ICOT Technical Report: TR-0778

TR-0778

An Efficient Message Transfer Mechanism Bypassing Transit Processors

by

H. Nakashima (Mitsubishi) & Y. Inanura

June, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome

(03)3456-3191 ~ 5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

An Efficient Message Transfer Mechanism Bypassing Transit Processors

Hiroshi Nakashima
(Mitsubishi Electric Corp.)

Yū Inamura
(ICOT Research Center)

Abstract

This paper describes an efficient mechanism of inter-processor message transfer on loosely-coupled/message-base parallel processing systems. This mechanism eliminates *transit* processors, which merely relay messages transferred between other processors, using *one-way* communication with three additional physical messages.

本稿は，疎結合型のマルチプロセッサにおいて，他のプロセッサ間のメッセージ通信を単に中継するだけのプロセッサをバイパスする方式について述べたものである。本方式では三種類の付加的なメッセージを用いて，中継プロセッサのバイパスが可能な一方向通信を実現している。

1 Introduction

For large scale parallel processing, it is desirable that programming languages have capability to represent parallelism in problems naturally. In various programming languages proposed for parallel processing, parallel logic programming languages, such as GHC [Ueda 85], and parallel object-oriented programming languages, such as ABCL [Yonezawa 86], will be hopeful candidates because concurrent processes communicating messages each other are easily and naturally described in them. It is also natural to map these processes onto loosely-coupled/ message-base parallel processing systems [Nakajima 89, Takada 89].

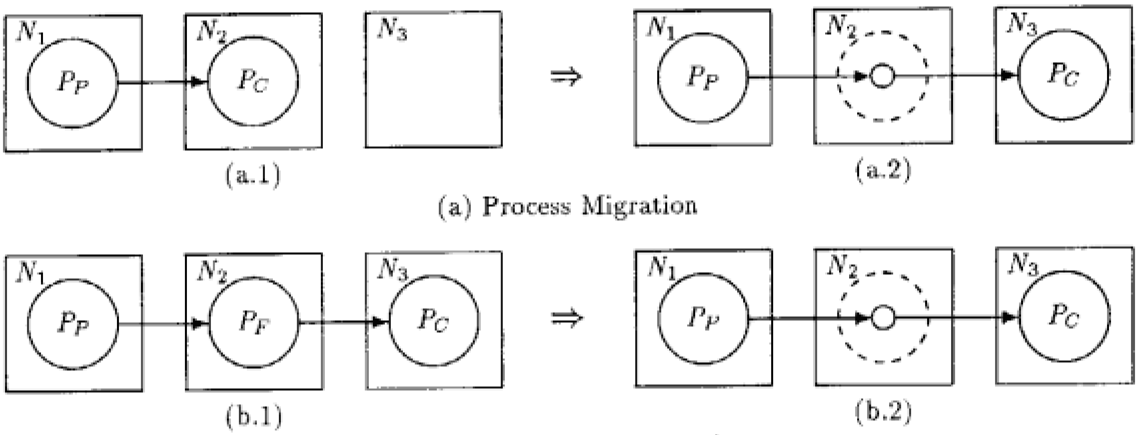
From the viewpoint of efficiency, however, the implementation of those languages on such systems is not so easy, because the cost of communication between processor nodes is often much higher than that of computation. Therefore, the number of *physical* inter-processor messages for a *logical* inter-process message should be minimized for efficient implementation.

In order to minimize the number of messages, *transit* processors, which merely relay messages transferred between other processors, have to be bypassed. For example, when a process P migrates from a processor node N_1 to N_2, it is expected that messages directed to P will not pass through N_1 but reach N_2 directly. This bypassing is easily implemented using *two-way* communication in which one logical message transmission takes two physical messages, forward and backward. In *one-way* communication systems which is more efficient than two-way systems, however, bypassing seems difficult because the sender is usually ignorant of receiver's activity.

In this paper, we propose an inter-processor one-way message passing mechanism which is capable of bypassing transit processors with a backward message telling the sender that the receiver migrated. This mechanism also has capability to preserve the order of messages using a forward and a backward message, assuming FIFO property of physical network.

This paper is organized as follows: Section 2 shows how a processor is made transit, and how conventional schemes deal with transit processors; section 3 presents the basic scheme and mechanism to bypass transit processors; section 4 discusses implementation details comparing related works with ours; and section 5 gives the conclusion.

中継プロセッサをバイパスするメッセージ通信方式
中島 浩（三菱電機，hiroshi@isl.melco.co.jp），稲村 雄（ICOT）



(a) Process Migration

(b) Short Circuit Termination

Figure 1: Transit Processors

2 Problems and Conventional Schemes

2.1 Transit Processors

Figure 1 shows typical situations in which a processor is made transit. That is;

(a) The process P_C on the processor node N_2 is the consumer of the producer P_P on N_1 (a.1). If P_C migrates from N_2 to N_3, N_2 becomes a transit processor (a.2). Similarly, if P_P migrates from N_1 to other processor, say N_0, N_1 becomes transit.

(b) The process P_F on the processor node N_2 is the filter between P_P on N_1 and P_C on N_3 (b.1). If P_F terminates connecting its input and output, N_2 becomes a transit processor (b.2).

Both situations, process migration and short circuit termination, will often occur in process-oriented or object-oriented parallel programming. Thus, it is greatly expected to bypass the transit processor N_2 and transmit messages from N_1 to N_3 directly.

2.2 Two-way Communication

In parallel logic programming languages, such as GHC or its modified version, KL1 [Chikayama 88], unifications of list cells are usually used for inter-process communication. If the producer P_P and consumer P_C are allocated on different processor nodes, N_1 and N_2, the unification for a list cell might require two physical messages, one of which is *forward* (producer to consumer) and the other is *backward*.

For example, [Ichiyoshi 87] describes the following mechanism.

(a) P_P and P_C share an uninstantiated logical variable S_1 on N_1, and P_C has an external reference R_1 pointing S_1. P_C sends a backward message *%read*(S_1, R_1), which demands that N_1 send the value of S_1 to R_1 if S_1 is instantiated.

(b) P_P unifies a list cell whose car is a message M and cdr is a new variable S_2. This makes N_1 send a forward message *%answer*($R_1, [M|S_2]$) which carries the answer for *%read*.

This *two-way* communication mechanism can easily deal with the problem of transit processors, because P_C always informs P_P of its location. This advantage, however, is not admired, because it is gained by much overhead, two physical messages for a logical message.

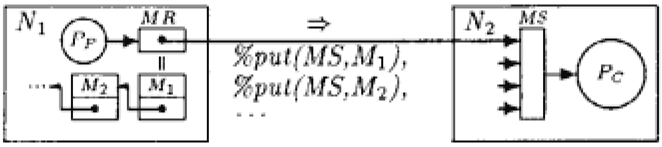


Figure 2: One-way Communication in KL1

2.3 One-way Communication

KL1 has an efficient mechanism, *built-in merger*, for inter-process communication [Inamura 89]. This mechanism provides constant time, non-deterministic n-ary merge operation of *streams* represented as lists. It might also enable processors to make *one-way* communication through merged streams, because its implementation lets producers know that their outputs are connected to a special structure for merge.

Figure 2 shows a possible configuration of the one-way communication using the merger. The input of the consumer process P_C on N_2 is connected to the structure representing merger MS. One of the input streams of the merger is directed by a special external reference MR on N_1. When the producer process P_P unifies MR with list cells, one-way forward messages *%put(MS,M_i)* will be transferred to MS.

This configuration is very similar to that of inter-processor communication between concurrent objects in parallel object-oriented languages, such as ABCL [Yonezawa 86]. For example, MS and MR are corresponding to internal and external object descriptors described in [Takada 89].

Note that this one-way communication mechanism stands on *FIFO assumption* that the physical communication line between N_1 and N_2 preserves message order. Also note that bypassing transit processors is difficult in this mechanism, because a consumer might not have any information about its producers.

3 Bypassing Transit Processors

3.1 Basic Scheme

Figure 3 shows the basic scheme of the proposed mechanism. In Figure 3(a), P_P and P_C are producer and consumer processes allocated on processor nodes N_1 and N_2 respectively. R_1 and R_2 are external and internal *process pointer* for P_C, both of which might have capability to merge message streams (not shown in the figure). These pointers also have queues to keep postponed messages. In this state, communication between P_P and P_C is performed in *one-way* manner on the FIFO assumption.

If the process P_C migrates to another processor node N_3, R_2 changes its state to external, and directs a new internal process pointer R_3 on N_3, as shown in Figure 3(b). That is, N_2 becomes a transit processor. This state, however, is temporary and should be changed to the final state shown in Figure 3(c) in which the pointer R_1 directly points R_3. Note that the temporary state (b) will also appear when the process P_P migrates rather than P_C. In both cases, the state transition from (b) to (c) is performed in the same manner.

For the transformation from (b) to (c), the following physical inter-processor messages are exchanged.

%put(r,m,s) .... Send a logical message m from an external process pointer s to a process pointer r. If r is an internal process pointer, s is ignored.

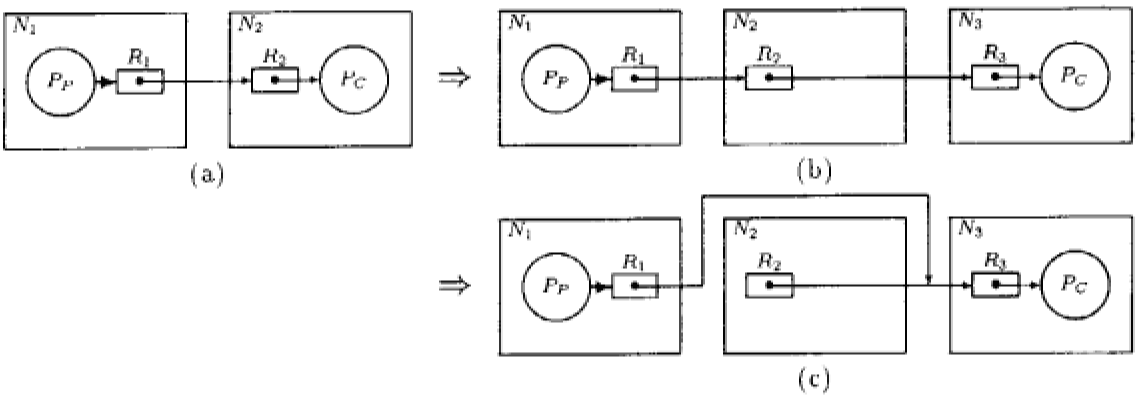


Figure 3: Bypassing Transit Processors

%moved(s) .... Tell s, the sender of *%put*, that the *real* receiver moved somewhere. This message is transferred as the reply of *%put* if the receiver of *%put* is an external process pointer.

%confirm(r,s) .... Request the confirmation that all *%put* messages from s preceding *%confirm* are received by the real receiver. This message is sent by s and relayed by r to the real receiver.

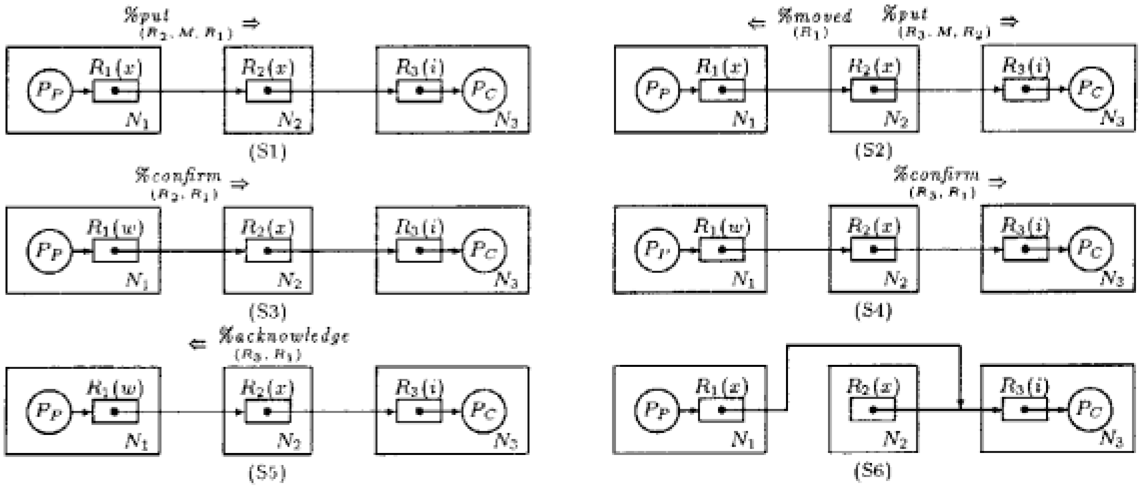
%acknowledge(s,r') .. Reply to s that *%confirm* is received by the real receiver r'.

3.2 Mechanism

The scenario from Figure 3(b) to (c) is as follows (Figure 4).

(S1) A message *%put*(R_2, M, R_1) is sent from R_1 on N_1 to R_2 on N_2.

(S2) As the reply of *%put*, R_2 send *%moved*(R_1) to R_1, because R_2 is an external process pointer. R_2 also relays *%put* to its destination R_3.



i: internal, x: external, w: external-waiting

Figure 4: Scenario of Bypassing

Table 1: Actions of Process Pointers

message	state		
	internal	external	external-waiting
%put(x,m,y)	**eat**	**reply**(*%moved(y)*) **relay**(*%put(d,m,x)*)	**keep**
%moved(x)	–	**reply**(*%confirm(d,x)*) **transform**(*waiting*)	**ignore**
%confirm(x,y)	**reply**(*%acknowledge(y,x)*)	**relay**(*%confirm(d,y)*)	**keep**
%acknowledge(x,y)	—	—	**transform**(*normal*)
internal	**eat**	**relay**(*%put(d,m,x)*)	**keep**

(S3) R_1 receives *%moved*, it changes its state to *waiting* and sends *%confirm*(R_2, R_1) to R_2. In *waiting* state, R_1 postpones sending all messages from P_P (and other processes) until it receives *%acknowledge*. The postponed messages are kept in the queue of R_1.

(S4) R_2 receives *%confirm* and simply relays it to the destination of R_2, because R_2 is an external process pointer. That is, *%confirm*(R_3, R_1) is sent to R_3.

(S5) R_3 receives *%confirm* and sends *%acknowledge*(R_1, R_3) to R_1 as the reply, because R_3 is an internal process pointer. The FIFO property of the communication line between N_1 and N_2 and that between N_2 and N_3 promises that all messages from R_1 to R_3 through R_2 preceding *%confirm* are received by R_3 in the same order in which they are sent.

(S6) R_1 receives *%acknowledge* and sends all messages in its queue to R_3. It also changes its state to *normal* (non-*waiting*) and directly points R_3 which is specified in *%acknowledge*.

The actions described above are summarized in Table 1. The columns of the table are corresponding to the states of process pointer x, internal, external and external-waiting. Each row shows the action taken when the process pointer receives a physical message from y, or it is requested to send a logical message by a process allocated on the same node (*internal*). The following are the explanations of the actions.

eat .... Pass the logical message to the consumer process, or keep it in the queue if there are postponed messages.

relay(*message*) .... Relay the received *message* to the destination, d, pointed by the process pointer.

keep .... Keep the received message in the queue.

reply(*message*) .... Send *message* as the reply of the received message.

transform(*state*) .. Changes the state of the process pointer to *state*. The state transition to *normal* causes sweeping out the contents of the queue.

ignore .... Ignore the received message.

4 Implementation Issues

4.1 Migration of Producer

In [Kukula 88], Kukula proposed a mechanism to bypass transit processors for an object-oriented system OX. Kukula's mechanism handles migration of producers and consumers separately. When a producer migrates, it reports its departure to consumer's process pointer.

Then it moves to another processor node and informs the consumer of its arrival*. In order to keep message order, the consumer must make a sidetrack queue for messages from new location received before it catches the departure message from old location. Note that multiple sidetrack queues are necessary in case that the producer rapidly repeats migration.

In contrast with OX, our mechanism easily handles producer's migration in the way exactly same as consumer's migration. When a producer migrates, it makes a external process pointer R_n on the new processor node to direct process pointer R_o on the old node. If R_o is internal, the producer can start message transmission immediately. Otherwise, the situation is same as that shown in Figure 3(b), and successive transmission of *%put* will trigger the procedure described in 3.2.

In latter case, of course, exchanging *%put* and *%moved* in step (S1) and (S2) can be omitted. That is, the procedure can be started from step (S3) by setting the state of R_n to *waiting* and sending *%confirm* immediately. However, this optimization is optional and requires that the producer remembers the state of R_o. There is the other alternative in which the procedure starts from step (S3) regardless the state of R_o. This method brings unnecessary message exchange in case that R_o is internal, but will be appropriate for *backward pointer* method as discussed later.

4.2 Migration of Consumer

When a consumer migrates, the internal process pointer R_o for the consumer should change its state to external. It is also necessary to keep messages for the consumer in the queue of R_o, until the consumer arrives new processor node. This queueing is easily performed by setting the state of R_o to external-waiting. When the consumer arrives new processor node, it makes a internal process pointer R_n, and send *%acknowledge* to R_o. This message will sweep out the contents of the queue, and change the state of R_o to external. This mechanism is much simpler than that for OX which needs an additional message and state.

It is easy to change the state of R_o, if the consumer process has a reference to R_o. For example, Takada proposed a distributed implementation of ABCL in which an object has *self* pointer to its descriptor [Takada 89]. On the other hand, a KL1 process consuming a merged stream only has the reference to the queue (list) top [Inamura 89]. The merger structure (or process pointer), has the reference to the queue tail which is an uninstantiated variable. In this configuration, the consumer cannot report its migration to the merger structure. Thus the consumer silently migrates with the reference to the queue top, and it will fetch elements in the queue in two-way manner described in 2.2.

This problem can be solved if queue elements are represented as a special data type, say *stream*, other than but unifiable with list. The message *%read* to *stream* data causes a special reply, *%merged*, which orders the consumer to make a merger structure (internal process pointer) on its processor node. Then the consumer will send *%acknowledge* to the *stream* data to sweep out the queue and change the state of the merger structure. In case of queue empty, *%read* is *hooked* to an uninstantiated variable. After that, when a producer puts an element to the queue, *%merged* is sent to the consumer by the unification of the element with the variable.

4.3 Backward Pointer

The message *%put* has a backward pointer to its sender, s. Since s is usually ignored by the receiver, it seems a good idea to remove s from the message and attach it to receiver process

*The arrival message contains new location of the external pointer associated to the producer, as discussed later.

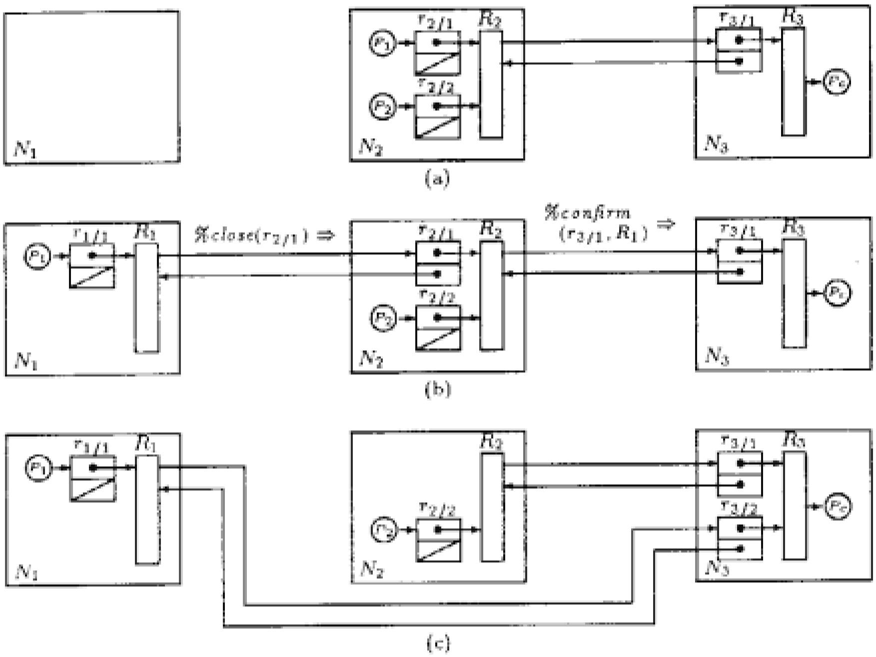


Figure 5: Backward Pointer

pointer, as Kukula does in OX implementation. However, this optimization makes it greatly difficult to merge multiple streams at a process pointer.

In contrast with OX, our mechanism easily handles stream merge because a consumer can be ignorant of locations and population of producers, owing to the backward pointer in *%put*. Thus, a producer can duplicate its message stream arbitrarily, and distribute them to other processes which may be on other processor nodes.

On the other hand, if *duplication* and *addition* of streams are distinguished, the implementation of backward pointer method becomes fairly easy. In KL1, for example, the *duplication* of a stream makes an erroneous multiple writer stream, and putting an element to the stream usually causes unification failure. For *addition*, a producer unifies a vector, whose elements are streams to be merged, instead of usual cons cell. This operation will let consumer's process pointer know where producers are.

Figure 5 shows an example configuration with backward pointers. Additional indirection cells $r_{i/j}$ are *receiver* cells each of which has the pointer to process pointer R_i to forward messages. A receiver cells also has a backward pointer to the external process pointer, if it referred from other processor node. When the producer P_1 migrates, it sends *%confirm*($r_{2/1}$, R_1) to the receiver regardless the state of the process pointer as described in 4.1 (Figure 5(b)). When the message reaches the receiver connected to an internal process pointer, a new receiver cell $r_{3/2}$ is created with the backward pointer to R_1 (Figure 5(c)).

Garbage collection for receiver cells and process pointers is also possible. A producer may *close* the stream to its consumer, and send a message *%close(r)* if the receiver r is external. Closing a stream will reclaim the receiver cell. If the receiver is the last one, the process pointer will be reclaimed too, closing its output stream. The message *%close* is also sent directly following *%confirm* from the external process pointer which like to confirm (Figure 5(b,c)).

5 Conclusion

An efficient message transfer mechanism which is capable to bypass transit processors has been presented. This mechanism requires only three additional physical messages to preserve the order of logical messages transmitted through non-bypassed and bypassed routes. The action of the receiver of these messages is simple and well defined.

When a producer and/or its consumer migrates, messages for bypassing are also used to establish the connection between them. This makes the implementation of our mechanism much easier than previous works. Language specific implementation details, such as the detection of consumer's migration and the management of merged streams, are also discussed.

We are now precisely designing the implementation of the proposed mechanism for KL1. As for other languages, such as object-oriented languages, we have started basic studies about efficient inter-processor communication scheme including our mechanism. These work will greatly contribute to research activities on parallel processing, especially for parallel programming language design and dynamic load balancing.

Acknowledgments

We would like to thank Takashi Chikayama, Kazuo Taki, Katsuto Nakajima, and all those who contributed to the distributed implementation of KL1.

References

[Chikayama 88] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 230–251, 1988.

[Ichiyoshi 87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proc. 4th Intl. Conf. on Logic Programming*, pp. 257–275, 1987.

[Inamura 89] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Technique Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. North American Conf. on Logic Programming 1989*, pp. 907–921, 1989.

[Kukula 88] J. H. Kukula. Object Relocation in OX. In *Proc. 1988 Intl. Conf. on Computer Design*, pp. 8–10, Sept. 1988.

[Nakajima 89] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. 6th Intl. Conf. and Symp. on Logic Programming*, 1989.

[Takada 89] T. Takada and A. Yonezawa. The Implementation of an Object-Oriented Concurrent Programming Language on Distributed Environments. *Computer Software*, Vol. 6, No. 1, pp. 17–29, Jan. 1989. (in Japanese).

[Ueda 85] K. Ueda. Guarded Horn Clauses. Technical Report 103, ICOT, 1985. (Also in *Concurrent Prolog: Collected Papers*, The MIT Press, 1987).

[Yonezawa 86] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *Proc. Object-Oriented Programming Systems, Languages and Applications*, pp. 258–268. ACM, Sept. 1986.