language*, Proc. FGCS'88, November 1988.