Functional Parallelism in an Operand State Saving Computer

J.B. Harvill
Department of Computer Science
North Texas State University
Denton, Texas 76203

Abstract

Multiple, high-level operators are assigned to a general operand. The operators are implemented with individual micro-processors and are "attached" dynamically to the memory location representing the "current" value of the operand. The operators then asynchronously use each new operand value as it is stored and perform their operations in parallel. The proposed architecture represents a true M I S D (Multiple Instruction Stream - Single Data Stream) computer. Its architecture can provide effective parallelism and reduced programming complexity for a large class of both numeric and non-numeric computer problems.

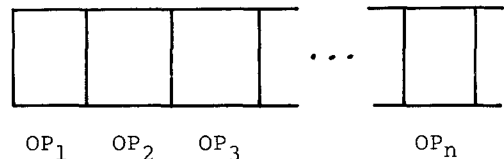
Operand State Saving Computer

The operand state saving computer is an effort to create a computer whose organization more closely simulates a portion of the human brain's apparent memory structure [1, 2]. A primary attribute of the von Neumann computer is that the input and assignment capabilities cause an irreversible change of state of memory; but human memory doesn't work this way. The concept of a destructive store is alien to human cognition [1, 2]. When humans assign a meaning to a symbol, we do not expect that meaning to be in effect only until the next assignment to the same symbol [3]. Even if a symbol's value has been superceded, humans are still able to remember the earlier values, their approximate time boundaries, and order.

The research of Nielsen [2] and Underwood [1] clearly indicates that human memories are temporally ordered and temporally tagged. Thus, it appears that for the computer to simulate the human memory process, the concepts of time and successive memory states are necessary, with it being desirable for a computer to automatically maintain historical information related to its activities. With such a computer, the program's historical information can be used without the need for the programmer to explicitly provide for its acquisition, storage, and maintenance.

The operand state saving computer then is one that automatically saves its previous operand values. The number of previous values of each operand saved depends upon the requirements of the program, the form and amount of memory available, and the implementation of the state saving mechanism (hardware, software or a combination of both). A hardware design for an operand state saving computer, REVEX, is described in [4].

A program operand, OP, now can be represented as an array of memory locations, where OP_1 contains the value of the first definition of the variable OP during the current execution of the program; OP_2 contains the value of the second definition; OP_3 the value of the third; and so forth, until OP_n, which contains the last and current value of the variable OP.



Further, the system maintains automatically the value of n for each of these operands.

As the program executes, OP may be defined to have new values and the "past" values of the operand are automatically maintained by the system; not lost as they are by today's computers.

Programming Language Evolution

The "current state" von Neumann computer forces a programmer to "know" at the time a value is created that the program must save that value for future use. Values needed for future use can only be remembered if the programmer chooses <u>different names</u> to refer to each

different past state of the operand.

Programmers must then explicitly create the program to use arrays to store the historical values of an operand. These individually named, historical values of the operand, A(1), A(2), ..., etc., are not now a one-to-one mapping of problem quantities to program operands that is desirable. Instead, the program represents a one-to-many mapping. The control and use of this forced multiple mapping can seriously increase the complexity of an algorithm representation.

Because the operand state saving computer does automatically maintain the past values of an operand, it allows the associated programming language to take an important evolutionary step towards the capability of natural language by facilitating the use of attributive, tensed qualifiers in place of the typical, more complex subscripts:

PREVIOUS A	instead of	A(N-1)
FIFTH A	instead of	A(5)
NEXT A	instead of	N=N+1 "generate" A(N)

The ability of a language to communicate about things that are out of sight, in the past or future, or even non-existent is called displacement. Hockett and Ascher [5] list displacement as one of the mandatory characteristics of a natural language. Harvard's Roger Brown [6] goes further, stating that displacement may be the most critical property of natural language. Because non-immediate results constitute the major part of what humans consider to be reality, one of the major functions of language is to make this experience available for use [1].

The automatic availability of these past operand values also makes plural addressing very easy and natural:

LAST 5 A'S instead of: A(N), A(N-1), A(N-2), A(N-3), A(N-4)

ALL A'S instead of:

```
DO # J=1,N
# ...A(J)...
```

The simplified plural addressing makes the use of plural operators easy and natural:

SUM OF ALL A'S instead of:

```
SUM=0
DO # J=1,N
# SUM=SUM+A(J)
```

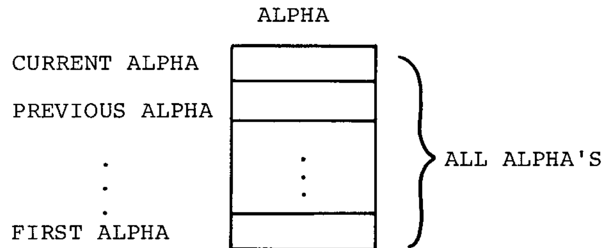
IF (ANY A = 10) ... instead of:

```
DO # J=1,N
IF(A(J).EQ.10)...
# CONTINUE
```

Studies comparing tensed programming language constructions with typical high level language constructions show that the use of the tensed constructions can reduce the representation complexity of a large class of algorithms by approximately 75% [7].

General Variables [8]

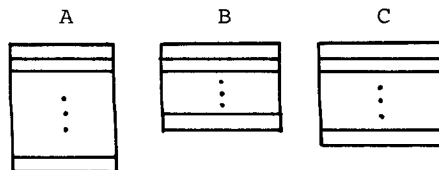
Pictorially, an operand of the state saving system,ALPHA, can be visualized as an array of memory cells, each containing one of the values of ALPHA:



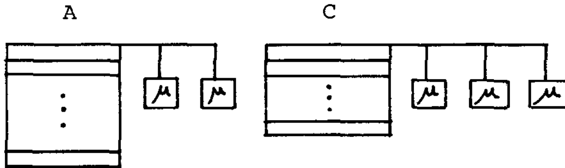
A newly created value of the general variable ALPHA will be "pushed" onto the array, logically moving all of the previous values of ALPHA down one location in the array. The creation of a new value of ALPHA also causes the system maintained "count-of-ALPHA" to be updated to the correct value. Now any value of ALPHA can be addressed by using the proper natural language qualifier in the attributive position of the general variable name, ALPHA.

Organization of the Functionally Parallel Computer

Consider an operand state saving computer which supports general variables in addition to destructive store scalar variables. For the general variables, the system saves, counts and makes addressable the last n_i values of the general variable V_i . The operand space of a program using general variables A, B, and C can be pictured:



For each high level operation to be performed on a variable, both scalar and general, it is desired to dynamically "attach" a micro-processor performing the desired operation to the variable. The system will then use the definition of a new value of the variable to trigger the associated micro-processor to begin performing its high level operation while the primary control stream continues with the execution of the program.



It is obvious that high level operators have different operand requirements for input and output. Most are more complicated than are pictured in the diagram above. Before continuing, consider some distinctions between operators that are based upon the types of input and output operands they require.

Classification of Operators

The high level operators will be grouped according to the types of input and output operands they require or tolerate.

Operands can be classified into the following groups:

1. Single scalar value

2. Single general value

   (Multiple values of the same logical problem variable.)

3. Plural scalar values

4. Plural general values

Operators must be considered that can utilize any of these four forms as input and which can produce any of these four forms as output. The computing system must allow the micro-processors the ability to define values in the primary memory of the computer. The micro-processors must be able to both receive values from the primary memory and to feed the results of their operation back to the primary memory for storage.

Consider examples of high level operators which can use the different possibilities of input variables and output variables.

Group 1 - Input Scalar.

Output	Operators
Scalar	Sine, Square root
General	Index set generation
Plural Scalar	Attribute generation
Plural General	Random number generation Strobed high level oper.

Group 2 - Input General.

Output	Operators
Scalar	Average, Definite Integ.
General	Moving average Finite difference
Plural Scalar	Attribute generation Statistics
Plural General	Distribution Difference table

Group 3 - Input Plural Scalar.

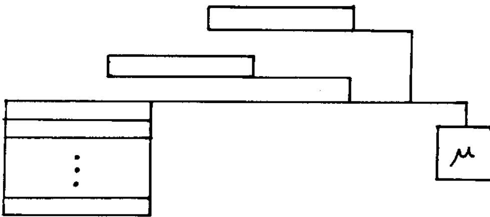
Output	Operators
Scalar	Maximum, minimum
General	Polynomial evaluation
Plural Scalar	Attribute governed selection
Plural General	Index generations

Group 4 - Input Plural General.

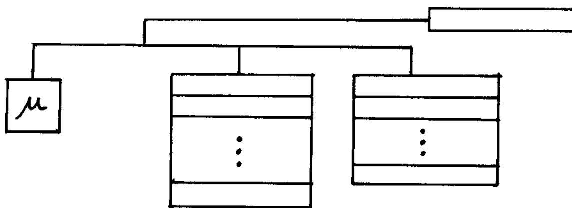
Output	Operators
Scalar	Correlation coefficients
General	Correlation function
Plural Scalar	Multiple regression
Plural General	Function filtering

Because one goal of this research is to conserve the asynchronous character of the system, only groups 1 and 3 are considered for implementation. The examples listed under groups 2 and 4 above will be implemented assuming the data to be acquired "On-the-fly".

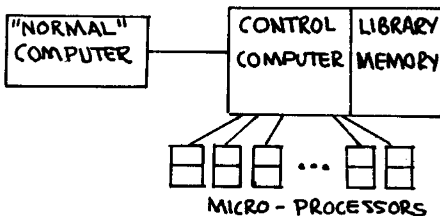
There is a need then to be able to "attach" the micro-processors to either single or plural, general or scalar variables for input:



There is the similar need for the micro-processors to be able to create output values and send them to either singular or plural, scalar or general variables.

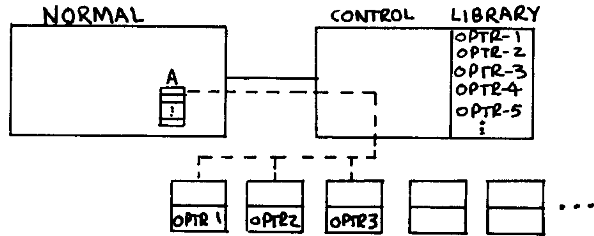


The system supporting parallel computation will then consist of the "normal" computer which supports general variables in a primary memory and a control computer connected to a library memory which in turn is connected to a resource of micro-processors, each with their own working memory:

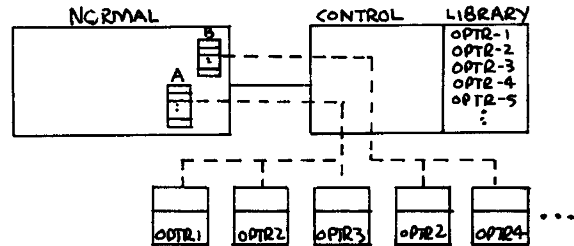


The library memory contains a program to implement each of the high level operators supported by the system and each of the micro-processors is capable of receiving and executing any of the high level operators.

Assume, for example, that the main program requires operation-1, operation-2, and operation-3 be performed on all the values of operand-A. The copies of the library programs for these three operations will be written into the working memories of the first three micro-processors and linkages will be set up for the input and output values of each operator.



In addition, assume that the program requires operation-2 and operation-4 be performed on general variable-B. Copies of these two operators will be written into the working memories of the micro-processors 4 and 5 and appropriate input/output linkages set up between the control computer and primary memory



The system now performs the following task during execution of the primary program: When a value is defined for operand-A, that definition causes a copy of the new value to be sent to the buffer memory of the control computer using an address that is associated with the general variable A during compilation. The re-definition of this location in buffer memory triggers the micro-processor through one cycle of its operation.

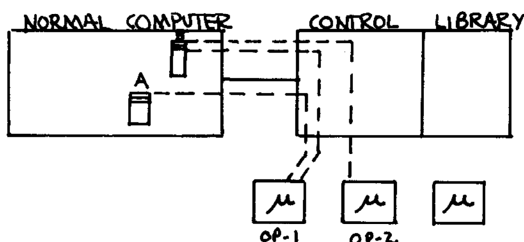
Variable length buffering can be used for the values in the control computer's memory. The size of the buffers is a function of local program economics and the operators involved. When the buffers are full, the storage of information into the general variable must be stopped until there is buffer space available in the control computer.

Thus, as each new value of one of the variables is created and stored in the memory of the primary processor, it is also received by the control computer which feeds it to the appropriate micro-processors for that particular general

variable. The micro-processors then *perform their operations in parallel* while the "normal" computer continues with its program creating new values of other variables. The arrival of a new value of a variable at the control computer is the asynchronous trigger which causes the operator micro-processors attached to that variable to perform their operations using that new value.

The input-output buffers for each micro-processor can be either in the control computer's memory or in the individual work space of each micro-processor.

High level operator micro-processors can also be "attached" to the output operands of other operator micro-processors so that nested functions can be calculated thus creating many possible levels of parallel operation.



These concepts are an example of a Direct Shared Memory Multiprocessor for the operator definition and operand control function [9]. A DSB, Direct Shared Bus (global bus), controlled by a micro-processor can provide a hardware solution to the requirements of the system.

M I S D Configuration

The above computer system is an example of a true M I S D (Multiple Instruction Stream - Single Data Stream) computer as defined by Flynn [10]. Examples of M I S D configurations in the literature have been weak [11] (CDC Star) or of older manually operated equipment [10].

Language Syntax

Computer languages are a function of:

1. Our human traits
2. Our cognitive processes
3. Our natural languages
4. The problems being solved
5. The calculating machines available

Virtually all computer languages *today are operator oriented. This seems* to be the case because the primitive operators understood by our computing systems are of much lower level than the operators needed to solve our problems. The process of programming becomes a constructive process using the computer primitives in various combinations and sequences to cause the computer to perform the higher level operations required by the problem. Computer languages supporting this constructive process are naturally oriented to the operator instead of the operands.

```
Operator(oprnd-1, oprnd-2, ..., oprnd-n)
```

In fact, the standard module of a computer program today is the procedure call:

```
PROC(arg-1, arg-2, ..., arg-n)
```

Multiple Operations

Commands which specify multiple operations can be organized in two ways:

1. Operator oriented with multiple operands specified.

   Example: "Paint the fence, barn, and house."

The verb-object grouping can either imply a precedence ordering for the execution of the operator or not, depending upon the particular operator and the particular operands involved.

2. Operand oriented with multiple operators specified.

   Example: "Wash, sand, and paint the door."

   "Wash, wax, and tune the car."

The door example infers an ordering of the multiple operations, while the car example infers only a partial necessary sequence. The semantics of each verb will determine its ability to be performed as a function of its context with respect to the other operators and its operand.

The form chosen in natural language seems to depend upon the primary interest of the command giver. If the primary interest is in the operators, then there is a tendency to use operator oriented syntax. However, if the main interest is in the data, the operand oriented structure is often used.

There are many computer users who are much more interested in their data than they are in spending their time

specifying higher level abstractions in a computer program. They simply want to take their data, apply several high level operations to the data, and study the results. Examples of these types of users are those who want to perform statistical analyses upon data, i.e. medical researchers, sociologists, business analysts, psychologists, etc. Many users also want to take streams of data and perform modifying operations upon them. Examples of this type are seismic analysts, text processors, compiler writers, etc.

For such users it would appear that a computing system which supports an operand oriented language will provide an easier computing facility to use.

Two forms of syntax are useful in such a language:

1. Form one is the counterpart of the typical procedure call today:

OPERAND(oprtr-1, oprtr-2, ..., oprtr-n)

2. Form two is more like the natural language use of object oriented sentences:

oprtr-1, oprtr-2, ..., oprtr-n(OPERAND)

Both forms are parsable and have equivalent representation complexity.

Example

1. READ ALL A'S

   AVG, STD, VAR(A'S)

   WRITE STD, AVE, VAR

Representation Complexity

The operand oriented syntax presents the user with a significant reduction in representation complexity for appropriate applications.

Consider the previous example:

Y = AVG, STD, VAR(B'S)

In an operator oriented syntax the command would be written:

Y = AVG (B'S)

Y = STD (B'S)

Y = VAR (B'S)

Calculating Halstead's Software Science [12] measures of algorithm complexity for each case:

Case 1. Operand oriented syntax

Y = AVG, STD, VAR (B'S)

Operations: = , AVG, STD, VAR, (group), End of Stmt.

Operands: Y, B'S

$n_1 = 6$ $N_1 = 6$

$n_2 = 2$ $N_2 = 2$

$n = 8$ $N = 8$

Volume = 24 Level = .33

Difficulty = 3 Intelligence = 8

Effort = 72

Case 2. Operator oriented syntax

Y = AVG(B'S)
Y = STD(B'S)
Y = VAR(B'S)

Operators: = , AVG, STD, VAR, (group), End of Stmt.

Operands: Y, B'S

$n_1 = 6$ $N_1 = 12$

$n_2 = 2$ $N_2 = 6$

$n = 8$ $N = 18$

Volume = 33.36 Level = .11

Difficulty = 9 Intelligence = 3.67

Effort = 300.24

Thus, where three operations are to be performed on the same general variable, the ratio of effort between the operand oriented syntax and the operator oriented syntax is 72/300 = .24 . This indicates that the operand oriented notation requires only 24% of the effort for the user to create than today's typical operator oriented notation.

The calculation for effort can be generalized to give the ratios of effort between the two types of syntax for any number of operators. However, since it is well known that human's short term memory generally only contains 7 ± 2 "chunks" of information [13] and thus that most of the time users would specify no more than nine operations to be performed at one time, the ratios will only be calculated for 1 through 9 operations.

Case 1. Operand oriented syntax

M is the number of operations

$n_1 = M + 3 \qquad N_1 = M + 3$

$n_2 = 2 \qquad N_2 = 2$

$n = M + 5 \qquad N = M + 5$

$\text{Volume} = (M + 5) \log_2 (M + 5)$

$\text{Level} = 2/(M + 3)$

$\text{Difficulty} = (M + 3)/ 2$

$\text{Intelligence} = \frac{2}{(M+3)} (M+5) \log_2 (M+5)$

$\text{Effort} = \frac{(M+3)}{2} (M+5) \log_2 (M+5)$

Evaluating Effort for M = 1 to 9:

M	Effort
1	31
2	49
3	72
4	100
5	133
6	171
7	215
8	265
9	320

Case 2. Operator oriented syntax

M is tne number of operators

$n_1 = M + 3 \qquad N_1 = 4M$

$n_2 = 2 \qquad N_2 = 2M$

$n = M + 5 \qquad N = 6M$

$\text{Volume} = (M+5) \log_2 6M$

$\text{Level} = 2/M(M+3)$

$\text{Difficulty} = M(M+3)/2$

$\text{Intelligence} = \frac{2(M+5)}{M(M+3)} \log_2 6M$

$\text{Effort} = \frac{M(M+3) \; (M+5)}{2} \log_2 6M$

The evaluation of effort for M = 1 to 9 is:

M	Effort (operator syntax)
1	31
2	125
3	300
4	578
5	981
6	1535
7	2265
8	3195
9	4351

These measures give the following effort ratios between the operand syntax and the operator syntax:

M	Ratio
1	1.00
2	.39
3	.24
4	.17
5	.14
6	.11
7	.09
8	.08
9	.07

Thus, for a typical case of five operators, the operand oriented syntax requires only 14% of the effort to create when compared to today's operator oriented notation.

Conclusion

A computer organization is presented which utilizes independent micro-processors to implement high level operators needed for the execution of a computer program. The operation of the micro-processors is organized by a separate control computer and each individual operator micro-processor is triggered into a cycle of its operation by the definition of a new value of its argument(s). This organization can give a high degree of parallel operation, depending upon the rate of creation of argument values, the specific high level operators implemented, the speed of the micro-processors, and the size and type of memory available.

The system is extensible. Operators

can be added to the system after they have been given primitive status by the user. If desired, subprograms can be implemented by the micro-processors to operate in a functionally parallel mode.

Since the control computer is assumed to have a large resource of micro-processors, algorithms can be available in the library which allow an operator to use more than one micro-processor for its implementation, increasing the overall speed of the processor.

Finally, because of the simplicity of the operand oriented syntax, it is suggested that this computer organization is a useful one for a "dedicated" type of mini-computer or hand-held calculator for the general user who is more interested in getting answers about his data than in writing complex computer programs. With such a system, the user would only need to choose the operators to be applied to the data stream.

A statistical computer is an example of such a system that would interest the computer user community. Current technology could be used so that instead of passing a program of calculation steps through the calculator, the data would be passed through after the user has pushed his desired operator buttons. It appears that such a computer might receive acceptance from many computer users today.

List of References

1. Underwood, Benton J. "Attributes of Memory." Psychological Review 76 (June 1969).

2. Nielsen, J. M. Memory and Amnesia. San Lucas Press, 1958.

3. Tseytin, G. S. "Features of Natural Languages in Programming Languages." 4th International Congress for Logic, Methodology and Philosophy of Science. Bucharest, 1971.

4. Camp, H., Harvill, J.B., and Nylin, W. "REVEX, A Computer Design for Reversible Execution." Proceedings of the Fifth Annual Computer Science Conference of the Federation of North Texas Area Universities, Dallas, Texas. 1978.

5. Hockett, C. F. and Ascher, R. "The Human Revolution." Current Anthropology 5 (May 1964).

6. Brown, Roger. A First Language: The Early Stages. Cambridge, Mass.: Harvard University Press, 1974.

7. Harvill, J.B. "A Programming Language Model for an Operand State Saving Computer." PhD. Dissertation, Southern Methodist Univ., Dallas '78.

8. Quine, Willard Word and Object. The M.I.T. Press, Cambridge, Mass., 1960.

9. Anderson, G. and Jensen, E. "Computer Interconnection Structures: Taxonomy, Characteristics and Examples." Computing Surveys , Vol. 7, Number 4, Dec. 1975.

10. Flynn, Michael "Some Computer Organizations and Their Effectiveness." I.E.E.E. Trans. on Computers, Vol. C-21, No. 9, Sept. 1972.

11. Thurber, K. and Wald, L. "Associative and Parallel Processors." Computing Surveys, Vol. 7, No. 4, Dec. 1975.

12. Halstead, Maurice Elements of Software Science. New York, Elsevier, 1977.

13. Miller, G. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information." Psychological Review, 1956, 63.