JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 8, 231–244 (1990)

HCL: A Language for Low-Level Image Analysis

JOSEPH J. PFEIFFER, JR.

Department of Computer Science, New Mexico State University, Las Cruces, New Mexico 88003-0001

Pyramids have received a great deal of interest as a data structure for use in hierarchical computer vision. This paper describes a language for the description of pyramid algorithms, supporting conceptually parallel pattern-matching and arithmetic operations. The language is defined by adding constructs from a notation for pyramid algorithms called Hierarchical Cellular Logic to the general-purpose programming language C. Initially developed to support vision research on PCLIP II, a cellular processor with a pyramidal interconnection topology, the language is also implementable on a wide range of parallel architectures. © 1990 Academic Press, Inc.

1. INTRODUCTION

Pyramids and cellular processors are separate developments which have both been used with great success in computer vision over the past decade. Pyramids are a very useful data structure for hierarchical vision and other multigrid algorithms, while massively parallel cellular processors are useful for high-speed computations involving local neighborhood processing. A number of current projects are attempting to construct cellular processors using pyramidal topologies, in order that the power of the pyramid data structure can be combined with the speed of the cellular processor.

A difficulty which has existed to date has been the lack of a language suitable for developing algorithms for these pyramidal cellular processors. A pyramid algorithm normally is expressed as a number of procedure and function calls to a library performing pyramid operations. As this results in using procedure call syntax to express operations on the fundamental data type of the algorithm, a great deal of expressive power is lost. This is exactly the issue which has led to the development of languages such as FORTRAN 8X supporting matrix operations. A second possible approach, the encoding of algorithms in the assembly language of a cellular processor (an approach which has also been tried), suffers from exactly the same faults as developing serial algorithms in assembly language and is equally unworkable.

The need, then, is to develop a notation in which to express and develop pyramid algorithms. This need was met in part by Tanimoto's Hierarchical Cellular Logic (HCL), a notation used to describe pyramidal pattern-matching operations [14]. The elementary operations in this notation are Boolean operations between nodes in pyramids and pattern-matching operations on local neighborhoods of nodes in a pyramid. Unfortunately, this is more of a mathematical notation than a programming language, so developing algorithms using it is difficult.

This paper describes a pyramid programming language based on Hierarchical Cellular Logic and the programming language C. The basic approach is to add pattern-matching operations from Tanimoto's notation to C and to overload C operators to provide the remaining functionality. The language has been used in developing a number of computer vision algorithms and has shown itself to be well suited for this purpose.

The paper is divided into seven sections. Following this introduction, Section 2 provides some background for HCL. This includes existing parallel languages, cellular processors, pyramids, the PCLIP II pyramidal cellular logic processor, and Hierarchical Cellular Logic. Section 3 describes the goals of the HCL project and details some of the design decisions which led to the language as it exists today. Section 4 defines the extensions to C which result in HCL, including language features and user interactions. Section 5 contains two examples of HCL programs, showing the use of the language in practical problems. Section 6 has information regarding the implementation of the language and the impact of this implementation on portability. Finally, Section 7 contains conclusions and presents some thoughts regarding the development of languages supporting cellular processors with arbitrary topologies.

2. BACKGROUND

HCL has been developed to meet a particular need, growing out of the current state of hierarchical computer vision and parallel cellular processors. This section provides some of this background for the language.

2.1. Cellular Processors

Computer image processing is a very challenging area of research, not least because of its demand for high-speed

0743-7315/90 $3.00

computing. The algorithms used for low-level vision, such as convolutions, have also proved to be easily programmed for parallel computers, providing an avenue of research into special-purpose computer hardware.

A particular class of parallel computers which has proved useful for low-level computer vision is that of cellular processors. This family of architectures makes use of a large processor grid, with one processing element (PE) assigned to each pixel in the image array performing operations in parallel throughout the array.

The PEs used in the grid are typically one-bit processors, with some amount of local memory and a pattern-matching or data selection unit, as shown in Fig. 1. Each PE has access to communication registers in the PEs which are immediately adjacent to it (these are its neighbors). A pattern is matched in the neighbors (or the contents of one neighbor may be selected), and the result is used as one input to a Boolean processor. The other input comes from the local memory of the PE. The result from the Boolean processor is returned to the local memory and to the PE's communications register, where it is available to adjoining processors for the next instruction cycle. The processor is capable of performing a number of operations and is oriented around bit-serial arithmetic operations. A single control unit executes a program, dispatching instructions to all of the PEs in the array, which execute them in parallel.

Some of the cellular processors which have been proposed or constructed are CLIP4 [8], MPP [1], PAP [16], and the Connection Machine [3].

2.2. The Pyramid Data Structure

A pyramid is an image data structure which combines features of arrays and quadtrees, providing a representation of a resolution hierarchy which also yields computational improvements in cellular processors [15].

In a pyramid, an image is represented by a series of N arrays, each of them one-fourth the size of the array directly below it in the pyramid. Consequently, Level 0 of the pyramid (the top level) is a single node, Level 1 is a 2×2 array, Level 2 is a 4×4 array, and in general, Level n is a $2^n \times 2^n$ array. In an eight-level pyramid, the base is a 128×128 array at Level 7. The connections between nodes at various levels of the pyramid are provided by a quadtree. This is a structure analogous to the binary tree, but with each node having four children instead of just two. A four-level pyramid, with only a small part of the quadtree shown for the sake of legibility, is shown in Fig. 2.

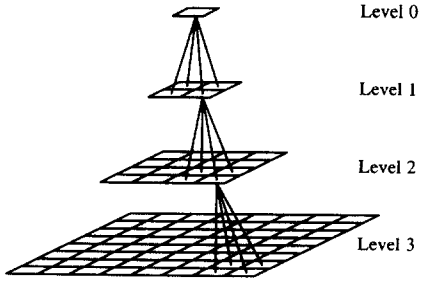

FIG. 2. Pyramid data structure.

The local neighborhood of a node in a pyramid is shown in Fig. 3. The node labeled "Center" has a total of 13 neighbors, including a parent (in the level above the node), eight siblings (at the same level as the node), and four children (at the level below the node). Exceptions to this rule exist at Level 0, which serves as the root of the pyramid and has no parent; at Level $N - 1$, which contains the leaves and has no children; and around the perimeters of the levels, where the sibling sets are incomplete. The neighbors are all labeled as shown in Fig. 3; data are shifted between nodes in the pyramid along these neighbor links.

An operation is performed in a pyramid by considering the neighbors of a node as operands, calculating a result based on these operands, and placing the result at the node. Typically, the operation will be performed in parallel on a large subset of the nodes in the pyramid. So, one might estimate a derivative by subtracting the South neighbor of each node in a pyramid from its North neighbor and assigning the result to the node.

Ordinarily, a pyramid is built by operating on an image at the base level. The process is iterative, with the construction of the pyramid moving up from the base toward the

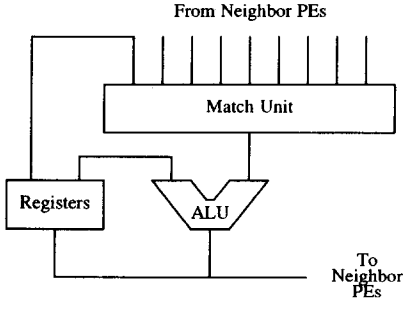

FIG. 1. Generic SIMD computer processing element.

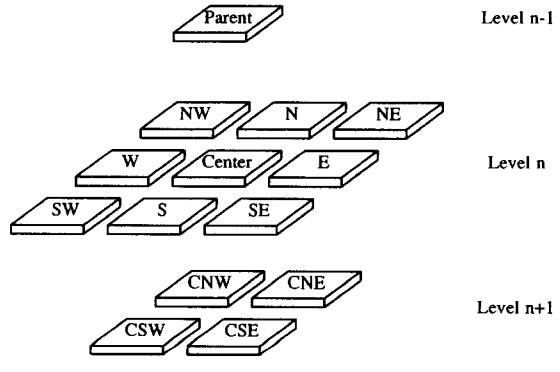

FIG. 3. Local neighborhood of a node in a pyramid.

root. At each step in the iteration, a function is applied to the nodes at the current top level of the constructed pyramid to produce the values at the next level up. Some examples of pyramid construction techniques are averaging (used to produce lower-resolution images for searching or to eliminate noise) and locating a maximum pixel value (useful for hot-spot detection). In all these cases, each node at the level being created examines its children to obtain a value. In some sense, a given level of a pyramid holds a coarser representation of the data at lower levels.

The advantage of the pyramid over a single array can be seen in considering the time required for an operation such as hot-spot detection in the two topologies, assuming an appropriate parallel architecture is available in each case. For an array processor (such as MPP), the number of steps required is linear in the width of the array; for a pyramid processor, the number of steps is logarithmic. This improvement is typical for a broad range of pyramidal algorithms. The distance in the graph of an array of size $n \times n$ from one corner to a diagonally opposite corner is $n - 1$, while in a pyramid, this graph distance is reduced to $2(\log_2 n - 1)$. This implies that for algorithms involving global measurements within an image, computational efficiency can be enhanced by making use of the pyramid [13].

2.3. The PCLIP II Cellular Logic Processor

A relatively recent development in research in cellular processors has been the consideration of topologies other than grids and toruses. So, for example, the Connection Machine uses a hypercube for its interconnection. Another possibility, with direct applicability to hierarchical computer vision and other multigrid algorithms, is to use a pyramid as the interconnection topology for a cellular processor.

PCLIP II (*P*yramid *C*ellular *L*ogic *I*mage *P*rocessor) is such a processor, currently under development at New Mexico State University. It is a descendant of the earlier PCLIP processor, developed at the University of Washington [17]. It uses a complete neighborhood for each PE and has eight levels, with a 128×128 base level. Following the pyramid computation model described above, there is a processor located at each node; operations are performed in parallel by these processors on elements of their local neighborhoods to obtain new values. Input and output of images occur on the base level of the pyramid, while output of global results is possible at the root.

Algorithm studies using PCLIP II, and its predecessor PCLIP, have led to a number of useful results, but they have also pointed out the need for a programming language other than the PCLIP II assembler for writing pyramid programs. The development of HCL is the result of this attempt to build a general-purpose pyramid programming language.

The original motivation for HCL was simply as a language for PCLIP; once HCL was developed, however, the PCLIP instruction set was modified to provide the set of pyramid operations required for the language. There has been a continuing series of modifications and optimizations, with algorithm implementations resulting in changes to HCL, HCL requirements dictating the PCLIP instruction set, and the instruction set modifying the microarchitecture. The refinements have been extensive enough that the current design is considered a second-generation pyramid processor, and hence is now called PCLIP II.

2.4. Hierarchical Cellular Logic

Tanimoto's Hierarchical Cellular Logic is a notation permitting the description of pyramidal operations in an architecture-independent manner by providing abstract definitions of pattern matching operations. The fundamental object in Hierarchical Cellular Logic is a binary pyramid (a pyramid whose value at each node is either a 0 (false) or a 1 (true)). Two operators are defined, called AND_Match and OR_Match. These operators perform pattern matches, comparing a pattern against the local neighborhoods of all of the nodes in a pyramid, in parallel. Their results are pyramids, with the value true at exactly those nodes in the input pyramid at which the pattern match was successful.

The operators take two arguments, consisting of a pattern and a pyramid. A pattern is a list of 14 elements,

$$\langle P\ PPP\ PPP\ PPP\ PPPP \rangle.$$

Each of the pattern elements, P, is either a `0`, a `1`, or an `x`. The pattern elements refer to the 14 neighbors of a node in the pyramid (refer to Fig. 3), including the node itself. The order of pattern elements is Parent, NorthEast, North, NorthWest, West, Center, East, SouthWest, South, SouthEast, NorthWestChild, NorthEastChild, SouthWestChild, and SouthEastChild. In matching a pattern element against a neighbor of a pyramid node, a `0` element must be matched against a 0 neighbor, and a `1` element against a 1 neighbor. An `x` element is a "don't care"; it is ignored in the pattern-matching operations to follow.

The syntax of the two pattern-matching operators is

AND_Match(*Pattern*)*Pyramid*

and

OR_Match(*Pattern*)*Pyramid*

In an AND_Match, every pattern element which has a value other than "don't care" must match its corresponding neighbor for a pattern match to be successful at a node. The node's parent must match the parent pattern element; its NorthWest neighbor must match the NorthWest pattern element; and so forth. In an OR_Match, at least one of the pattern elements must match.

Boolean and arithmetic operators are defined in HCL to be element-wise; that is, adding two pyramids together adds their corresponding nodes. The arithmetic operations are built from a sequence of Boolean operations. The normal infix notation is used here. Constants may be used freely; the interpretation is that a constant used in a pyramidal expression refers to a pyramid with the same value at each node. A set of predefined pyramids which will be of interest in a moment is the family of pyramids Q_L. Each of these refers to a pyramid which contains all 1's at Level L and is completely filled with 0's elsewhere in the pyramid.

Repeated applications of an operation, and its transitive closure, are permitted. The usual superscript and star notations are used here.

Finally, Tanimoto permits operations to be restricted to subsets of the nodes in the pyramid. This is not a primitive operation but rather is defined by

$$[Op \mid Subset](pyramid) \equiv (Op(pyramid) \cap Subset) \cup (pyramid \cap \overline{Subset}).$$

The effect of this is to apply the operation to the set of nodes defined by the subset and to retain the previous values for the remaining nodes.

We can see an example of the use of the notation by considering the creation of an AND-pyramid. This is a binary pyramid with an input image at its base level, segmented into foreground (areas of interest, represented by true nodes) and background (the remainder of the image, represented by false nodes). At each level above the base, a node is in the foreground if and only if all four of its children are also foreground. The purpose is to establish paths between the nodes of a region going up through higher levels of the pyramid, but without establishing paths between regions. Effectively, this extends the foreground regions from the base plane up into the pyramid, producing pyramidal regions exhibiting the same connectivity as the original image (that is, no regions which are disconnected at the base can be connected at a higher level by the construction of an AND-pyramid). The advantage of the AND-pyramid is that, for operations such as region-growing, the graph diameter of a region is reduced by the presence of paths through higher levels of the pyramid.

The development of one level of the pyramid is accomplished by

$$[\text{AND\_Match}(\text{x xxx xxx xxx 1111}) \mid (1 - Q_{L-1})](\text{B}).$$

This applies the AND_Match operator to a pyramid, requiring all four of a node's children to be foreground before setting a new node as foreground. The restriction prevents the base level of the pyramid from being changed. If we take the transitive closure of this operation,

$$[\text{AND\_Match}(\text{x xxx xxx xxx 1111}) \mid (1 - Q_L)]^*(\text{B}),$$

the pattern match is applied repeatedly until further application results in no change to the result image. At this point, the definition of an AND-pyramid is met in the result pyramid. A concrete example of the construction of an AND-pyramid will appear later, as an example of the use of HCL.

2.5. Array Languages

A number of programming languages have been developed which allow conceptually parallel operations to be performed on arrays and matrices. An example of a language of this sort is APL, written in the early 1960s. APL has an exceptionally rich set of operators, permitting such operations as addition of matrices, dot and cross products, and various matrix restructuring operations [4, 5]. The language has been criticized, in fact, for having such a rich set of operators as to be difficult to comprehend. Also, the language emphasizes the use of vector and matrix operations nearly to the exclusion of serial flow of control; consequently, the flow of control constructs are minimal and primitive.

An alternative approach to the development of a language with parallel constructs is the addition of parallel constructs to an existing serial language; two examples of such extension languages are Matrix Pascal [2] and FORTRAN 8X, a new FORTRAN standard which will include such operators. An advantage which may be seen in these languages over APL is the retention of the comparatively rich scalar and control flow constructs from their base languages. As such, they retain the considerable investment in language design from the base language and build on it, rather than having to start from scratch.

Finally, a number of special-purpose languages have been developed providing parallel operations for parallel processors. Some of these languages include Actus for ILLIAC IV [9], Parallel Pascal for the Massively Parallel Processor [12], and Vector C for the Cyber 205 [7]. All of these provide for parallel operations on arrays as built-in capabilities and so should be considered as special cases of the parallel-extension languages described in the previous paragraph. These languages are typically flawed by idiosyncracies from their processor architectures being reflected in the language. So, for example, Vector C has severe limitations on the possible combinations of vectors in operations, reflecting the instruction set of the Cyber 205.

A promising development is the implementation of parallel-extension languages on parallel hardware, for example, the Connection Machine implementation of FORTRAN 8X. This brings the power of a massively parallel SIMD machine to bear on a general-purpose language, rather than one solely for the CM community.

A survey of languages for vector and cellular processors

may be found in Perrot and Zarea-Aliabadi [10]. A flaw in all of these languages, from the perspective of the development of pyramid algorithms, is that none of them support the pyramid data structure. HCL is intended to meet that need.

3. DESIGN GOALS AND CHOICES

The goal of this project was to support the pyramid data structure and pyramid operations by embedding Hierarchical Cellular Logic in a programming language. To do this, a careful examination of the language and the notation was required, to determine the best marriage of the two, remaining true to the philosophy of each while producing a seamless blend. As will be seen below, this did lead to compromising the notation; in particular, the repeated operations and transitive closure are not implemented due to the prior existence of `while`, `for`, and `do-while` loops in C. Also, the notation for restricting an operation is based on the syntax of the C `?:` conditional expression operator. This results in algorithm descriptions which are not as compact as those in the notation, but it avoids redundant constructs in the language.

As discussed above, HCL was developed in response to a specific problem: expressing pyramid algorithms naturally, clearly, and concisely. In addition, it was desired that the implementation be simple, flexible, and easily modified, so that the language could be modified to reflect experience with actually using it. These two goals led to a number of subgoals and choices, as follows:

- An existing language should be extended, rather than a totally new language created. There are a number of motivations for this. Most importantly, the language is intended for use in massively parallel SIMD environments. As such, it might best be considered as a serial language supporting parallel operations. This implies that adopting the serial control flow constructs from a suitable existing language, and adding parallel data types and operators, is a viable strategy. Developing yet another set of serial constructs, with semantics similar to those of all the other serial languages in existence, would be a diversion of effort from the primary objective. The implementation can also be simpler in this case, as the extensions may be placed "on top of" an existing implementation. The particular language chosen is C. It is well suited to this use as a base language, for two reasons. First, the language itself already possesses a rich set of operators for complicated expressions, which are well suited to overloading as is done in HCL. Some examples of useful constructs in C are its bit manipulation operators, which are useful for work with binary images, and its `?:` operator, which is used here to implement the restriction operators of HCL. Second, C compilers are already based on preprocessors manipulating the input language. An HCL implementation which simply adds an extra pass through another preprocessor simplifies prototype implementations and improves portability. This is exactly how the HCL prototype was developed, with pyramid constructs translated into procedure and function calls to a pyramid processing library.
- The pyramid data structure should be a first-class type in the language, like structures and arrays. Most of the extensions should be in the underlying language syntax, rather than in the form of a procedure or macro-call library. This assists in giving a sense that elementary constructs are indeed elementary and makes performing pyramid operations more intuitive.
- Rather than creating a large number of new constructs, existing language features should be extended for use with the pyramid data structure and pyramid operations wherever possible. There are two motivations for this goal. First, it was desired to maintain the internal consistency of the language to the greatest extent possible, and second, it was felt that adding unnecessary new constructs would create unwarranted clutter and complexity. This was particularly true as the new constructs would (of necessity) relate only to the pyramid operations, violating the previous goal of a seamless integration. This adheres to the language design principles of simplicity, orthogonality, and regularity.

4. LANGUAGE FEATURES

HCL consists of a set of extensions to the programming language C for performing low-level vision algorithms. Four modifications are made to C. First, new data types for pyramids and patterns are added to the language and existing operators are overloaded to provide operations on them. Second, new operators for pattern matching are added to the language. Third, three new statements are added to the language to control the processing environment. Finally, a small number of procedures and functions are provided for input, output, and routing.

4.1. Data Types

Two fundamental data types are added to the language to permit the manipulation of pyramids: pyramids and patterns. These are both first-class, elementary data types much like integers or floating point numbers. As with other data types, it is possible to have arrays of pyramids or patterns, pointers to them, and so forth.

4.1.1. *Pyramids*

In HCL, the pyramid data type is a direct implementation of the pyramid data structure, as defined before. The nodes of a pyramid may be any type in C other than pointers (this reflects a common restriction on cellular processors, which are typically unable to perform indirect memory accesses).

Pyramids are added to the possible C declarations by adding a storage class `pyramid` to the existing set (`auto`, `register`, `static`, `extern`, and `typedef`). Variables declared using the `pyramid` storage class are declared as struct-declarators, rather than simply as declarators (as is the case with other storage classes). This has the effect of allowing precision specifications in pyramid declarations (as shown in the following examples). In addition, a type-specifier of `bit` is added to the existing set (`char`, `short`, `int`, etc.).

Declaring a pyramid reserves space for an eight-level pyramid, with sufficient space for each node. Some examples of valid declarations are:

```
pyramid input_pyramid:8;

unsigned int pyramid output_pyramid;

pyramid struct {
    float x, y;
} d;

struct {
    int a, b, c;
    bit pyramid d;
};
```

The first declares a pyramid of eight-bit, signed nodes. The second has an unsigned integer at each node. The third has a structure at each node, whose two elements are each floating point. The fourth declares a structure consisting of three scalar elements, and a pyramid of bits.

On hardware supporting arbitrary length data, as in bit-serial SIMD processors, pyramids will be packed to occupy precisely their requested space. Otherwise, they are guaranteed to have at least the specified precision.

There are a number of predefined constant-valued pyramids. The `true` and `false` bit pyramids hold the Boolean values true and false, respectively. In accordance with C practice, true is represented as 1 and false as 0. The unsigned `level` pyramid contains, at each node, that node's level in the pyramid. The unsigned `x` and `y` pyramids contain the x and y addresses of each node.

The presence of multiple resolutions reflects the need to optimize performance on bit-serial hardware. In this environment, maximum speed is obtained by keeping all operands and intermediate results to the minimum length necessary to maintain accuracy. It would not be at all surprising, for example, for an algorithm to obtain eight-bit data; perform arithmetic involving weighted sums of these data, which requires floating point; and finally produce one-bit results.

4.1.2. *Patterns*

Patterns are also available as data types. As in Hierarchical Cellular Logic, a pattern consists of 14 elements. The possible values and order of the elements also correspond to Hierarchical Cellular Logic.

A constant pattern takes the syntax

P PPP PPP PPP PPPP.

A variable of type pattern may also be declared; `pattern` has been added to the list of type-specifiers, so a pattern variable is declared as

`pattern` *name*;

When a pattern is declared, sufficient space is reserved to hold all of the elements of the pattern.

4.2. Operations

A variety of operations is provided. In general, an attempt is made to preserve the orthogonality of the language by extending each existing operator to include the new types.

4.2.1. *Overloaded C Operations on Pyramids*

The complete set of C Boolean, logical, arithmetic, shifting, comparison, and selection operations is extended to pyramids in HCL. The operations are performed component-wise: adding one pyramid to another means adding each element of the first to its corresponding element in the second. Comparison and logical operations result in bit pyramids.

The result of an arithmetic or comparison operation between two pyramids is long enough to correctly represent any possible result, up to the limits of the precision of the underlying hardware. An addition of two pyramids will be one bit longer than the longer of the two, a comparison will have a one-bit result, and so forth. On an assignment, the result will be truncated to fit the destination. When arithmetic is performed on two integer pyramids of different lengths, the shorter one is extended to match the longer one. If it is signed, it is sign-extended; if not, it is zero-extended. An exception to this rule is the bit pyramid; these are zero-extended for arithmetic operations and assignments and sign-extended for logical operations. This corresponds to the intuitive semantics of using bit pyramids in a number of algorithms developed to date.

The `C ?:` operator is supported, with the following semantics: in an expression

exp1 `?` *exp2* `:` *exp3*

exp1 is evaluated first and must result in a bit pyramid or a scalar. If it is a bit pyramid, *exp2* is evaluated at all nodes for which *exp1* is `true`. At all nodes for which it is `false`, *exp3* is evaluated. If *exp1* is a scalar, the normal C semantics apply.

This syntax, rather than the syntax of Hierarchical Cellular Logic, is used for restricting operations to subsets of a pyramid. In addition to fitting the base language syntax, the `?:` operator is more flexible.

4.2.2. *Overloaded C Operators on Patterns*

The C Boolean operators are also overloaded to operate on patterns. The intent here is to provide a mechanism in the language for the creation of complex patterns based on simpler building blocks. The definitions of intersection and union are provided by ordering the possible pattern element values `0` < `1` < `x`. We now define the intersection of two pattern elements as their minimum, and the union as their maximum. Inverting a pattern will replace a `0` element with a `1`, replace a `1` element with a `0`, and leave an `x` element unchanged.

For a pair of well-formed patterns, applying the AND_match of the intersection of two patterns to a pyramid will yield the same result as applying each of the original patterns individually and taking the intersection of the resulting pyramids. Similarly, for well-formed patterns, applying the OR_Match of the union to a pyramid will give the same result as the union of the result of applying the two patterns.

A pair of patterns is considered well formed for these purposes if there are no elements containing a `0` in one of the patterns and a `1` in the other. In effect, the goal was to establish a set of definitions describing the interactions of don't-cares with other pattern elements; the interactions between `0` and `1` elements are merely included for completeness. The choice of operations which do not establish a Boolean algebra is deliberate. A Boolean algebra would require that `x & ~x ≡ 0`, and `x|~x ≡ 1`, which would not correspond to the intuition being captured here.

The use of these operators on patterns provides a more intuitive and readable mechanism for defining patterns than simply building them by hand. We can see, for example, that after setting

```
pattern NW, W, SW;

NW = x 1xx xxx xxx xxxx;
W = x xxx 1xx xxx xxxx;
SW = x xxx xxx 1xx xxxx;
```

the obvious expression for a pattern matching any of the three nodes to the left of the center of a neighborhood is

```
NW | W | SW.
```

This has proved to be particularly useful in defining complex patterns for improved pyramid generation algorithms [11].

4.2.3. *Pattern-Matching Operations*

Two new operators are provided for pattern matching in pyramids. The operations are `&⟨`*pattern*`⟩` (AND_Match) and `|⟨`*pattern*`⟩` (OR_Match). These operators have semantics identical to those of the corresponding Hierarchical Cellular Logic operators, described in Section 2.4. The results of the match are also partially controlled by the `boundary` statement, which we will describe later.

Syntactically, the pattern-matching operators are treated as unary operations acting on a pyramid and have the same precedence as unary minus or bitwise negation. A statement such as

```
pyr1 = pyr2 & |⟨x x1x 1x1 x1x xxxx⟩ pyr3;
```

is interpreted as applying an OR_Match to the pyramid named `pyr3`, then ANDing it with `pyr2`, and finally assigning the result to `pyr1`.

4.3. New Statements

Three statements are added allowing control of the processing environment. The `boundary` statement is used to control the behavior of pattern matches on the boundary of a pyramid, and the `visible` and `invisible` statements are used to control display of pyramids.

4.3.1. *Boundary*

A difficulty with performing pattern-matching operations in the pyramid lies in defining the results at the boundaries. For example, the root of the pyramid has no parent, the nodes at the base level have no children, and so forth. The `boundary` statement is used to give values to these border nodes. By default, the boundaries are treated as though the nodes just beyond the limit of the pyramid have the value `false`. However, the boundary may be treated as a `true` region or given the same value as the nearest node in the pyramid with the statements

```
boundary true;
boundary replicate;
```

The default may be restored with

```
boundary false;
```

4.3.2. *Visible and Invisible*

It is possible to make pyramids visible or invisible through the use of the `visible` and `invisible` statements. The statement

```
pyr visible;
```

causes the system to display the contents of *pyr* on a raster display screen. Whenever an assignment is made to *pyr*, the new contents are displayed. The appearance of this display can be seen in Figs. 4 through 8; when the entire pyramid is visible, it is shown as a number of arrays, with Level 0 at the top of the image and Level 7 at the bottom.

A number of options exist for the `visible` statement. First, it is possible to display the pyramid at an arbitrary location on the screen by using the `at` option. The statement

pyr `visible at` (x, y);

selects the screen position for the lower left-hand corner of the pyramid. The screen is defined in (x, y) node coordinates, with $(0, 0)$ at the lower left-hand corner of the screen. The default location is $(0, 0)$.

The size of the pyramid may also be expanded for display purposes, using the `size` option,

pyr `visible size` *expand*;

The expansion is accomplished through node replication. An expansion of 1 corresponds to no expansion at all.

Either the entire pyramid or only its base plane may be displayed, depending on the `as array` option. If this parameter is not present, the pyramid is displayed as a pyramid (i.e., as a stack of arrays); if it is present, only the base plane is displayed.

The options may be combined in any order, so the statement

`inpyr visible as array at` (10, 20);

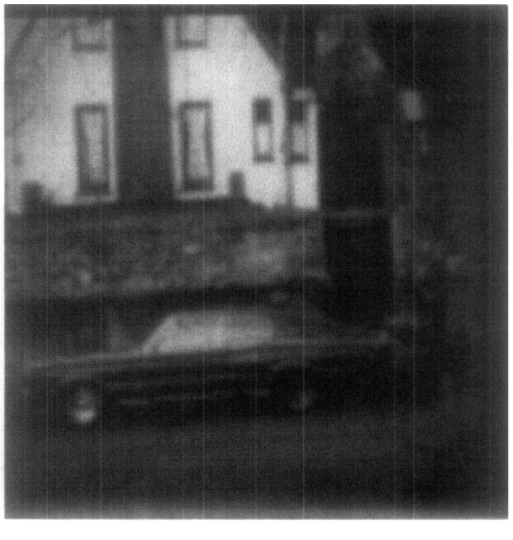

FIG. 4. Raw image data.

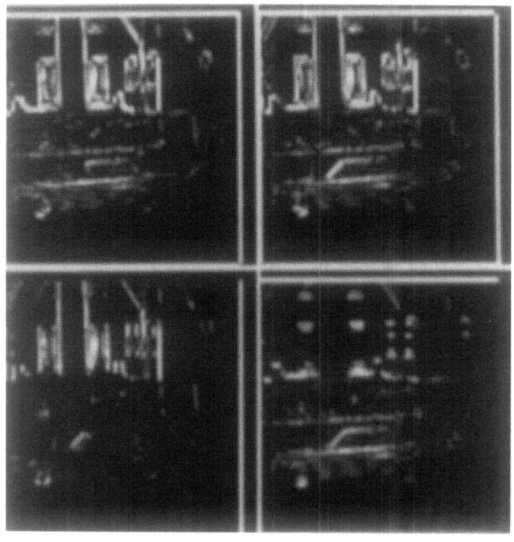

FIG. 5. Gradient components.

would display the base of the pyramid named `inpyr` with its lower left corner at location (10, 20) on the screen.

The operation complementary to `visible` is `invisible`. The statement

`invisible` *pyr*;

stops the display of *pyr*. It is removed from the screen at that point.

4.4. Procedures and Functions

While the majority of the changes which have been made to C in producing HCL are syntactic or semantic extensions to the language, a number of additional facilities are pro-

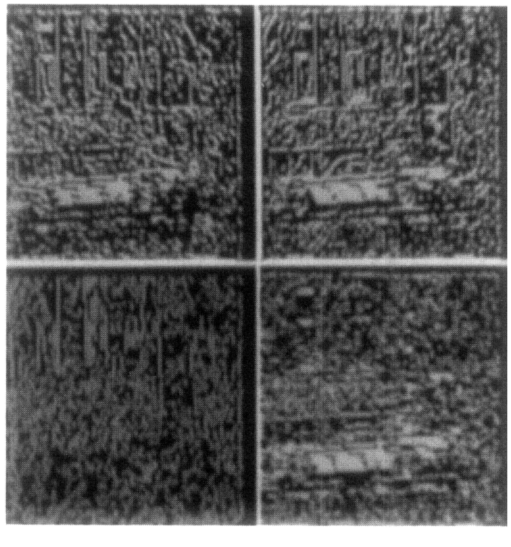

FIG. 6. Filtered local maxima.

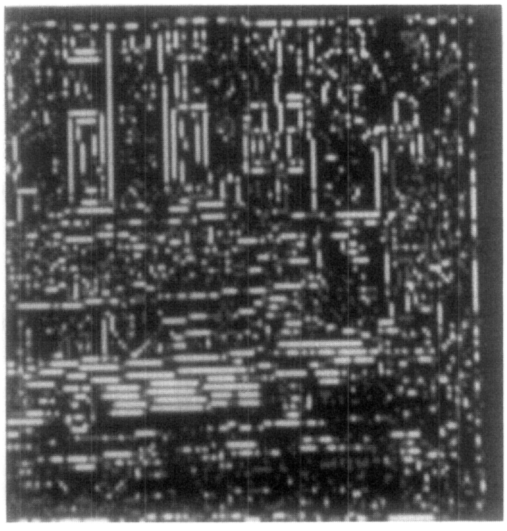

FIG. 7. Edge support map.

vided which are expressed more naturally as procedures and functions. These generally deal with pyramid input and output or user interaction.

4.4.1. *Nzr, Count, and Sum*

Three functions, `nzr()`, `count()`, and `sum()`, are used to obtain statistics regarding image contents.

The `nzr()` function is the simplest of the three. The expression

`nzr(`*expr*`)`

returns a byte, with each bit corresponding to one level in the pyramid. A bit is set `false` if its level contains only zero nodes, and `true` otherwise.

The `count()` operator,

`count (`*expr*`),`

counts the number of nonzero nodes in the pyramid, returning a short integer.

The `sum()` operator is the most complex of the three. The expression

`sum(`*pyr*`)`

adds together the values of all the nodes in the pyramid, returning a long integer. Note that for a bit pyramid, the `count()` and `sum()` operators are equivalent except for the length of the returned value.

4.4.2. *Seed*

Users are able to enter bit pyramids containing seed nodes through the use of the `seed()` function, which takes as argument a pyramid and returns a bit pyramid. A call to

`seed(`*pyr*`)`

causes a cursor under locator control to appear on the

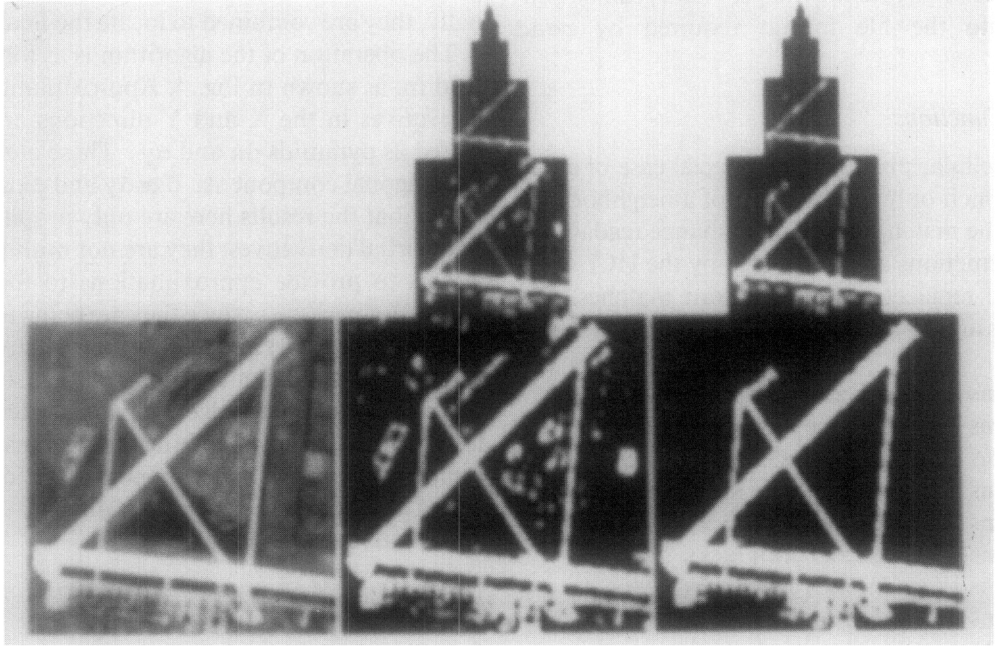

FIG. 8. Pyramid region filling.

screen. The user may move this cursor to an arbitrary location on the screen and press a button device. If the cursor is located over the pyramid given as an argument to `seed()`, HCL determines which node is covered and produces a bit pyramid with value `false` at all locations except the user-specified node, which is `true`. Pressing the button when the cursor is not over the specified pyramid has no effect; the function will not return until the condition is satisfied. Note that the specified pyramid must be visible before calling `seed()` (an error will occur otherwise).

4.4.3. *Input*

The `input()` function reads an input image from an input processor and returns a pyramid. The input processor is expected to provide a 128 × 128 image. The returned pyramid has the image read from the input processor at its base level and is zero at all other levels. This is expected to be the most common mechanism for obtaining data for analysis in HCL programs.

4.4.4. *Read_image and Write_image*

A facility exists to read and write image files. Calling the `read_image(`*name*`)` function with the name of an image file as its argument will return an image read from disk.

The `write_image(`*expr*, *name*`)` procedure takes a pyramid and a file name as arguments and writes the pyramid to the file.

Ordinarily, a file read with `read_pyramid()` will have been created by a previous call to `write_pyramid()`. Consequently, `input()` remains the primary mechanism for obtaining data from the environment. Data obtained from other sources (e.g., satellite imagery data) can be converted easily into the file format required by `read_image()`.

4.4.5. *Routing Functions*

Routing in a cellular processor is a special case of a pattern match, in which only one element of a neighborhood is considered in the match. In order to enhance readability, a set of routing functions are recognized by the HCL compiler as syntactic replacements for pattern matches. The routing functions are

`P(`*expr*`)` means `&⟨1 xxx xxx xxx xxxx⟩` *expr*
`NW(`*expr*`)` means `&⟨x 1xx xxx xxx xxxx⟩` *expr*
`N(`*expr*`)` means `&⟨x x1x xxx xxx xxxx⟩` *expr*
`NE(`*expr*`)` means `&⟨x xx1 xxx xxx xxxx⟩` *expr*
`W(`*expr*`)` means `&⟨x xxx 1xx xxx xxxx⟩` *expr*
`E(`*expr*`)` means `&⟨x xxx xx1 xxx xxxx⟩` *expr*
`SW(`*expr*`)` means `&⟨x xxx xxx 1xx xxxx⟩` *expr*
`S(`*expr*`)` means `&⟨x xxx xxx x1x xxxx⟩` *expr*
`SE(`*expr*`)` means `&⟨x xxx xxx xx1 xxxx⟩` *expr*
`CNW(`*expr*`)` means `&⟨x xxx xxx xxx 1xxx⟩` *expr*
`CNE(`*expr*`)` means `&⟨x xxx xxx xxx x1xx⟩` *expr*
`CSW(`*expr*`)` means `&⟨x xxx xxx xxx xx1x⟩` *expr*
`CSE(`*expr*`)` means `&⟨x xxx xxx xxx xxx1⟩` *expr*

5. EXAMPLES

Two examples of algorithms developed using HCL are now presented. The first shows the arithmetic and logical capabilities of the language by locating edges in an image using a simple edge detection operator, and the second shows the use of a pyramid in performing region-filling. In this second example, we also compare the notation for the algorithm with that of Tanimoto.

5.1. Edge Detection

The edge detector program shown as Algorithm 1 exercises the arithmetic capabilities of HCL. It is in some sense simpler than the region-filling algorithm to follow, as it requires no iteration. Also, it uses only the base level of the pyramid; no hierarchical operations are required.

Edge detection is an extremely common, important and frequently time-consuming task in image processing and computer vision applications. The goal is to locate places in the image where strong changes are taking place in image intensity, in order that a line drawing may be produced showing the important features and ignoring the areas in the image where nothing of interest is occurring.

The example algorithm is a simple derivative-based edge detector. After the partial derivatives in the input image are measured in four directions, the four partial derivative images are searched for local maxima in order to establish edge candidates. These candidates are then passed through a simple noise filter to weed out isolated, likely noisy nodes. Finally, they are combined to locate the final edge set.

The operation of the algorithm is as follows: an image is read in, as shown in Fig. 4. Approximations to the partial derivatives in the X and Y directions are computed and stored as pyramids `dx` and `dy`. These are used to generate the diagonal components, `dxmdy` and `dxpdy`. It should be noted that the results here are only proportional to the actual partial derivatives; they are not multiplied by constant factors to provide approximations to the actual partials. This is not necessary, since they are being used only for local maximum searches and so only their relative values are relevant. The four gradient components are shown in Fig. 5.

A set of candidate edge pixels is located for each of the four components by searching for local maxima in the magnitude of the gradient components. The candidate sets are stored in the `dx_edges, dy_edges, dxmdy_edges,` and `dxpdy_edges` bit pyramids.

Algorithm 1: *Edge Detection*

```
#define ABS(x) ((x>=0)?x:-x)
main(){
```

```
unsigned short pyramid inpyr;
short pyramid dx, dy, dxmdy, dxpdy;
bit pyramid dx_edges, dy_edges,
dxmdy_edges, dxpdy_edges;
unsigned pyramid edge_set:2;

/* Establish graphics */
edge_set visible as array size 2;

/* Read in an image */
inpyr = read_file
("/special/vision/images/chargerl");

/* Get derivative magnitudes */
dy = N(inpyr) - S(inpyr);
dx = E(inpyr) - W(inpyr);
dxmdy = ABS(dx - dy);
dxpdy = ABS(dx + dy);

dx = ABS(dx);
dy = ABS(dy);

/* Get candidate edge sets */
dx_edges = (dx > E(dx)) & (dx > W(dx));
dy_edges = (dy > N(dy)) & (dy > S(dy));
dxmdy_edges = (dxmdy > NW(dxmdy)) &
(dxmdy > SE(dxmdy));
dxpdy_edges = (dxpdy > NE(dxpdy)) &
(dxpdy > SWdxpdy));

/* Filter edge candidate sets */
dx_edges &= |⟨x 111 1x1 111 xxxx⟩dx_edges;
dy_edges &= |⟨x 111 1x1 111 xxxx⟩dy_edges;
dxmdy_edges &= |⟨x 111 1x1 111 xxxx⟩
dxmdy_edges;
dxpdy_edges &= |⟨x 111 1x1 111 xxxx⟩
dxpdy_edges;

/* Get edge support measure */
edge_set = dx_edges + dy_edges +
dxmdy_edges + dxpdy_edges;
}
```

In order to eliminate candidates resulting from noise in the image, each candidate set is matched against a pattern requiring that a candidate pixel have at least one neighbor which is also a candidate. If none of the neighbors are also candidates, the original assignment is assumed to be in error and the pixel is removed from the candidate set. The filtered candidate sets are shown in Fig. 6.

Finally, the likelihood that each pixel is an edge is determined by counting the number of candidate sets in which it appears. The result of this operation is shown in Fig. 7.

The figures were generated by several executions of the program, with different pyramids visible on different executions; the listing shows the program for the final execution.

5.2. Pyramid Region-Filling

The region-filling program given as Algorithm 2 makes greater use of the pattern matching and pyramidal abilities of HCL and shows some features and constructs not apparent in the edge detection example.

Region-filling is a common operation in both vision and graphics; in vision, it is used as a step in component labeling algorithms, and in graphics, it is used in shading. The algorithm shown here is in the class of "flood-fill" algorithms, extended to make use of the pyramid.

Tanimoto presents this algorithm in [14]. The first step is to define an "AND-pyramid" as described in Section 2.4. The AND-pyramid was shown in that section to be created by

$$\text{AND\_pyr}(B) \equiv [\text{AND\_Match}(\text{x xxx xxx xxx 1111}) \mid (1 - Q_L)]^*(B).$$

Given this definition, we may take two pyramids X and Y, with X containing a single foreground node called the seed and Y containing a thresholded image. The function

$$\text{PyramidFill}(X, Y) \equiv [\text{OR\_Match}(\text{1 111 111 111 1111}) \mid \text{AND\_pyr}(Y)]^*(X)$$

iteratively expands a foreground region surrounding the seed node, but restricted to foreground in the AND-pyramid. This expansion continues until the region around the seed has been filled. The HCL example below demonstrates this algorithm but also includes the details of acquiring the image pyramid and thresholding it.

This example begins by thresholding the input image `in_pyr`, giving `thresh_pyr`. Second, an AND-pyramid is constructed above the threshold image, also in `thresh_pyr` (this corresponds to Tanimoto's definition of AND_pyr). Finally, starting from a seed node, a region in the pyramid is filled using a dilation operation, resulting in `filled_pyr` (this corresponds to Tanimoto's definition of PyramidFill).

Both the phases of pyramid construction and region filling in this example need to iterate until no change takes place in an image. In both cases, a pyramid called `old_pyr` is used to check this. In both loops, we begin by copying the working pyramid into `old_pyr`, then perform an operation on the working pyramid, and then take the exclusive OR of the working pyramid and `old_pyr`. Any node changed by the working step will show up in the exclusive OR, and so *nzr* will return a nonzero result. In the first loop, used to build the AND-pyramid, the loop is implemented as a `for` loop; in the second, used to perform the actual fill, the loop is a `do-while`.

This also shows the use of the ?: operator that HCL uses for restricting operations to subsets of the pyramid. In the first loop, the AND_match is restricted to the correct level; in the second, the OR_match is restricted to foreground.

For the particular image shown, use of the hierarchical algorithm allows the pyramid to be filled in roughly half the time of an algorithm using a cellular processor acting on an array.

All of these operations are shown in Fig. 8, with the input image on the left, the constructed pyramid in the middle, and the filled pyramid on the right.

Algorithm 2: *Pyramidal Region Filling*

```
main( ) {
  short pyramid in_pyr;
  bit pyramid thresh_pyr, filled_pyr,
  old_pyr;
  int L;

  /* Establish graphics */
  in_pyr visible as array at (62, 128);
  thresh_pyr visible at (192, 128);
  filled_pyr visible at (322, 128);

  /* Read and threshold an image */
  in_pyr = read_image
  ("/special/vision/images/airport2");
  thresh_pyr = in_pyr > 190;

  /* Build an AND pyramid */
  for (L = 6; nzr(old_pyr ^ thresh_pyr); L--)
  {
    old_pyr = thresh_pyr;
    thresh_pyr = (Level == L) ?
      &⟨x xxx xxx xxx llll⟩old_pyr : old_pyr
  }

  /* Obtain a seed node */
  filled_pyr = seed(in_pyr);

  /* Fill the region identified by the seed node
  */
  do {
    old_pyr = filled_pyr;
    filled_pyr = thresh_pyr ?
      |⟨l lll lll lll llll⟩filled_pyr : false;
  } while (nzr(old_pyr ^ filled_pyr));
}
```

6. IMPLEMENTATIONS

A version of HCL has been implemented and used in developing computer vision algorithms. While PCLIP II is not yet available, the current HCL implementation models the eventual environment fairly closely. A precompiler converts the HCL program into C by translating the pyramid constructs into calls to a communications library. This allows the HCL program to act as a frontend process, executing the serial portion of the algorithm and sending pyramid operations to a second process for execution.

This somewhat indirect method was chosen in order for PCLIP II simulations to be usable for executing HCL programs. As PCLIP II itself functions as a backend processor, this results in an HCL implementation which will be correct when the actual hardware becomes available and will not require modification at that time. In addition, it is possible to run with register transfer simulations, instruction set simulations, or working hardware with the particular implementation completely hidden from the user's HCL program. So far as the program can tell, the only difference between these environments is in the execution speed.

While portability was not one of the original goals of HCL (it was intended specifically as a language for PCLIP II), the resulting language is in fact suitable for a variety of architectures. With the language implementation divorced, in large measure, from the pyramid operations code, the degree to which HCL can be ported to another environment is very much a function of the extent to which pyramid operations (independent of language) can be supported in that environment. As was mentioned before, PCLIP II has been modified over time to support HCL. At this point, the instructions generated by the HCL compiler are strictly a set of generic pyramid operations, such as "perform an n-bit addition between operands starting at bits x and y." Consequently, implementation on parallel hardware other than PCLIP II is just a matter of writing a pyramid operations interpreter for that processor. PCLIP II itself is simply a special case with the interpreter in microcode and the microarchitecture reflecting the pyramid topology.

As a specific example, it is very unlikely that it will be possible to implement HCL on nonpyramidal cellular processors with any degree of efficiency, due to communications costs. This is inherent in the implementation of any pyramid algorithm, regardless of language; it seems likely that a language similar to HCL but using arrays rather than pyramids would be possible. Of course, Parallel Pascal is very much in this mold. An implementation on the Connection Machine would be possible, as arbitrary communications are relatively inexpensive on that machine.

On the other hand, pipelined processors would seem to be likely candidates for successful implementations. A very challenging, and interesting, project is that of implementing HCL on a processor such as the FPS T-Series. This architecture uses a number of pipelined vector processes at the vertices of a hypercube, with operating systems functions provided by a host. Two challenges are present here, both related to the distribution of pyramid nodes across the nodes of the hypercube. First, optimal efficiency requires a careful balancing of operations on a node and communications between nodes, as interprocessor communications are much

slower than vector operations on this machine. Second, balancing the time required for the operations on each node is difficult due to the varying vector lengths at each level of the pyramid. Neither of these problems, however, is expected to be insoluble, and a T-Series implementation is planned. This implementation will be the first to actually use a physical backend processor with the HCL implementation, as discussed above. The particular relationship between the frontend and backend is also of interest here. The frontend is a SUN 3/160C (a Unix workstation equipped with a high-resolution graphics display). The SUN is connected via Ethernet to a MicroVAX II, which in turn is the host for the T-Series. In the planned implementation the SUN will execute an HCL program, with pyramid instructions sent to the MicroVAX. The MicroVAX will interpret them and pass them on to the vector processing nodes. Visible pyramids and the results of `nzr()`, `count()`, and `sum()` operations will be sent back to the SUN. This implementation will permit the SUN to execute serial operations concurrently with the pyramid operations being performed by the T-Series. It was consideration of the requirements of this implementation which led to some of the language definitions regarding storage of pyramids.

Finally, it has been demonstrated that pyramid operations are easily performed on the PIPE [6]. We would anticipate, therefore, that HCL could also be efficiently ported to this machine.

7. CONCLUSIONS

HCL represents an attempt to embed a mathematical notation in a programming language while remaining true to the philosophies of each. By paying careful attention to the existing language features, and the semantics of the notation, it has been possible to create a successful marriage between the two.

Since the notation was intended for a particular, practical purpose (expressing operations on pyramids), embedding it in a programming language provides a good mechanism for expressing pyramid algorithms. This has led to a much simpler environment for developing and testing these algorithms.

While the language has been successful in meeting its goals, additional features would be desirable and should be investigated. These relate to the generality of the topologies and routing functions provided.

First, a weakness of this language, and of all other similar SIMD languages, is that it is tied to a particular topology—in this case, pyramids. Ideally, it should be possible to develop a language which will be useful in a variety of environments, including arrays, pyramids, overlapped pyramids, hypercubes, and other multidimensional topologies. Such a language would be capable of defining the local neighborhood of a node and then performing SIMD operations on this local neighborhood. The topology defined would need to be mapped as efficiently as possible to the hardware on which an algorithm is to be performed.

We can see at least four levels of algorithm/architecture topology matches, with varying difficulties of implementation. First, if the algorithm topology is identical to the architecture topology, no work is required to match the architecture to the algorithm. Running pyramids on PCLIP II or arrays on MPP would be in this class. Second, if the algorithm topology is simply a higher-dimensional analog to the architecture topology, the mapping is trivial since processing elements can simply emulate multiple nodes. Using octtrees on PCLIP II or a cube (128 on a side) on MPP would be in this class. Third, closely related topologies may require simple routing or renumbering of the processing elements. Overlapped pyramids on PCLIP II or three-dimensional arrays on the Connection Machine are in this class. This class would be the most difficult to work with, since it requires finding subgraph isomorphisms. Finally, some topologies are simply incompatible in any realistic sense. Hypercube algorithms on PCLIP II or pyramid algorithms on the Connection Machine would be completely impractical.

This would provide a truly hardware-independent SIMD language, and as such would be useful across a broad class of underlying architectures. Further development of SIMD languages should proceed along these lines.

A second potential enhancement is in the area of the routing functions between nodes in the pyramid. The functions defined (AND_Match and OR_Match) were developed as extensions of bitwise combining functions. It has been pointed out that an alternative view of the functions would consider them as two special cases of reduction. It would be possible to define matching as a general reduction operator, applying an arbitrary associative, commutative operation (*e.g.*, addition or multiplication) to the nodes in the neighborhood matching the pattern vector. The bit pattern vector itself could be replaced with a vector of weights to be applied to the neighbor elements. This also warrants further investigation.

ACKNOWLEDGMENTS

The reviewers were exceptionally helpful. Several of the features of the language, and much of its generality past a narrow "PCLIP II language," are a direct result of their responses to an early version of the paper.

REFERENCES

1. Gilmore, P. A., Batcher, K. E., Davis, M. H., Lott, R. W., and Burkley, J. T. Massively parallel processor. GER-16684, Goodyear Aerospace, Akron, OH July 1979.
2. Gudenberg, J. W. von. An introduction to matrix Pascal: A Pascal extension for scientific computation. In Miranker, W. L. (Ed.). *A New Approach to Scientific Computation.* Academic Press, London, 1982, pp. 225–246.

3. Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.
4. Iverson, K. E. APL in exposition. 320-3010, IBM Corp., Jan. 1972.
5. Iverson, K. E. An introduction to APL for scientists and engineers. 320-3019, IBM Corp., Mar. 1973.
6. Kent, E. W., and Tanimoto, S. L. Hierarchical cellular logic and the PIPE processor: Structural and functional correspondence. *Proc. 1985 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* Miami Beach, FL, Nov., 1985, pp. 311–319.
7. Li, K.-C., and Schwetman, H. Vector C: A vector processing language. *J. Parallel Distrib. Comput.* **2,** 2 (May 1985), 132–169.
8. Pass, S. D. A user's guide to CLIP4. 81/7, Image Processing Group, Department of Physics and Astronomy, University College London, London, Sept. 1981.
9. Perrot, R. H. A language for array and vector processors. *ACM Trans. Programming Languages Systems* **1,** 2 (Oct. 1979), 177–195.
10. Perrot, R. H., and Zarea-Aliabadi, A. Supercomputer languages. *ACM Comput. Surveys* **18,** 1 (Mar. 1986), 5–22.
11. Pfeiffer, J. J., Jr. Heuristic improvements for pyramid generation with reduced graph diameter, In preparation, 1988.
12. Reeves, A. P. "Parallel Pascal: An extended Pascal for parallel computers. *J. Parallel Distrib. Comput.* **1** (1984), 64–80.
13. Tanimoto, S. L. A pyramidal approach to parallel processing. *Proc. Tenth International Symposium on Computer Architecture,* Stockholm, Sweden, June 1983.
14. Tanimoto, S. L. A hierarchical cellular logic. University of Washington Tech. Rep. 83-10-06, Oct. 1983.
15. Tanimoto, S. L., and Pavlidis, T. A hierarchical data structure for picture processing. *Comput. Graphics Image Process.* **4** (1975), 104–119.
16. Tanimoto, S. L., and Pfeiffer, J. J., Jr. An image processor based on an array of pipelines. *Proc. 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* Hot Springs, VA, Nov. 1981, pp. 201–208.
17. Tanimoto, S. L., and Pfeiffer, J. J., Jr. Data processing system having a pyramidal array of processors. U.S. Patent No. 4,622,632, Nov. 11, 1986.

Received November 21, 1986; revised June 1, 1988

JOSEPH J. PFEIFFER, JR., earned the B.Sc. (1979), M.Sc. (1982), and Ph.D. (1986) degrees at the University of Washington. Since 1984, he has been an assistant professor in the Department of Computer Science at New Mexico State University. His current research interests center on the development of computer architectures integrating features required for low-level and high-level computer vision.