A HIGHER-LEVEL LANGUAGE
FOR A LARGE PARALLEL ARRAY COMPUTER

by

Leonard Uhr

Computer Sciences Technical Report #354

May 1979

A Higher-Level Language for a Large Parallel Array Computer*

Leonard Uhr, University of Wisconsin

Abstract

This paper describes and discusses several pattern recognition programs that have been coded for and test-run on the CLIP parallel array computer. These programs were coded using a first version of a "higher-level language" whose compiler outputs object code for either CLIP3 (a 12 by 16 array of processors) or CLIP4 (a 96 by 96 array of processors). It was possible to code in a few hours simple programs with from 20 to 50 instructions that compiled into CLIP programs with from 200 to 1100 instructions.

Descriptors: Parallel Computers, Pattern Recognition
Scene Description, Parallel Arrays
Higher-Level Languages

Introduction

This paper A) examines large, general-purpose, programmable parallel array computers; B) describes a new "higher-level" programming language (PASCALPL**) for one such computer (CLIP, see Duff, 1976, 1978); and C) describes several examples of pattern recognition programs coded in that language. PASCALPL (Uhr, 1979) is a first version of a higher-level language for parallel processing. Its compiler, which is coded in Snobol/Spitbol and was run on University College London's IBM 360/65, output object code

* This research was partially supported by a UK NRC Senior Research Fellowship, US NSF grant MCS76-07333, and a University of Wisconsin Graduate School grant for research leave.

** PASCALPL might stand for:

P	PArallel PAttern
A	Array And
S	SCene
C	Cellular Co-occurrence
A	Analysis Array Automata
L	Logic
P	Programming
L	Language

(it was tested on about 8 different programs) that then assembled without any errors when input to the CLIP assembler (which inputs CLIP machine language code and outputs a binary object deck). The assembled programs then executed without error on CLIP3 (the 12 by 16 prototype CLIP system).

Because CLIP3 is too small an array to resolve more than one very simple object, it was not possible to examine extensively how well these programs ran. And they would need additional work to develop sufficiently powerful and properly adjusted sets of "transformations" - i.e., feature-detectors, compounding characterizers, and other types of indicants of what might be in the scene. But these programs were indeed able to recognize simple objects (e.g. A, B, square, circle, (very simple) tree, chair) input with a light pen on a scope face that discretized the patterns into 12 by 16 binary images. And stepping through the programs indicated they were doing this in reasonable ways.

Large programmable general-purpose arrays of many processors working in parallel (like CLIP4, with its 10,000 processors, and future similar but larger systems that it would now be feasible to build with 1,000,000 or even more processors) offer such great promise, especially for image processing, pattern recognition, scene analysis and description, and perceptual systems in general, that it seems important to develop the programming tools that will allow us to use them effectively, and to begin to demonstrate their usefulness.

A Brief Survey of Parallel Array Processors

Parallel array computers are designed with specific kinds of tasks in mind, where large arrays of information must be dealt with. The two types of task on which research has focussed are a) image processing and pattern recognition and b) the solution of large sets of equations. These have typically led to quite different kinds of hardware - probably chiefly because numerical problems must work with large numbers, at least 32 bits long, for reasonable precision, whereas image processing and pattern recognition rarely work with more than 4 or 8-bit values (e.g. for grey scale or weights), and often work with only binary values (for black-on-white transformations of the raw image). And image processing must contend with very large arrays, on the order of 500 by 500 or 1,000 by 1,000 "pixels" (picture elements).

We will therefore ignore the very expensive supercomputers (e.g. ILLIAC-4, CRAY-1) that have been developed for numerical problems (like weather forecasting and wind tunnel analyses). But it is important to point out that all of these computers are general-purpose. Because they are designed to handle specific classes of problems efficiently (as are all computers that are actually designed to compete in the market place), they are often dismissed as "special-purpose," in contrast to networks of processors connected in n-cube or other interconnection patterns. But the "more general" networks have not today been built to contain more than 10 or 20 processors, and plans for even a few hundred are still quite vague, and excessively expensive (on the order of $5,000 or more per processor). Even worse, the

problems of developing operating systems, programming languages and efficient programs for such networks are horrendous; and it seems likely that, say, 1,000 processors connected in a "general" network pattern would not increase throughput more than 5, 10 or 20 times (unless they were re-configured into an array whose structure more suitably mirrored the parallel structure of the processes being effected).

CLIP4 (which, designed and built by M.J.B. Duff (1976, 1978) and his associates at University College London, should be running some time in 1979) is the largest parallel computer in existence or near completion (or, so far as I am aware, even in serious design phase). Its large 96 by 96 array of (roughly) 10,000 hardware processors was possible only because a special chip was designed to contain 8 processors on each single chip. This has several important consequences: A) Once such a chip is checked out and produced in large quantities it will be relatively straightforward to build arrays still larger than 96 by 96. B) Chips are cheap (the CLIP4 chip, which is rather large and will be produced in relatively small quantities of a few thousand, costs about $28; if it were produced in quantities of hundreds of thousands or millions its cost would drop to only $5 or $2). C) Therefore this type of computer will be cheap (CLIP4 will be marketed for about $70,000). D) A 1,000 by 1,000 array could be built for $5,000,000 or less.

The only other large array of which the author is aware is the Distributed Array Processor (DAP) designed and built by Stuart Reddaway and his associates (see Flanders et

al., 1977), at England's ICL (Europe's largest computer manufacturer). DAP is presently running in a prototype 32 by 32 processor version, and the first full machine (a 64 by 64 array) is to be delivered to Queen Mary College (of the University of London) around July to November 1979. In contrast to CLIP4's price of $70,000, DAP (which was designed primarily for numerical problems but, because of several nice design features, including the large array size and connections between each processor and its near-neighbors, appears to be well suited for perceptual tasks) costs roughly $1,000,000 and (at least at present) can be bought only as an add-on to a large ICL computer that itself costs from $2,000,000 to $6,000,000. (The great difference in price results from a variety of factors, including much faster and more expensive technologies and a much larger memory for DAP, and also funding and marketing policies.)

CLIP and DAP are the only large arrays of parallel processors that are truely parallel at the hardware level of which this author is aware. Several small networks of 9, 16 or some similar small number of processors have been built in an architecture that allows them to scan serially over a larger array. These systems typically take from 1 to 10 microsecond per instruction, and need 5 or 10 instructions to do what CLIP can do in 1 instruction. So where CLIP can transform a 10,000 pixel array in roughly 11 microseconds (its basic instruction time), these systems need from 10 to 100 milliseconds.

Two very powerful systems have been designed and built, Bjorn Kruse's PICAP (1976, 1978) at the University of

Linkoping (Sweden) and Stanley Sternberg's Cytocomputer (1978) at the Environmental Research Institute (Ann Arbor, USA), to scan the array using only a single processor (or, in the latter system, a pipeline). These systems need only 1 microsecond per pixel to effect a rather powerful picture-processing instruction. Because of many additional nice design features (e.g. that allow for matching on inequalities in the former system and a pipeline in the latter system), one instruction can often do a good bit more than a basic CLIP-like instruction. This means that PICAP can handle a 64 by 64 (its scan size) in about 4 milliseconds and the Cytocomputer can handle a 1,000 by 1,000 (its largest, and preferred, scan size) in 700 milliseconds, or (to the extent that the pipeline can be used fully) down to 10 or 20 milliseconds.

When all its processors can be used effectively (and this is very frequently the case for image enhancement tasks, and also for the "lower" and some of the "higher" levels of scene analysis), CLIP4 can increase speeds by 4 orders of magnitude, due to its 10,000 processors, when compared to a single processor built from the same technology. Its crucial advantage is that it will continue to process increasingly larger arrays of information in exactly the same amount of time. In contrast, a scanner like PICAP or the Cytocomputer needs 100 times more time for 1,000,000 as opposed to 10,000 cells in the picture array.

To summarize: An array that is parallel at the hardware level (like CLIP or DAP) takes no more time to process the entire image than to process one pixel. If the

processor must scan the image, it needs its basic picture-processing instruction time multiplied by the number of pixels it must scan. By using 10 or 20 scanners we can cut time down by one order of magnitude. By specially designing a processor for pattern recognition purposes (which seems, essentially, to mean having it fetch small amounts of information in parallel from nearest neighbors, and execute appropriate logical operations) we can cut time by two orders of magnitude. But by using increasing thousands of processors we can cut time by increasing thousands. Here CLIP becomes increasingly attractive, since the larger the number of processors we use the cheaper and easier to build each single processor must be.

The Architecture of CLIP-Like Parallel Array Computers

A CLIP array is a 2-dimensional array of cells R rows by C columns in size. Each cell contains one processor and attendant memory and accumulators. Each processor in the array has a direct hardwired connection to each of its nearest neighbors (Figure 1). An instruction specifies 1) the subsets of its neighbors from which each processor will get information passed to it, and which subset of information in its own memory it should also use, 2) what operation it must perform on this set of information, and 3) where in its own memory it should store the results. Thus each Processor (P_i) has N Neighbors ($P_{i_1}, P_{i_2}, \ldots P_{i_n}$) and W Words ($P_{j_1}, P_{j_2}, \ldots P_{j_w}$) of memory in which it can store information (Figure 2). Each Processor can effect any Machine Language Process in the computer's repertoire (with the SIMD restric-

tion that all processors effect the same process at each moment).

In any actual hardware embodiment, CLIP must assign particular values to the above parameters. CLIP4's specifications are as follows (with CLIP3, the smaller machine that has actually been built and run, given in Figure 3).

CLIP4 is a 96 by 96 array of processors, each hardwired to its 8 nearest neighbors (with 6 neighbors, giving a hexagonal array, as a programmable option). One word of information (in each processor's memory) can be passed in one instruction cycle to these neighbors, and a second word can also be accessed. (Each processor has two accumulators (called the A-register and the B-register) into which it can put two words of its own memory information, and therefore also look at, and a D-register into which it puts its results, in addition to the up to 8 words of information passed into it by its neighbors (from their A-register). Note that one of the two words from the processor's own memory is the word passed to the neighbors.)

CLIP4 has 32 words of memory for each processor, and each word contains 1 bit of information. Its machine language operations are basically a logical "OR" over the up-to-ten bits of information, plus enough logical "AND" and "exclusive-or" operations combining information from neighbors with information from the processor's A and B registers to make any logical function codable (as a sequence of instructions) and, it is hoped to make this coding relatively convenient and efficient. (Here we have the same problems any designer of a real-world machine, one that must be effi-

cient and economical, must face. The constraints of packing 8 processors, including all their memories and interconnections, onto a single chip, to keep the total system down to a manageable size (but CLIP4 still has over 1,000 LSI chips) forced its designers to specify this relatively small memory, word size, and hard-wired instruction repertoire. Their success is best attested to by the fact that they are building 10,000 processors for about 35,000 pounds ($70,000) - the price for a mere handful, or even a couple, of more conventional processors.)

Programming CLIP in PASCALPL

PASCALPL is a first attempt to develop a "higher-level" language for CLIP. It gives the programmer a variety of features that will only be mentioned in this paper. It is directed toward pattern recognition and scene description, and tries to free the programmer from having to contend with the details of the machine's architecture and the machine language code. PASCALPL was developed because the author's attempts to code pattern recognition programs for CLIP (in contrast to the much shorter image processing programs that have typically been coded) were resulting in the need for long sequences of hundreds, or thousands, of machine language instructions to handle relatively mundane feature detection and compound characterizing tasks. That it served this purpose well is attested to by the fact that a 48 statement PASCALPL program that was coded in a few hours compiled to 1085 CLIP statements that would have taken several weeks to code and debug.

There are many variant possibilities for such a language, and it is hoped that this first relatively hurried and crude attempt will stimulate a variety of suggestions for, and attempts at, improvement.

The flavor of the language can best be got by examining the following basic types of PASCALPL statements:

A) Nearest-Neighbor-Masking statements effect processes that are functions of the (8 or 6) directly-connected nearest neighbors. (These are much in the spirit of CLIP's assembly language instructions, although they simplify and regularize them in a number of ways. But a simpler and more general format should be implemented here. The programmer should be able to write any logical functions he chooses, in a straightforward way, and the compiler should then map them onto the equivalent CLIP instruction or, when necessary, the required short sequence of CLIP instructions.)

B) Compounding statements effect processes that are functions of more distant neighbors. (The programmer can specify whatever number of such neighbors he chooses, from whatever distance.) Thus, e.g., several different features, such as the strokes, joins and ends of a letter, can be compounded together, to determine whether a higher-level feature is in the input.

C) Implication statements specify sets of possible output names and weights that should be associated with them if the feature that has just been searched for has indeed been found. (At present these weights are simply added into the sums of weights got so far, but it is expected that the

language will be augmented to give the programmer several alternative options. And he can always code still other options directly if he so chooses).

Several other types of statements complete the language:

D) A number of macro-like constructs have been added to the language, so that frequently used sequences of code can be specified very simply. The ones presently in the system were chosen by the author because they were convenient for the kind of pattern recognition programs he was coding, and it is not at all clear how generally useful they might be. But this type of facility seems valuable, both to build up a library of macros for a variety of users, and to allow users to specify and code their own macros (each is typically a straightforward set of Snobol code that generates the set of CLIP instructions the macro has been defined to call).

E) The inequalities have been regularized and expanded, so that the programmer has a wider variety at his disposal than those offered in CLIP3 or CAP4 assembly languages, in a somewhat more powerful, more general and, at least to the neophyte programmer, more understandable form.

F) Similarly, arithmetic has been extended, and put into a more generally usable and understandable form.

G) A number of control options have been given the programmer. These allow him to replace code that would otherwise be continually needed by a single control statement, from which the Snobol translator then generates and plants the needed code.

The following sections briefly describe the major features of PASCALPL. Then several of the programs coded in PASCALPL are informally described. The Appendix presents some examples of code, to illustrate how the various constructs can be used. (See Uhr, 1979, for a complete description of the language.)

The Compounding Statement

The compounding statement is coded in the general form:

Compound = $Part_{1(mask)}OP_1Part_2(mask_{22_2})OP_2$

e.g.; TRIANGLE = $SLOPE_{1(3,4)}*SLOPE_2(1,4)*SLOPE_3(6,4)$

Here $SLOPE_1$, $SLOPE_2$, and $SLOPE_3$ each refer to the particular word in each processor's memory that contains information about whether that particular slope is present at that processor's location. (This is information that would have been got by previous nearest-neighbor-masking and compounding instructions).

The star (*) is the logical "AND" (other operations are + for "OR", @ for "EXCLUSIVE-OR" and - for "NOT").

The two numbers in the nearest-neighbor-mask (separated by a comma) indicate a) the direction in which the information is to be passed (Figure 1 shows the directions used by CLIP and PASCALPL), and b) the distance. Thus, in the example above $SLOPE_1$ is passed 4 steps from the Northwest to the Southeast.

The Nearest-Neighbor-Masking Statement

The nearest-neighbor-masking statement is the basic CLIP statement, and can be quite complex. It fetches information from one or two memory locations, passes one of these

pieces of information to the 8 or fewer neighbors specified in the instruction, computes a boolean function of all this information, and stores the result in the specified memory location. It can have a number of forms, but it will typically be of the following sort:

Output = Info,((mask)(op)($Info_2$))

e.g.: IMAGE = IMAGE(2468)*-IMAGE

(equivalent CLIP3 code is:

```
LD 1 C 1      (LoaD 1 into A-register, Clear B, store result in 1)
PR 0(2468)A,-A*P, S    (PRocess A thru directions 2468,
                              call it P, "AND" with A, use Square array)
```

(the programmer must assign a D-level - in this case 1 - to IMAGE.))

Here IMAGE refers to the word in each processor's memory. IMAGE is passed on (by the nearest-neighbor-mask) to the 4 horizontal and vertical nearest neighbors (e.g. the 2 passes it from North to South). A logical "OR" (which need not be specified within the nearest-neighbor-mask) is effected over all these 4 passed-on pieces of information about IMAGE. It is then logically and-ed with the negation of the information (from the cell itself only).

A number of variations on such a command are possible. For example, the output might be stored in a third memory word, e.g.: SLOPE/dl/u = IMAGE(15)+ $FEATURE_1$ = IMAGE)15)+$FEATURE_1$)

NEWIMAGE = IMAGE*NEGATIVE (here no information is passed to neighbors; rather, a simple logical function is computed over two pieces of information fetched from the (each) cell's own memory.)

Several more complex variants will only be mentioned here. These include single and double negations within a single instruction, and instructions that continue to pass information, via "recursive propagation" over longer distances. Examples of actual CLIP code are given below.

The Implication Statement

Whenever a feature, compound, characteristic (or any other type of information) implies possible alternatives an Implication statement of the following sort can be written:

Feature == $Implied_1(Wt_1)Implied_2(Wt_2)...Implied_n(Wt_n)$

e.g.: VERTICAL = POLE(5)TREE(4)LETTER-D(3)CHAIR(2)LETTER-I(4)

The weight (in parentheses) associated with each of the names is added into the total weight that the program is accumulating for that object. (In CLIP3 up to 16 such objects weights can be handled; in CLIP4 up to 96 objects weights will be handled.)

Several legal variants on this statement, that make for more efficient programming and simpler code, will be described later.

Brief Descriptions of Some Programs Coded in PASCALPL

The programs that have been coded in PASCALPL to date were all designed to recognize and describe objects input to the array, rather than simply to enhance or preprocess the image. They all followed a general procedure of getting simple, local features, then compounding these, and continuing to compound them, to give successively higher-level and more global features, characterizing compounds, parts of objects, and objects.

These programs tended to follow the parallel, layered, converging "cone"-"pyramid" discipline that the author (Uhr, 1972, 1976, 1978; Uhr and Douglass, in press), and a number of other researchers (e.g. Hanson and Riseman, 1974; Klinger and Dyer, 1974; Levine, 1978; Tanimoto, 1976, 1978) have been programming on serial computers. This led to the need for large numbers of compounding operations, and even larger numbers of shifting operations to move the several features into position as nearest-neighbors so that they could then be compounded together.

Therefore the programs take on an unusually simple structure. They begin with nearest-neighbor masking operations that effect the simple low-level pre-processing and local edge-detection functions. (Many of these were replaced by simple macros, e.g. to #EDGE#, i.e., get 4 local edges.) Then a sequence of compounding statements shifts and compounds lower-level features and characteristics into successively higher-level ones. Each of these implies one or more possible names, each with its associated weight. If the feature is found, anywhere, these weights are combined into the total weights for these names. Finally, the program chooses the most highly implied name. (If the array were large enough, as it will be when CLIP4 is used, the program would also have to find the region where that named object was implied, eliminate the object, and continue on to find other highly-implied objects in other regions of the scene.)

This is only one approach to scene analysis. But it is an approach that is highly parallel, and can take advan-

tage of the potential great increases in speed offered by a parallel computer like CLIP. PASCALPL was designed to expedite programs that take this approach, especially in its major Compounding and Implication statements. But it is a general-purpose language for a general-purpose computer, so any kind of scene analysis or pattern recognition program could be coded in it. On the other hand, there is little point in using a parallel computer for a serial program. The important questions are: what is the range of parallel programs that people would like to code; how commonly can serial programs be replaced by equivalent parallel programs; and what kind(s) of language(s) would best serve the kinds of programs people would like to code?

The author found it convenient to set up a code sheet in the form of a transition matrix whose column labels were the possible names of objects and whose rows were labelled with the features and characteristics to be looked for. First, one would write a feature-name (e.g. "vert" or "L-angle" or "window") as the row label, or simply draw the feature. Then the weights with which this feature implied each possible object would be put in the object's column for this feature-row. Finally, the actual PASCALPL code, for near-neighbor-masking or compounding, would be written for that row. The code for Impication statements would already be present, in the cells of the array!

This turned out to be extremely simple for the author to use, and reasonably easy to teach and get somebody else to use. (Only one other person beside the author was asked to code a program in PASCALPL. He spent about two

hours learning the language and one hour coding a program, one with about 20 statements that compiled into over 200 CLIP statements, and executed without bugs.) It is probably too narrow and constraining a discipline, but it seemed to be a simple way to start. And it is not as constraining as it might appear, since a very wide variety of programs can be coded in this way. In any case, PASCALPL offers many additional constructs, and can be used in a very large number of different ways.

About 8 programs were coded in PASCALPL, ranging from roughly 15 to roughly 50 statements in length, but generating CLIP3 object code ranging from roughly 200 to 1100 statements in length. These programs attempted to recognize from among 4 up to 15 different objects. They used roughly 10, 20, or 30 different features and characterizers. Shifting characterizers into near-neighbor position and merging weights into position and adding them took the bulk of the (object code) instructions and computing time. Finding the object-name with the highest weight took strikingly little time, since it could take advantage of parallel processing, as follows: A bar of 1s was shifted down the columns that stored the combined weights, and and-ed at each shift. This gives a 1 in the column(s) with the highest non-zero bit(s). The procedure continues recursively if more than one 1 is found. This is a small example of the power of a parallel computer such as CLIP. Far more important is the parallel computer's ability to find a feature or characteristic everywhere, simultaneously.

APPENDIX: Examples from CLIP Programs Generated from PASCALPL Code

The following presents some actual PASCALPL program-segments, and the CLIP code into which the Snobol compiler translated them.

Figure 4 shows the start of the program as it resides in CLIP, with the statements shown as comments (starting with ";") numbered 1 to 54 showing the input source code in PASCALPL. PASCALPL-generated comments also indicate that the control modes specified at the start of the program commanded that CLIP3 code for a SQUARE (i.e., 8-neighbor) array be generated, with the integers 0, 1, ... being used by the programmer to directly specify the D-levels.

The program outputs that the programmer must remember to use T and F for True (1) and False (0) rather than 1 and 0, which might be confused with any 1 and 0 specified for the D-levels (this is not necessary if the programmer names the D-levels with alphanumeric letter-strings.)

Figure 5 shows the start of the program, up to output statement 56 (numbered at the right), which was generated by input statements 1 through 10 and part-way through 11 (numbered at the left in Figure 4, and repeated, as comments, each starting with "IN= ", just before the output code generated in Figure 5). (This listing is the output of the CLIP assembler, which numbers the code, but not the comments, in octal, starting with location 200).

The first instruction, #INSCOPE#, generates CLIP3 output code numbered 1 through 5 to establish the loop needed to input with a light pen, plus 6-7 to set D-level 15 to

contain all 0s (to initialize weights, which will be accumulated in D15).

The second input instruction (which comes immediately after CLIP3 instruction 7 and has the comment "CONTOUR" at its far right) generates CLIP3 statements 8 and 9. (D-level 0 is set to equal 1 if the center cell contains a 1 and any of the eight neighbor cells contains a 0). Next come statements getting the 4 local straight edges, each followed by the pair of CLIP3 statements it translates into. (Note that the system's printers print an underline or left-arrow instead of the circumflex used to indicate the center cell). Without teaching CLIP3 programming (which can be got from CLIP internal technical reports at University College London), I should mention that whereas PASCALPL gets the "AND" of the center cell and its two neighbors, thus forming the straight line, CLIP3 code must negate the two neighbors, "OR" the negation, negate again, and "AND" the result with the center cell. Thus PASCALPL allows for much simpler and more straightforward logical operations (which it then converts to the actual operations needed at the machine hardware level, where only the "OR" but not the "AND" of the neighbor cells is wired into the hardware).

Next comes the macro ERASE 1,2,3 and 4. Macros could have been used to effect #CONTOUR# and #EDGES# (for the 4 edges) and they would have generated exactly the same code. It seems obvious that such macros are easier to learn and use. But they may well be too easy, allowing a programmer to code with only the vaguest idea of what his program is doing. And they may well be too specific to my personal

way of doing things (though I have found them quite useful and convenient), and lead the programmer into a narrow set of practices.

After statement 39 comes a new type of implication statement. Here the 1 to the left of the equal sign specifies the D-level to be tested. If it contains at least one instance of the feature it stores (that is, if not all of its cells are 0) then the weights specified are added into the weights of the names 1,2,..., the first integer added to the first name (1), the second integer to the second integer to the second name (2), etc. The hash mark ("#") indicates 'chop the ordered set of integers that follow'. The letters A through E indicate negative weights, the dots indicate no weight (i.e. zero weights, so nothing need be added or subtracted for that name.) The code that follows gets and tests the specified D-level and, if it contains the feature, sets up the sub-routine calls to be made if the feature is found. (Note how 5,7 and 10 (octal) are ignored, and how 6,11 and 13 are given negative weights to be added in to the total weights.)

References

Duff, M. J. B., CLIP4: a large scale integrated circuit array parallel processor, Proc. IJCPR-3, 1976, 4, 728-733.

Duff, M. J. B., Review of the CLIP image processing system, Proc. National Computer Conf., 1978, pp. 1055-1060.

Flanders, P. M., Hunt, D. J., Reddaway, S. F. and Parkinson, D., Efficient high speed computing with the distributed array processor. In: High Speed Computer and Algorithm Organization, New York: Academic Press, 1977, pp. 113-128.

Kruse, B. The PICAP picture processing laboratory, Proc. IJCPR-3, 1976, 4, 875-881.

Kruse, B. Experience with a picture processor in pattern recognition processing, Proc. National Computer Conference, 1978.

Hanson, A. R. and Riseman, E. M., Pre-processing cones: a computational structure for scene analysis, Tech. Rept. 74C-7, Univ. of Mass., 1974.

Klinger, A. and Dyer, C. Experiments on picture representation using regular decomposition, TR Eng. 7497, UCLA, 1974.

Levine, M. D. A Knowledge-Based Computer Vision System, In: Computer Vision Systems, A. R. Hansen and E. M. Riseman (eds.) New York: Academic Press, 1978, pp. 335-352.

Sternberg, S. R. Cytocomputer real-time pattern recognition, Paper presented at: Eighth Annual Automatic Imagery Pattern Recognition Symposium, National Bureau of Standards, Gaithersburg, Md., April 3-4, 1978.

Tanimoto, S. L. Pictorial feature distortion in a pyramid, Comp. Graphics Image Proc., 1976, 5, 333-352.

Tanimoto, S. L. Regular Hierarchical Image and Processing Structures in Machine Vision, In: Computer Vision Systems, A. R. Hansen and E. M. Riseman (eds.), New York: Academic Press, 1978, 165-174.

Uhr, L. Layered "recognition cone" networks that preprocess, classify and describe. IEEE Trans. Computers, 1972, 21, 758-768.

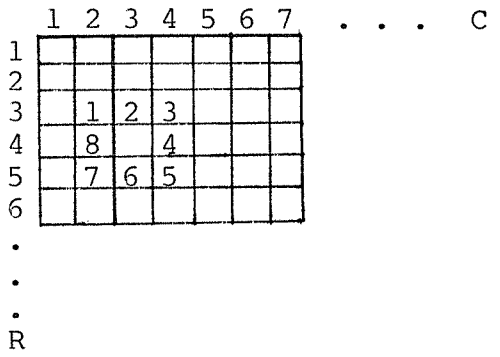
Uhr, L. "Recognition cones" that perceive and describe scenes that move and change over time, Proc. IJCPR-3, 1976, 287-293.

Uhr, L. "Recognition Cones," and Some Test Results; The Imminent Arrival of Well-Structured Parallel-Serial Computers; Positions, and Positions on Positions. In Computer Vision Systems, A. R. Hansen and E. M. Riseman (eds.), New York: Academic Press, 1978, 363-378.

Uhr, L. PASCALPL: A language for programming scene description and pattern recognition systems on a parallel array computer. Univ. of Wisconsin Computer Sciences Dept. Tech. Report, 1979.

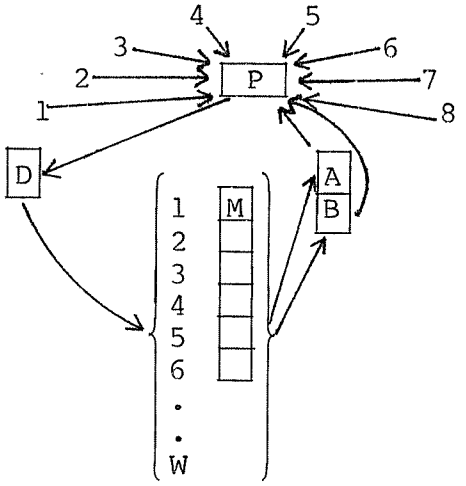
Uhr, L. and Douglass, R. A parallel-serial recognition cone system for perception: some test results, Pattern Recognition , in press.

Figure 1. An Overview of CLIP Architecture



A 6 row by 7 column sub-array of a CLIP array of R rows by C columns. Each cell contains one processor and its associated memory words and accumulators. In square mode (as shown here, and embodied in CLIP3 and CLIP4) the processor is connected by direct wires to its 8 nearest neighbors. Which of these neighbors passes information to the processor is determined by the process instruction being executed. Neighbors are designated by integers from 1 to 8 (as shown for cell 4-3 above).

Figure 2. The Basic CLIP Process Instruction



A single CLIP cell. The Processor (P) computes a boolean function whose arguments can include any 2 of the W words of information in its Memory that are fetched into its A-register and B-register, plus the 8 words of information that are simultaneously fetched into the A-registers of its 8 nearest neighbors. The results are placed in the D-register, and then can be stored in the word of memory designated by the instruction.

Figure 3. Specifications of CLIP3, CLIP4, and Possible Future CLIPs

System:	CLIP3	CLIP4	CLIP?A	CLIP?B
Overall Specs:				
Array Size	12x16	96x96	500x20	1000x1000
Number of Processors	192	9216	10,000	1,000,000
Each Processor:				
Number of Memory Words (D-Levels)	16	32	256	512
Size of Memory Words	1 bit	1 bit	1 bit	4 bits
Number of Neighbors	8 or 6	8 or 6	8 or 6	8 or 6

The two conjectured future CLIP arrays are merely illustrations of how CLIP might be expanded.

Figure 4. The Beginning of an Example PASCALPL Program

```
;!CTL MODES= CLIP3, SQUARE,
;!CTL INAMES= 0
;  2; PASCALPL PROGRAM FOR 15 OBJECT-NAMES
;  3;  OBJECTS ARE =#  ABCOUCTTPFSTPDF
;!CTL DNAMES= 0
;T AND F MUST BE USED FOR 0,1 IN BLN EXPR
;DON'T PUT INTEGER NAME OF B IN MASK EXCEPT AFTER '+'
;4     START #INSCOPE# ;
;5       0 = -0(1-8)*0    ; ; CONTOUR
;6       1 = 0(*15←)      ; ; 4 EDGES
;7       2 = 0(*26←) ;
;8       3 = 0(*37←) ;
;9       4 = 0(*48←) ;
;10        #ERASE#1,2,3,4, ;
;11         <1=#     1111.A..A.A2A21 ;
;12     L11      <2=#     12...222222.2A1 ;
;13     L12      <3=#     ..111.11AAA2A21 ;
;  14;    4 SQUARE ANGLES
;15     L13      <4=#     21...2.2B2222A1 ;
;16     SQANGLS 5 = + 4(#12,2)*2(#54,2) ;
;17           =#     1.BAB112C22.2B ;
;18     L27    6 = + 2(#34,2)*4(#76,2) ;
;19           =#      1.BA....C22.2B ;
;20     L39    7 = + 2(#12,2)*4(#56,2) ;
;21           =#      1....2..C22.2B ;
;22     L51    10 = + 2(#78,2)*4(#32,2) ;
;  23;   SQUARE-EDGED COMPOUNDS
;24           =#      1...B212C22.2B ;
;25     L64     11 = + 5(#1,3)*7(#7,3)    ; ; SQUARE
;26           =#      ..B.A..1B23.B.. ;
;27     L80     11 = + 6(#5,3)*10(#1,3)    ; ; SQUARE
;28           =#      ..B.A..1B23.B.. ;
;29     L96     11 = + 3(#1,3)*1(#3,3)    ; ; ← (INVERTED V)
;30           =#      2.A.B.21B..3.3. ;
;31     L112    12 = + 1(#7,3)*3(#5,3)    ; ;  V
;32           =#      A.A.1..1B....3. ;
;33     L128    12 = + 12(6,4)*11(#2,4)    ; ; DIAMOND
;  34;  3 LONG EDGES
;35           =#      ..B1..11BBB.B5. ;
;36     L148    12 = 1(1,8)*1(0) ;
;37           =#      2.AA.......1.1A ;
;38     L160    13 = 2(2,8)*2(0) ;
;39           =#      12AA.23.332B2BA ;
;40     L172    14 = 3(3,8)*3(0) ;
;41           =#      2.AA...1...1.1A ;
;42     L184    11 = + 11(#2,6)*12(3#4,4)*14(#81,4)*4(6,4) ;
;43           =  1(5)    ; ; IMPLIES A-LETTER STRONGLY
;44     L231    11 = + 13(8,3)*5(#43,3)   ; ;FLAG-PART
;45           =#      11..B1...3.B.B. ;
;46     L246    11 = + 2(8,2)*2(4,2)    ; ; PARALLEL VERTS
;47           =#      1.A.13.2A11.A.. ;
;48     L256    11 = + 1(84,2)+3(84,2)*13(84,1)   ; ;POLE+BRANCHES
;  49;    CURVES - LOCAL + SHIFTED
;50           =#     1.B...5.B..... ;
;51     L270    11 = 0(*1-4)*-0 ;
;52       12 = + 1(#3,2)*4(#23,2) ;
;53       11 = 11+12 ;
;54         =#     .2.2.1.1B ;
```

Figure 5. Continuation of the Example PASCALPL Program

```
        ;CLIP PROGRAM STARTS HERE.
        ;IN=  SHOWS INPUT SOURCE CODE AND COMMENTS
        ;SUBROUTINES NEEDED AND ADDED ARE -
        ;     UPWT,DNWT,MAX.
        ;####IN=  ;
        ;####IN=  ;; PASCALPL PROGRAM FOR 15 OBJECT-NAMES
        ;####IN=  ;;  OBJECTS ARE =#  ABCOUCTTPFSTPDF
        ;####IN= START #INSCOPE# ;
0200    START:    LD  0D, 0D, 0, S      ;  1
0201       PR A       ;  2
0202          HI  50       ;  3
0203          EG  0       ;  4
0204          BR  0,  4,  START      ;  5
0205     LD C, C, 15       ;  6
0206     PR A       ;  7
        ;####IN=   0 = -0(1-8)*0   ;; CONTOUR
0207           LD    0, C, 0   ;  8
0210        PR  0(1-8)-A, P*A,  S  ;  9
        ;####IN=   1 = 0(*15<)      ;; 4 EDGES
0211           LD    0, C, 1   ;  10
0212        PR  0(15)-A, -P*A,  S  ;  11
        ;####IN=   2 = 0(*26<) ;
0213           LD    0, C, 2   ;  12
0214        PR  0(26)-A, -P*A,  S  ;  13
        ;####IN=   3 = 0(*37<) ;
0215           LD    0, C, 3   ;  14
0216        PR  0(37)-A, -P*A,  S  ;  15
        ;####IN=   4 = 0(*48<) ;
0217           LD    0, C, 4   ;  16
0220        PR  0(48)-A, -P*A,  S  ;  17
        ;####IN=    #ERASE#1,2,3,4, ;
0221                    LD  C,  C,  16      ;  18
0222         PR  1       ;  19
0223           LD  16,  C,  16      ;  20
0224           PR  0(2)A,P*A, S      ;  21
0225           LD  16,  C,  16      ;  23
0226           PR  0(4)A,P*A, S      ;  24
0227           LD  16,  C,  16      ;  26
0230           PR  0(6)A,P*A, S      ;  27
0231           LD  16,  C,  16      ;  29
0232           PR  0(8)A,P*A, S      ;  30
0233        LD 1,  16,  1       ;  32
0234          PR  A*P       ;  33
0235        LD 2,  16,  2       ;  34
0236          PR  A*P       ;  35
0237        LD 3,  16,  3       ;  36
0240          PR  A*P       ;  37
0241        LD 4,  16,  4       ;  38
0242          PR  A*P       ;  39
0243        LD 0,  16,  0       ;  40
0244          PR  A*P       ;  41
        ;####IN=     <1=#    1111.A..A.A2A21 ;
0245                   LD  1,  C,  1      ;  42
0246         PR  A       ;  43
0247       PR # -A       ;  44
0250       BR   1, P, L11       ;  45
0251        LR    1 (1)       ;  46
0252        LR    1 (2)       ;  47
0253       BS    UPWT       ;  48
0254        LR    2 (1)       ;  49
0255        LR    1 (2)       ;  50
0256       BS    UPWT       ;  51
0257        LR    3 (1)       ;  52
0260        LR    1 (2)       ;  53
0261       BS    UPWT       ;  54
0262        LR    4 (1)       ;  55
0263        LR    1 (2)       ;  56
```