March 1986

Report No. STAN-CS-86-1126
Also numbered KSL-86-14

CAREL: A Visible Distributed Lisp

by

Byron Davies

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL 86-14

March, 1986

CAREL: A Visible Distributed Lisp

by

Byron Davies

KNOWLEDGE SYSTEMS LABORATORY
Departments of Medical and Computer Science
Stanford University
Stanford, California 94305

KSL 86-14

CAREL: A Visible Distributed Lisp

Byron Davies

Knowledge Systems Laboratory
and Center for Integrated Systems
Stanford University
Palo Alto, California

and

Corporate Computer Science Center
Texas Instruments
Dallas, Texas

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing, under contract number W-266875.

Table of Contents

Abstract

CAREL is a Lisp implementation designed to be a high-level interactive systems programming language for a distributed-memory multiprocessor. CAREL insulates the user from the machine language of the multiprocessor architecture, but still makes it possible for the user to specify explicitly the assignment of tasks to processors in the multiprocessor network. CAREL has been implemented to run on a TI Explorer Lisp machine using Stanford's CARE multiprocessor simulator [Delagi 86].

CAREL is more than a language: real-time graphical displays provided by the CARE simulator make CAREL a novel graphical programming environment for distributed computing. CAR-EL enables the user to create programs interactively and then watch them run on a network of simulated processors. As a CAREL program executes, the CARE simulator graphically displays the activity of the processors and the transmission of data through the network. Using this capability, CAREL has demonstrated its utility as an educational tool for multiprocessor computing.

1. Context

CAREL was developed within the Advanced Architectures Project of the Stanford Knowledge Systems Laboratory. The goal of the Advanced Architectures Project is to make knowledge-based programs run much faster on multiple processors than on one processor. Knowledge-based programs place different demands on a computing system than do programs for numerical computation. Indeed, multiprocessor implementations of expert systems will undoubtedly require specialized software and hardware architectures for efficient execution. The Advanced Architectures Project is performing experiments to understand the potential concurrency in signal understanding systems, and is developing specialized architectures to exploit this concurrency.

. The project is organized according to a number of abstraction layers, as shown in Figure 1-1. Much of the work of the project consists of designing and implementing languages to span the semantic gap between the applications layer and the hardware architecture.

The design and implementation of CAREL depends mainly on the hardware architecture level. The other levels will be ignored in this summary, but are described briefly in the full paper. At the hardware level. the project is concentrating 011 a class of multiprocessor archi tecrures. The class is roughly defined as MIMD, large grain, locally-connected, distributed

Layer	Research Question
Applications	Where is the potential concurrency in signal understanding tasks?
Problem-solving frameworks	How do we maximize useful concurrency and minimize serialization in problem-solving architectures7
Knowledge-representation and inference	How do we develop knowledge representations to maximize parallelism In inference and search?
Systems programming language	How can a general-purpose symbolic programming language support concurrency and help map multi-task programs onto a distributed-memory multiprocessor7
Hardware architecture	What multiprocessor architecture best supports the concurrency in signal understanding tasks?

Figure 1-I: Multiple layers in implementing signal understanding expert systems on multiprocessor hardware

memory multiprocessors communicating via buffered messages. This class was chosen to match the needs of large-scale parallel symbolic computing with the constraints imposed by the desire for VLSI implementation and replication. Like the FAIM-1 project [Davis and Robison 85], we consider each processing node to have significant processing and communication capability as well as a reasonable amount of memory - about as much as can be included on a single integrated circuit (currently a fraction of a megabit, but several megabits within a few years). Each processor can support many processes. As the project progresses, the detailed design of the hardware architecture will be modified to support the needs of the application as both application and architecture are better understood.

The hardware architecture level is implemented as a Simulation running on a (uniprocessor) Lisp machine. The simulator, called CARE for "Concurrent ARray Emulator" (sic), carries out the operation of the architecture at a level sufficiently detailed to capture both instruction run times and communication overhead and latency. The CARE simulator has a programmable instrumentation facility which permits the user to attach "probes" to any object or collection of objects in the simulation, and to display the data and historical summaries on "instruments" on the Lisp machine screen. Indeed, the display of the processor grid itself is one such instrument.

2. Introduction

The CAREL (for CARE Lisp) language is a distributed-memory variant of QLAMBDA [Gabriel and McCarthy 84] and an extension of a Scheme subset [Abelson and Sussman 85]. CAREL supports futures (like Multilisp [Halstead 84]), truly parallel LET binding (like QLAMBDA), programmer or automatic specification of locality of computations (like Par-Alfl [Hudak and Smith 86] or Concurrent Prolog [Shapiro 84], and both static assignment of process to processor and dynamic spread of recursive computations through the network via remote function call. Despite the length of this list of capabilities, CAREL is perhaps best described as a high-level systems programming language for distributed-memory multiprocessor computing.

The CAREL environment provides both accessibility and visibility. CAREL is accessible because, being a Lisp, it is an interactive and interpreted language. The user may type in expressions directly and have them evaluated immediately, or load CAREL programs from files. If the multiprocessing features are ignored, using CAREL is just using Scheme. The multiprocessing extensions in CAREL are derived from those of QLAMBDA. For example, PARALLEL-LET is a simple extension of LET which computes the values for the LET-bindings concurrently, at locations specified by the programmer or determined automatically.

CAREL gains its visibility through the CARE simulator: CAREL programmers can watch their programs execute on a graphic display of the multiprocessor architecture. Figure 5-l shows CARE and CAREL with a typical six-by-six grid of processors. A second window on the Lisp machine screen is used as the CAREL listener, where programs are entered. As a CAREL program runs, the simulator illuminates each active processor and each active communication link. The user may quickly gain an understanding of the processor usage and information flow in distributed CAR EL programs. CARE instruments may also be used to gather instantaneous and historical data about the exection of CAREL programs.

The rest of the paper is divided into a discussion of the philosophy of CAREL, a description of the language CAREL. and some illustrated examples of CAREL in action on the CARE simulator.

3. Philosophy and Design

The CAREL language was developed with a number of assumptions in mind. The following assumptions are stated very briefly for this summary but appear in expanded form in the full paper:

1. CAREL (like Multilisp) was designed to augment a serial Lisp with "discretionary" concurrency: the programmer, rather than the compiler or the run-time support system, decides what parts of a program will be concurrent. CAREL provides parallelism through both lexical elaboration and explicit processes [Filman and Friedman 84].

2. Similarly, CAREL was designed to provide discretionary locality: the programmer also decides ***where*** concurrent routines will be run. A variety of abstract mechanisms are provided to express locality in terms of direction or distance or both.

3. CAREL generally implements ***eager*** evaluation: when a task is created, it is immediately started running, even if the result is not needed immediately. When the result is needed by a strict operator, the currently running task blocks until the result is available.

4. CAREL is designed to automatically manage the transfer of data, including structures, between processors. CAREL supports general methods to copy lists and structures from one processor to another, and specialized methods to copy programs and environments.

5. CAREL is designed to maintain "architectural fidelity": all communication of both data and executable code is explicitly handled by the simulator so that all costs of communication may be accounted for.

6. CAREL provides certain specialized "soft architectures", such as pipelines, overlaved on the processor network.

7. Through CARE, CAREL graphically displays the runtime behavior of executing programs.

8. Finally, and unfortunately, CAREL ignores resource- management, including the problem of garbage collecting data and processes on multiple processors. Resource management is a very important problem, but CAREL doesn't yet have a solution for it. CAREL currently depends on the memory management of the Lisp machine on which it

4. The Language

This section presents a language description of CAREL and examples - with graphics - of its use. The functions and special forms of CAREL were selected roughly as the union of the capabilities of QLAMBDA (as extended for distributed memory) and Par-Alfl. There has been no attempt as yet to create a minimal but complete subset of CAREL.

On top of Scheme subset, CAREL supports the following functions and special forms:

PARALLEL-LET: a special form for parallel evaluation of LET binding. Optionally, the programmer may specify the locations at which the values for binding are to be eval ua ted.

PARALLEL-LAMBDA: a special form to create asynchronously running closures. Optionally, the programmer may specify the location where the closure is to reside. The closure may also include state variables so that it's behavior may vary over time.

PARALLEL: a parallel PROGN, evaluating the component forms concurrently.

PARALLEL-MAP: a parallel mapping function which applies a single function to multiple arguments at multiple locations, returning a list of the results.

MULTICAST-MAP: a parallel mapping function which evaluates the same form at multiple locations and gathers up the values returned in the order in which they are returned.

FUTURE: a special form specifying a form to be evaluated and the site at which the evaluation should take place. Returns 8 future encapsulating the value that will eventually be returned.

TOUCH/FORCE: a function to force a future to give up its value.

ON: evaluates a form at a specified location. Equivalent to (TOUCH (FUTURE . ..)).

In addition, CAREL supplies the following datatypes:

FUTURE-OBJECT: a datatype to encapsulate a value to be returned eventually after computing at a specified location

REMOTE-ADDRESS: a pointer to an object at a remote site

LOCATION: grid coordinates, neighbor/polar coordinates, or a keyword (:ANY, :ANY-NEIGHBOR, :ANY-OTHER)

The following describes the syntax of CAREL's functions and special forms, and gives illustrated examples of their use. Certain expressions are used repeatedly in the paragraphs that follow, so their definitions appear first:

location-form is any form that evaluates to something that can be interpreted as a location in the CARE network.

body is an arbitrary I ist of forms.

PARALLEL-LET:

(PARALLEL-LET ***parallel? bindings . body)***

parallel? is an arbitrary form, used to control the parallelism of the evaluation

bindings is a list of triples ***(variable*** *value-form location-form)*

As in QLAMBDA, ***parallel?*** is used to control whether the bindings should indeed be evaluated in parallel. ***If parallel?*** evaluates to () or #!FALSE, then the PARALLEL-LET is evaluated as an.ordinary LET, with the bindings being evaluated in (an unspecified) sequence, and the body being evaluated in an environment including those bindings.

If ***parallel?*** evaluates to T or #!TRUE, then the location-forms are evaluated concurrently and the concurrent evaluation of the value-forms is begun. The variables are immediately bound to the future-objects corresponding to the values to be returned, and the evaluation of the body is begun. The body may block temporarily on unfinished futures.

In all these cases, the value returned by the PARALLEL-LET is the (forced) value of the last form in the body.

PARALLEL-LAMBDA:

(PARALLEL-LAMBDA *parallel?* ***args*** *location-form state-bindings*

. body)

Evaluating a PARALLEL-LAMBDA sets up a closure at a remote site specified by *location* and returns a function of the specified arguments. When this function is applied, the list of evaluated arguments is sent to the remote closure, the remote evaluation is initiated, and a future is immediately returned. The remote closure created by PARALLEL-LAMBDA contains some state variables, bound in *state-bindings.* A state variable is changed by applying the PARALLEL-LAMBDA function to the arguments (:SET *variable-name value).*

parallel? is used, as in PARALLEL-LET, to determine whether parallelism is actually employed.

<u>PARALLEL:</u>

(PARALLEL . *body)*

The PARALLEL special form initiates the concurrent evaluation of the forms in the *body*. Control returns from PARALLEL when all of the forms have been evaluated. The value returned by PARALLEL is undefined.

<u>PARALLEL-MAP</u>:

(PARALLEL-MAY *function-form arguments-form locations-form)*

function-form evaluates to a function of one argument

arguments-form evaluates to a list, each member of which is to be used as an argument to the function

locations-form. evaluated to a list of locations.

PARALLEL-MAP, like MAP, applies a function repeatedly to arguments dra from a list and returns a list of results. Unlike MAP. PARALLEL-YAP performs the function applications remotely, and returns a list of futures that will eventually evaluate to the results.

<u>MULTICAST-MAP</u>:

(M ULTICAST-MA P *function-form locutions-form*)

MULTICAST-MAP invokes a function of no arguments at each location in a list of locations. MULTICAST-MAP immediately returns a list of futures corresponding to the values that will eventually be returned. Since the function called takes no arguments, the values returned can be different only if they depend on the local state of the processor at the location of evaluation, as embodied in the “global” environment of that processor.

MULTICAST-MAP-NO-REPLY:

(MULTICAST-MAP-NO-REPLY *function-form locations-form)*

MULTICAST-MAP-NO-REPLY invokes a function of no arguments at each location in a list, but does not cause results to be returned. The value returned by MULTICAST-MAP-NO-REPLY is undefined.

PIPELINE:

(PIPELINE *stage1 . . . stagen*)

where a stage ‘is:

(name args location-form state-vatiables . output-forms)

For each stage expression, PIPELINE establishes a remote-closure at the specified location. and then links the remote closures so that the output of one stage becomes the input of the next stage. The linked closures form the working part of the pipeline. PIPELINE then returns a function which, when applied, passes its arguments on to the first stage of the pipeline and immediately returns a future which will eventually contain the result that comes out of the pipeline. To ensure that the results that comes out of the pipeline correspond one-for-one with the sets of arguments that went in, the future-object to hold the result is created atomically with the entry of the arguments into the pipeline and is passed along with the data through the pipeline.

5. Some Examples

PARALLEL-LET:

```
;;; This subroutine concurrently performs trivial computations at the four
;;; corner neighbors of a given location and collects the results.
;;;
(define (cycle-corners-l start-location)
, (parallel-let t ((xl (list 1 2) (nelqhbor 0 start-location))
                   (x2 (list 3 4) (neighbor 2 (nelqhbor 1 start-location)))
                   (x3 (llst 5 6) (neighbor 3 start-location))
                   (x4 (list 7 8) (neighbor 5 (neighbor 4 start-locatlon))))
    (append xl x2 x3 x4)))

;;; CYCLE calls the subroutine starting at the current processor
;;;
(define (cycle) (cycle-corners-l *here*))
```

PARALLEL-MAP (see Figure 5-l):

```
;;; FOUR-CYCLE calls the CYCLE program at four different locations
;;; in the processor grid.
;;;
(define-(four-cycle)
  (parallel-map cycle-corners-l
                '((2 5) (5 2) (2 2) (5 5))
                '((2 5) (5 2) (2 2) (5 5))))
```

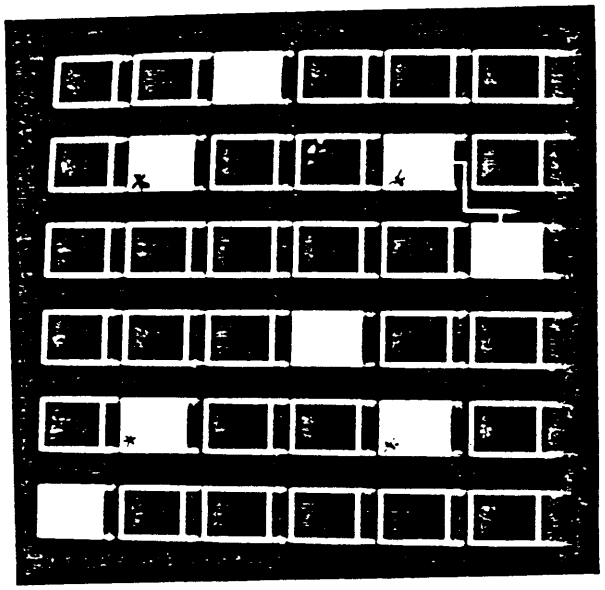

Figure 5-1: PARALLEL-MAP: Execution of the FOUR-CYCLE program. Active processors are displayed in inverse video. Active communications links are drawn as lines joing particular ports of the processor nodes. The processors hand-annotated with asterisks are the cycle centers. Each processor is at a different point in the cycle.

PARALLEL-LAMBDA:

```
;;; This creates a process at some other node in the network.
;;; returning an object which, when applied as a function to two
;;; arguments, evaluates a linear expression on those arguments.
;;;
(define (linear-evaluator al bl)
  (parallel-lambda t (x y) ':any-other ((a al) (b bl))
      (+ (* a x) (* b y))))
```

MULTICAST-MAP-NO-REPLY (see Figure 5-2):

```
;;; This activates the processor at each location in SITES.
;;;
(deflne (activate-locations sites)
  (multicast-map-no-reply (lambda () *here*) sites))
```

MULTICAST-MAP (see Figure 5-3):

```
;;; This sends a message to each location in the list SITES, asking It
;;; to return its location.
;;;
(define (Identify-yourself sites)
  (multicast-map (lambda () *here*) sites))
```

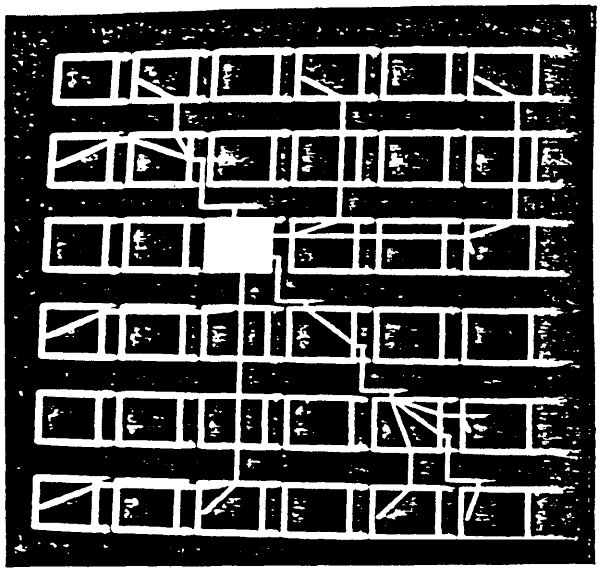
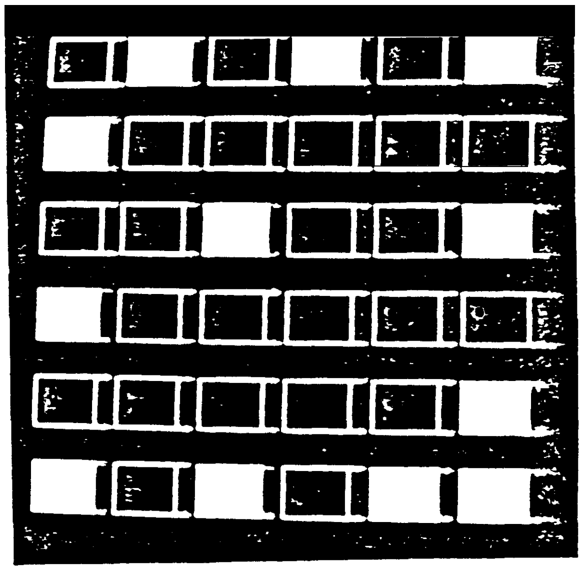

Figure 5-t: MULTICAST-MAP-NO-REPLY: Samples from the execution of the ACTIVATE-LOCATIONS program, showing how the multicast message is distributed and how the processors receiving the message are activated. Since no reply is required, the computation just dies out once the distributed programs are run.

PIPELINE:

```
;;; This sets up a pipeline across the bottom and up the right-hand
;;; side of the processor array.  This trivial pipeline simply adds
;;; 1 to the input value at each stage and passes the result on to
;;; the next stage.  It also prints out the result at each stage.
;;; using a printing mechanism "outside" the simulation.
;;;
(define (make-test-pipeline)
  (pipeline (s1 (x) '(1 6) ((a 1)) (print (+ a x)))
            (s2 (x) '(2 6) ((a 1)) (print (+ a x)))
            (s3 (x) '(3 6) ((a 1)) (print (+ a x)))
            (s4 (x) '(4 6) ((a 1)) (print (+ a x)))
            (s5 (x) '(5 6) ((a 1)) (print (+ a x)))
            (s6 (x) '(6 6) ((a 1)) (print (+ a x)))
            (s7 (x) '(6 5) ((a 1)) (print (+ a x)))
            (s8 (x) '(6 4) ((a 1)) (print (+ a x)))
            (s9 (x) '(6 3) ((a 1)) (print (+ a x)))
            (s10 (x) '(6 2) ((a 1)) (print (+ a x)))
            (s11 (x) '(6 1) ((a 1)) (print (+ a x)))))
```

6. Implementation

CAREL is implemented by a "semicircular"[1] interpreter, implemented in Zetalisp and drawing heavily on the CARE simulator. Some details Of the implementation are provided in the full paper. These include the' representation of CAREL datatypes, the use of a "global"

[1]Semicircular, not metacircular, because it is implemented in Lisp, but not in CAREL.

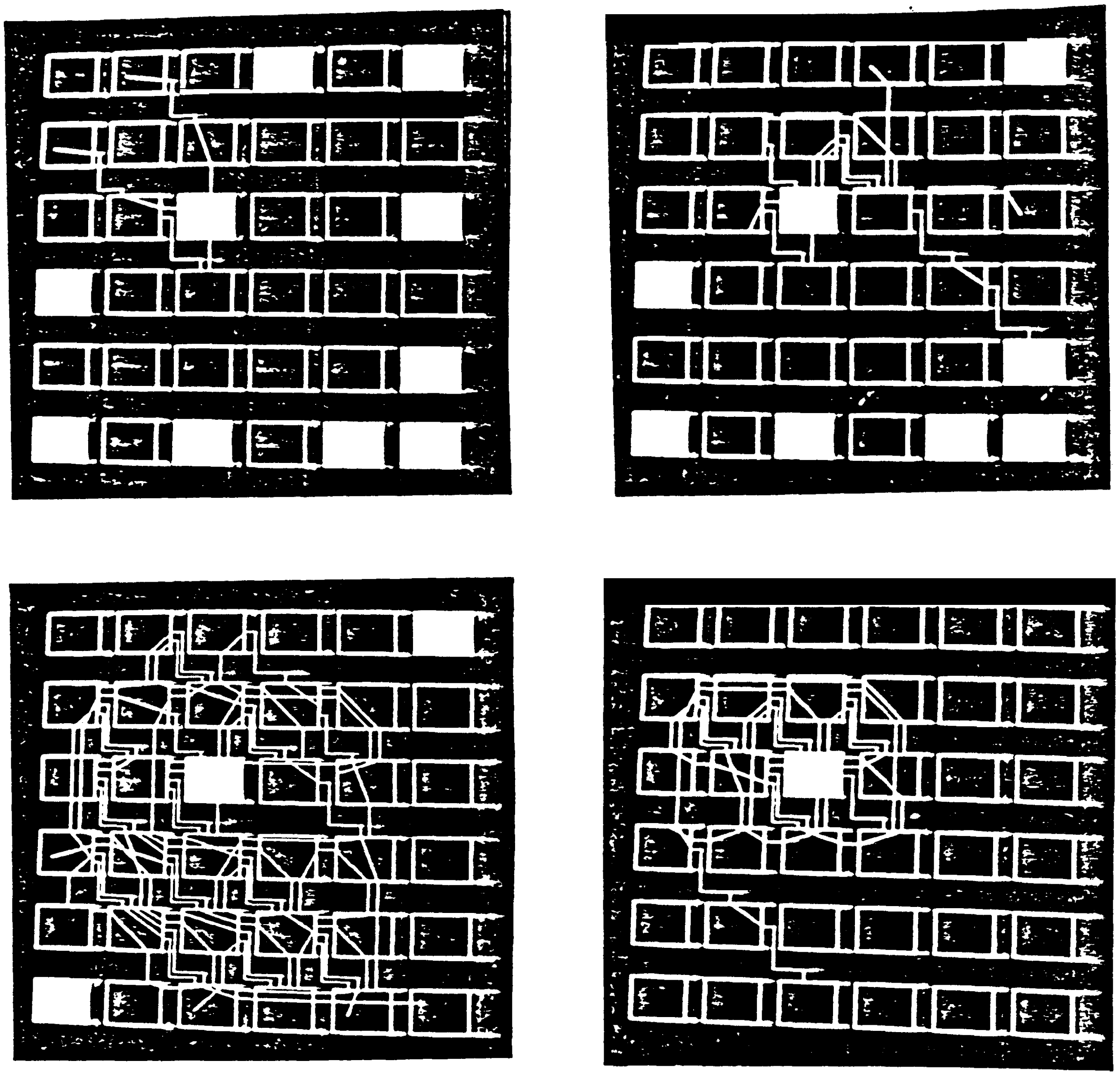

Figure 5-3: MULTICAST-MAP: Samples from the execution of the IDENTIFY-YOURSELF program. The multicast method is distributed as in Figure 5-2, but in this example the processors must send a value back to the requesting process, The network becomes congested as all the processors respond then gradually returns to rest as the messages reach their destination. The notion of a network "hot-spot" is clearly demonstrated.

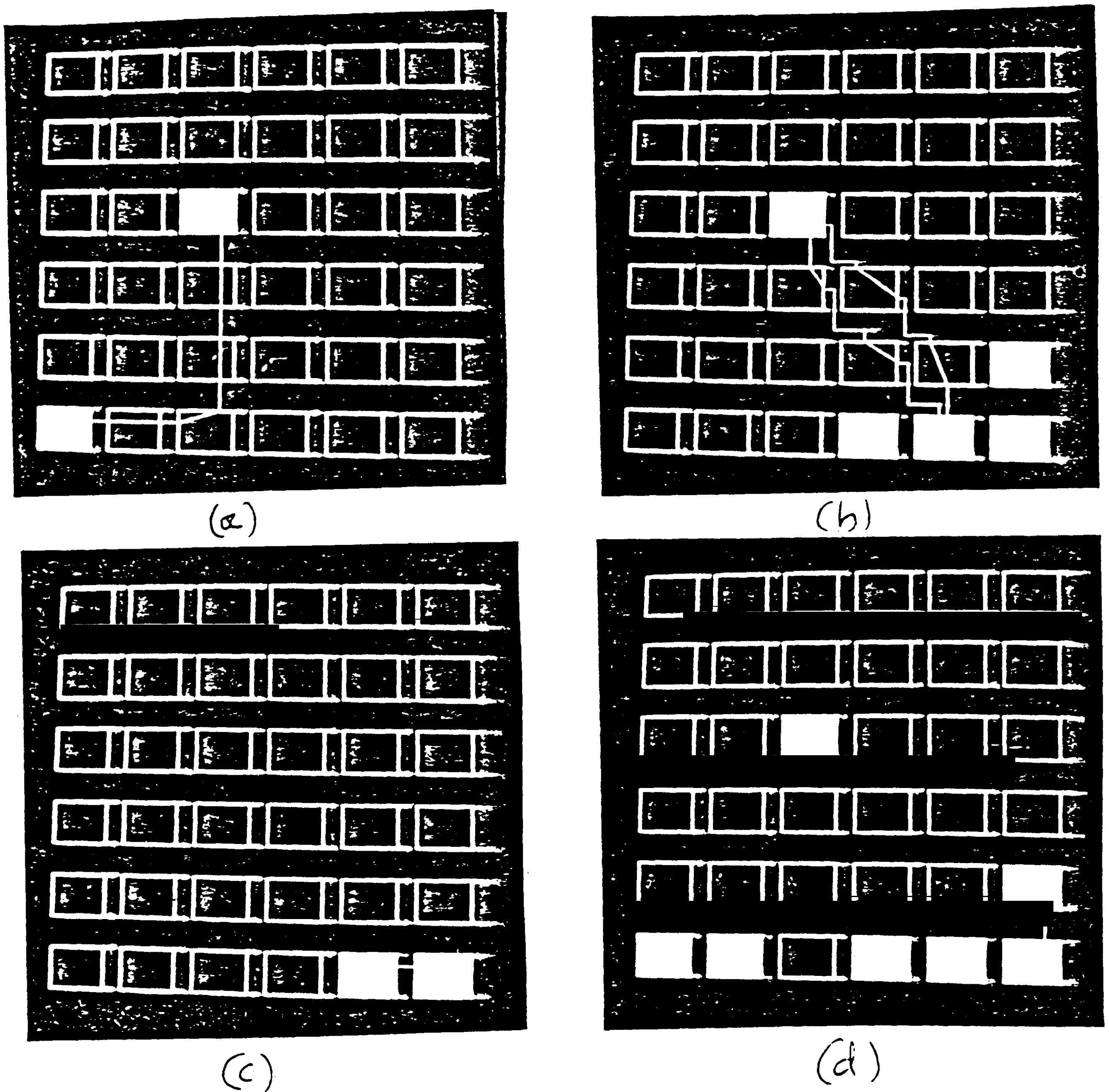

Figure 5-4: **PIPELINE**: Samples from the execution of programs constructing and using a CAREL software pipeline. The pipeline runs along the bottom and up the right side of the processor array. The pipeline is constructed in two passes- The first pass (a) establishes a process at each site and the second pass (b) links the processes together. The execution of the pipeline on a single argument (c) shows data flowing through the pipeline using only local communication. The last figure (d) shows multiple data items may flowing through the pipeline simultaneously, keeping multiple processors busy.

environment (full copies of which exist at each processor) and processor-local environments, and the interface to the CARE hardware simulator.

7. CAREL and Other Languages

CAREL was strongly influenced by three other languages: QLAMBDA [Gabriel and McCarthy 84], Par-Alfl [Hudak and Smith 86], and Actors [Agha 85]. QLAMBDA provided the idea of having two kinds of parallelism (which Filman and Friedman called parallelism by lexical elaboration and parallelism by explicit processes). CAREL addresses the question, "What would QLAMBDA look like on a distributed-memory multiprocessor?**.

Par-Alfl provided the notion of a dynamic variable $self that a process could use, reflectively, to determine where it was executing. The part of CAREL chat implements parallelism by lexical elaboration is very similar co Par-Alfl. CAREL adds the ability to deal with processes as first class objects.

Actors continues to serve as the "right thing" in the domain of languages for parallel symbolic computing. Calculating the difference between what CAREL can do and what Actors should do is always a valuable source of ideas for improvement. CAREL provides one particular set of primitives for describing both concurrency and locality. These primitives are powerful enough to implement a wide variety of interesting programs, but still provide less concurrency, less capability for managing synchronication, and less theoretical elegance than Actors. For example, CAREL enforces synchronization at the inputs and outputs of a function or closure: when APPLY is invoked, all the arguments must have been pre-evaluated, and multiple outputs are considered to be generated in a single list. In the Actor language SAL described by Agha, the inputs to an Actor may arrive at any time and in any order and outputs likewise may be generated asynchronously. Furthermore, Actors promise to make process management as invisible as memory management is in Lisp.

The plan for CAREL is to migrate it toward an Actor language. The CARE architecture is very close in spirit to the Actor approach. and would provide a nearly ideal environment for implementing Actors.

8. Acknowledgements

Implementation of CAREL was made possible by the existence of the CARE simulator, as implemented by Bruce Delagi and augmented by Eric Schoen. The author further wishes to acknowledge the intellectual support of the Stanford Advanced Architectures Project. Contributors to PARSYM, the netwide mailing list for parallel symbolic computing, have provided fruitful stimulation.

References

[Abelson and Sussman 85]
Harold Abelson and Gerald Jay Sussman with Julie Sussman.
Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, Massachusetts, 1985.

[Agha 85] Gul A. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
Technical Report, MIT AI Laboratory, March, 1985.

[Davis and Robison 85]
A. L. Davis and S. V. Robison.
The Architecture of the FAIM-1 Symbolic Multiprocessing System.
In *Proceedings of IJCAI-85*. 1985.

[Delagi 86 3 Bruce Delagi.
CARE User's Manual
Heuristic Programming Project, Stanford University, Stanford, Ca. 94305, 1986.

[Filman and. Friedman 84]
R. E. Filman and D. P. Friedman.
Coordinated Computing: Tools and Techniques for Distributed Software.
McGraw-Hill, New York, 1984.

[Gabriel and McCarthy 84]
Richard P. Gabriel and John McCarthy.
Queue-based multiprocessing Lisp.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984. 1984.*

[Halstead 84] Robert H. Halstead.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.* ACM, 1984.

[Hudak and Smith 86]
P. Hudak and L. Smith.
Para-functional programming: A paradigm for programming multiprocessor systems.
In *Proceedings of ACM Symposium on Principles of Programming Languages, January 1986.* ACM, 1986.

[Shapiro 84] E. Shapiro.
Systolic programming: A paradigm of parallel processing.
In *Proceedings of' the International Conference on Fifth Generation Computer Systems.* 1984.