A COMPUTER SYSTEM PROVIDING MICROCODED APL

Charles A. Grant
Mark L. Greenberg
David D. Redell

Center for Research in Management Science

University of California

Berkeley, California

A new computer system is now under development at the Center for Research in Management Science (CRMS) of the University of California, Berkeley. It is an inexpensive, medium-scale time-sharing system, whose primary application is the implementation of multiterminal, interactive simulation experiments for the purpose of social science research. This paper describes the APL-language subsystem implemented on this system.

The CRMS APL language system includes a microcoded APL interpreter which is implemented on a high-speed microprocessor. In addition to an extremely high rate of execution, CRMS APL offers a unified facility for terminal input/output, file accessing, and multi-process synchronization and inter-communication in APL.

A COMPUTER SYSTEM PROVIDING MICROCODED APL

Charles A. Grant
Mark L. Greenberg
David D. Redell

Center for Research in Management Science

University of California

Berkeley, California

Introduction

The Management and Behavioral Sciences Laboratory of the Center for Research in Management Science (CRMS)[†] is funded[††] to further computer-aided research in the social sciences. The major emphasis has been computer controlled, multi-subject simulation experiments. In 1970, it was decided to replace the Laboratory's aging computer facilities in order to allow experiments to be programmed in the APL language. An important requirement was rapid response to computationally demanding experiments involving up to 32 terminals. The possibility of developing an APL interpreter implemented in microcode was investigated [1].

[†]26 Barrows Hall, University of California, Berkeley, California 94720.

[††]NSF Grant NSF-GS-32138.

With the objective of providing flexible computer facilities which could meet the ever-changing needs of a research laboratory, work was begun in the spring of 1972 on a general-purpose time-sharing system which would include an APL-language subsystem employing a microcoded interpreter. The configuration of the CRMS Computer System is shown in Figure 1.

The function of the central processor is to implement a general-purpose time-shared operating system performing the functions of input/output management, memory management, scheduling of user programs, and provision of system library services. The current system includes 64k 32-bit words of core memory, two 12-million-word disk units, one tape drive, and provisions for the connection of up to 64 terminals. The APL processor is connected by "start" and "stop" signals to the central processor and has access to the core memory.

The central processor and the APL processor were both implemented as microprograms for four reasons:

(1) To provide flexibility in deciding which features of the system should be optimized.

(2) To allow efficient inclusion into the architecture of the system of such features as protection by capabilities, a large virtual address space, and process (software task) synchronization [2].

(3) To allow for microcode implementation of most of the (normally hardware-implemented) functions of peripheral device control units in order to reduce system complexity, cost, and maintenance expense. In particular, the terminal multiplexor,

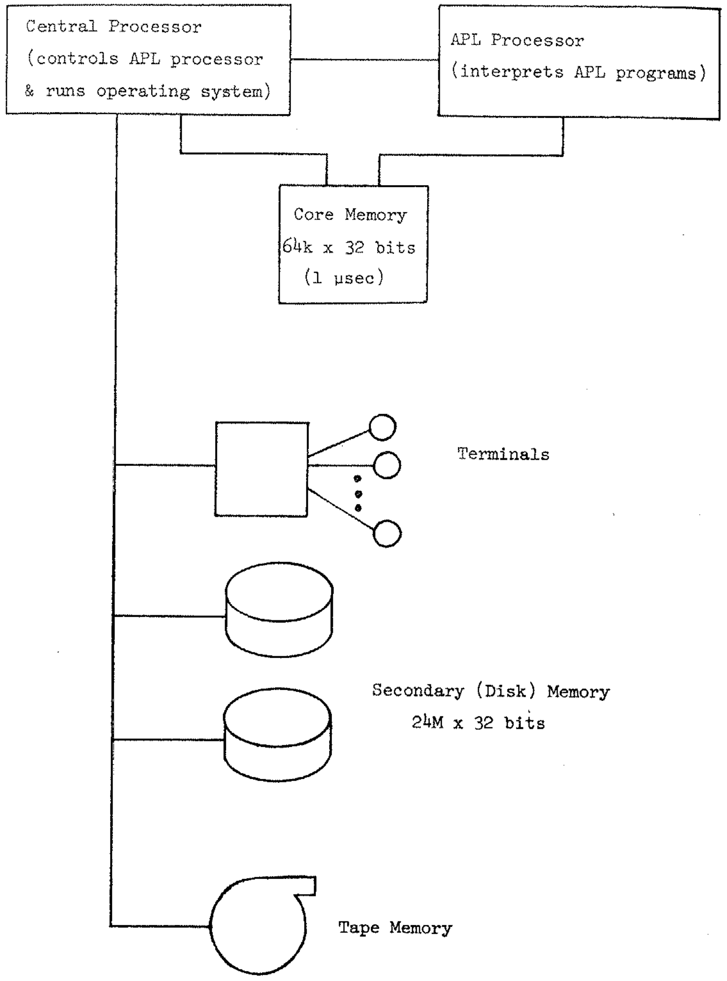

Figure 1.
CRMS Computer System Configuration

disk channel, printer channel, and tape channel are implemented as subroutines of the central-processor microprogram.

(4) To eliminate the interface problems which might have arisen had two dissimilar hardware processors been used.

The central processor and the APL processor are microprogrammed on separate Digital Scientific Corporation META4 computers. The micro-instruction time is 90 nanoseconds.

CRMS APL

The CRMS APL language is essentially the same as APL\360 [4], but does differ from it in four fundamental ways:

(1) Function parameters: CRMS APL provides for the passing of up to 15 arguments to a niladic, monadic, or dyadic function. For example:

```
A ← B F[X;Y;Z] C
```

says, "Call dyadic function F with three extra arguments, X, Y, and Z." Extra arguments may be classified as optional extra arguments in the function header. The calling function need not supply the parameters corresponding to optional extra arguments.

```
A ← B F[;Y;] C
```

is one such call. In this case, the formal parameter corresponding to the first and third extra arguments have the "undefined" (value error) value. A primitive function is provided which detects the presence of the "undefined" value without causing an error.

(2) Parameter passing: The ability to pass parameters by reference is included in CRMS APL. Unlike normal arguments, an argument passed by reference may be modified by the called function. Specification that a parameter is to be passed by reference is made in the function's definition header.

(3) Scope of names: Rather than "dynamic localization" of names as used in APL\360, CRMS APL uses "static localization." Thus, in CRMS APL a local variable of a function may be referenced only within the definition of that function and may not be referenced from a subsequently called function, as in APL\360. This change allows for more efficient interpretation of APL programs, yet imposes very little limitation on the APL programmer.

(4) Mixed arrays: The elements of a CRMS APL array may be any mixture of numbers and characters. This feature, described by Iverson [3], was omitted from APL\360.

Before execution is initiated on a CRMS APL program, it is translated into the "object language," which is processed by the APL processor. The translator is a program which runs on the central processor. This translation process includes the conversion of decimal numeric constants into internal form, the conversion of symbolic names into memory addresses, and the parsing of each statement into postfix Polish form. The parsing of statements before execution time is possible as a result of the "static localization" rule for determining the interpretation of a name. Parsing must be done at runtime with dynamic localization as used in APL\360 because correct syntactic analysis of a statement sometimes depends upon the exact sequence of function calls which led to the statement's execution [5].

Once translated, an APL program may be submitted to a module of the operating system called the APL Manager, which schedules the use of the APL processor among all the "ready" APL programs. The APL Manager communicates with the APL processor by using the "start" and "stop" signal wires and memory cells designated for inter-processor messages.

The APL processor is a microprogram (approximately 2000 32-bit microinstructions) which is divided into two parts: controller and executer. The controller idles until it receives a start signal. Upon receiving a start signal, the controller looks in the message cells for the addresses describing the location in core memory of the code and data "segments" of the program (q.v.). The current state of the program (e.g., program counter, stack length) is then loaded, and execution is begun. The controller is invoked again if a "stop" signal is received, if an execution error occurs, or if the running program executes an instruction requiring intervention by a central-processor program. At this point, the controller saves the state of the program and sends a signal and the appropriate message to the central processor.

The executer is designed such that no program error on the part of either the user or the translator can cause any memory accesses to be made outside the areas designated as the code and data segments. The executer essentially executes three types of CRMS APL instructions: function (call and return) instructions, stack instructions which push and pop operands on the stack, and operator instructions which operate on these operands. All operands are described by a 32-bit descriptor, where the descriptor is a normalized 32-bit floating-point number, a 24-bit integer, an 8-bit character, an indirect parameter word, the so called "undefined" value, or a pointer to an array. An array pointer locates the data for the array, as well as information which describes its shape. The instructions check the shape of their argument(s) for legality and then perform the indicated function on all elements of the argument(s). Such a procedure may involve allocation and/or de-allocation of array storage. The executer includes a microcoded dynamic storage allocator, which utilizes a designated area of the data segment. Reference counts are maintained on each block of array storage.

Since the large number of APL primitive functions could not all fit into the control storage of the APL processor, it was necessary to select a "base set" of primitive functions which would fit. Other primitive functions are implemented by the translator as "open" or "closed" subroutines (APL defined functions) which simulate the desired primitive functions. The most commonly called functions of typical programs are included in the chosen base set. Some space has been reserved in the control store of the APL processor for the implementation of other primitive functions which later evaluation might indicate should be included in the base set. The current base set is listed in Figure 2. Figure 3 shows some timing figures for execution of primitive functions both inside and outside the base set.

ASSIGN
ASSIGN INDEXED
ASSIGN NO RESULT
BRANCH
CATENATE
CEILING
CONVERT
DIFFERENCE
EQUAL
FLOOR
FUNCTION CALL
GET ORIGIN
IDENTITY
INDEX
MONADIC IOTA
LESS
LOGICAL PRODUCT
LOGICAL SUM
MAGNITUDE
NEGATIVE
NOT
PRODUCT
QUOTIENT
RAVEL
REFERENCE
RESHAPE
RETURN FROM FUNCTION
SET ORIGIN
SHAPE
SUPERVISOR CALL
TEST DEFINED
TEST NUMBER

Figure 2.
CRMS APL "Base Set" of microcoded primitive functions

```
∇ TIMING∆TEST; COUNT; FP; INT; T
  [1]          COUNT ← 0
  [2]          INT ← ⍳ 100
  [3]          FP ← INT + 0.1
  [4]    LL:   ⍝ TEST STATEMENT
  [5]          COUNT ← COUNT + 1
  [6]          → (COUNT < 10000) ρ LL
∇
```

Statement [4] of Test Function	Compute Time for Statement [4]	Compute Time for Entire Function
LL: ⍝ NULL LINE	--	1.3 SEC
LL: T ← INT	0.02 MSEC	1.5 SEC
LL: T ← ρ INT	0.07 MSEC	2.0 SEC
LL: T ← FP , INT	0.69 MSEC	8.2 SEC
LL: T ← INT + INT	0.70 MSEC	9.0 SEC
LL: T ← FP + FP	0.97 MSEC	11.0 SEC
LL: T ← INT × INT	2.1 MSEC	22.5 SEC
LL: T ← FP ÷ FP	2.7 MSEC	28.0 SEC
LL: T ← FP × FP	2.7 MSEC	28.0 SEC
LL: T ← INT + . × INT	8.1 MSEC	82.5 SEC
LL: T ← INT ∘ . × INT	8.4 MSEC	85.0 SEC
LL: T ← 0 ∊ INT	26. MSEC	260. SEC
LL: T ← + / INT	29. MSEC	290. SEC

Figure 3.
CRMS APL Timing Experiment

APL Runtime Supervisor

The APL Runtime Supervisor (ARS) is a central-processor program which provides an interface between APL programs, which run on the APL processor, and the outside world. It provides a command language by which a user can create, run, and debug APL programs. In addition, it provides mechanisms for

(1) creation of multiple processes (parallel tasks) under control of a single user,
(2) communication and synchronization among these processes,
(3) input/output to terminals,
(4) input/output to sequential files stored on disk memory.

A collection of one or more cooperating processes, all created by a user and executing on behalf of that user, is referred to as an experiment. Each process consists of a "code segment," which contains the object language of the program, and a "data segment," which contains all variable information associated with the process. In particular, the data segment includes:

(1) storage for global variable values,
(2) array storage,
(3) a storage stack for local variable values, function call information, and temporary results,
(4) storage for the current state of the process.

All processes within an experiment share the same code segment, but each process has a separate data segment; thus each process can be thought of as a separate parallel execution of the same program. An experiment is first initiated with a single process. Any process within an experiment may create a new process and start it executing by making a call on the ARS.

Processes may send or receive messages to or from other processes, terminals, or sequential files through objects implemented by the ARS called "mailboxes." The processes in an experiment may create any number of mailboxes. Any process in the experiment may access any mailbox by calling the ARS. A mailbox contains a first-in-first-out queue of messages. Each message is a single APL value, i.e., a scalar or an array of any rank. A process may put (send) a specified value into a specified mailbox or take (receive) the next message out of a specified mailbox. If a mailbox is empty when a receive is done, the receiving process will wait until a message is sent to the mailbox before it continues execution. A sending process may specify whether it should wait until the message it sends is received before continuing execution. These rules provide a general interprocess-synchronization mechanism. A process may also attach a terminal or a sequential file to a mailbox by calling an ARS function. A value sent to such a mailbox is automatically removed from the mailbox and output to the terminal or file. If a receive is performed on such a mailbox, the next value in the file or the next value typed at the terminal is received by the process.

This uniform interface provides great flexibility. For example, an experimenter can substitute a process ("robot" subject) for a terminal (real subject) without modifying his programs at all.

The operating environment provided with CRMS APL includes special facilities for orderly debugging of such systems of parallel processes.

Status

As of January 1973, the hardware for the system was operational. Now, January 1974, the APL subsystem is operational and has been used to develop a prototype experiment. The simulated APL primitive functions, including mathematical routines, are mostly completed. The current version of the operating system supports only one experiment at a time. Preliminary versions of the editing and debugging facilities are available. Work in progress includes integration and polishing of an easy-to-use interactive APL subsystem, and design and implementation of a multi-user time-shared operating system. Completion of the development is scheduled for summer of 1974.

Acknowledgments

This project is administered by Professor F. E. Balderston, Chairman of the Center for Research in Management Science, and the Laboratory Advisory Committee, of which Professor A. C. Hoggatt is Chairman. Many persons have contributed to the technical development, including Paul Gee, Wiley Greiner, James Harp, Ross Harrower, David Koch, Paul McJones, and George Morrow.

References

[1] Zaks, R., D. Steingart, and J. Moore, "A firmware APL time-sharing system," AFIPS Conference Proceedings, vol. 38 (1971 SJCC), pp. 179-190.

[2] Denning, Peter J., "Third Generation Computer Systems," Computing Surveys, vol. 3, no. 4 (Dec. 1971), pp. 175-216.

[3] Iverson, K. E., A Programming Language, Wiley, New York, 1962.

[4] Pakin, Sandra, APL\360 Reference Manual, 2d ed., Science Research Associates, Chicago, 1972.

[5] Hassit, A., J. W. Lageschulte, and L. E. Lyon, "Implementation of a High Level Language Machine," Communications of the ACM, vol. 16, no. 4 (Apr. 1973), pp. 199-212.