PROCEDURES FOR PARALLEL ARRAY PROCESSING ON A PIPELINED DISPLAY TERMINAL

by

Pat Hanrahan

Computer Sciences Technical Report #490

December 1982

Procedures for Parallel Array Processing on a Pipelined Display Terminal.

Pat Hanrahan

Image Processing Laboratory
Computer Sciences Department
University of Wisconsin
Madison, Wi.

Abstract

A variety of special purpose array processing architectures – pipelines, two-dimensional arrays of processors and recently pyramids – have been developed for image processing applications. Although the variety often stems from economic considerations, some rather subtle problems in computational complexity depend on the ability to access information stored in neighboring array locations. The original architectures often have fixed, immediate neighbor addressing; more recent architectures have more flexible, tree-like addressing.

This paper introduces a package of procedures written to emulate different array architectures by taking advantage of the flexible addressing of a pipeline processor. The system processes square arrays of 512 by 512 1-bit elements, allows arbitrary relative spatial addressing, and the ability to selectively freeze local processing. The design issues and implementation is discussed and several examples are described.

Keywords – Array Processing, Pyramids, Programming Systems

CR Categories– I.4.10 [**Image Processing**]: General– *image processing software*, I.5.5 [**Pattern Recognition**]: Implementation– *special architectures.*

1. Introduction

Techniques of parallel processing hold great promise of increasing the speed of common operations in image processing. Many calculations applied to images are pixel intensive – they are applied to every location in the image, and repetitive – that is, they are the same or similar at every pixel. Because typical images are very large, performing even the simplest operation can require a large amount of computational resources and therefore time. For this large class of operations the use of an array processor, a machine that simultaneously computes a local function at every cell over the entire image, can speed the application of the algorithm tremendously.

The University of Wisconsin Image Processing Laboratory (uwipl) has been working to develop parallel algorithms and hardware for these tasks. This work has revolved around a highly parallel computational model for computer vision, the *recognition cone*, developed by Leonard Uhr[1972]. Several serial implementations of this paradigm have been programmed, the latest and most robust is the system, *ICON*, developed by Larry Schmitt[1981]. At a lower level the research has been at two fronts. At the software level a new parallel language, PascalPl (Pascal Parallel Language), was designed to aid in the writing of larger more complicated parallel programs [Uhr, 1981; Stansbury and Uhr, 1980]. At the hardware level several simulations of existing array processors have also been developed. These include a Clip4 and Dap machine language interpreter [Stansbury, 1980]. Lastly, a new design for a 16x16 combination SIMD-MIMD array processor has been developed [Uhr, Lackey and Thompson, 1982].

Unfortunately, all the above projects although emphasizing parallel computation are still confined to a serial computer. For this reason the development of an emulation of an array processor using the color display terminal located in the University of Wisconsin Image Processing Laboratory was undertaken. The emulator provides a set of procedures embedded in Pascal (or C) that implement the instruction set of a true hardware *binary array processor*. This image processor although not truly hardware parallel contains a pipelined arithmetic logic unit which can apply standard arithmetic and logic functions to two input data streams and store the output into an image buffer. At the instruction or program level the machine functionally resembles an array of processors, 512 pixels square, with an instruction cycle time of 30 msec. The best current array processors are much smaller (the Mpp, for example is 128 square) but significantly faster (the Mpp instruction cycle time is 100 nanoseconds). Despite this, for a large class of algorithms the processor can execute an appropriate parallel program at a much faster rate than a conventional serial computer (perhaps 40-100 times that of a Digital Equipment Corporation VAX 11/780). Much more significantly because of the flexible addressing modes of a pipelined architecture, interconnections between processors can be dynamically reconfigured. This reconfigurability makes the pipeline a very desirable machine for the development of a testbed for parallel algorithms for image processing.

2. Design Considerations

Several key issues were addressed during the design of this system. The general goal was to emulate a set of instructions common to existing array processors, particular attention being paid to the *Clip4* (designed by M. Duff[Duff, 1978]), the *Dap* (designed by S. Reddaway[Reddaway, 1978]), and the *Mpp* (designed by K. Batcher[Batcher, 1980]). These are all examples of binary array processors since most of their operations act on single bit planes (although they often contain special capabilities to expedite n-bit operations). These architectures allow standard arithmetic and logic operations to execute in parallel over a two-dimensional array of data. In addition they contain powerful neighborhood addressing to fetch and store data at adjacent memory locations and the important computational ability to freeze or conditionally execute instructions at different locations in the array. Instead of modeling any single instruction set we choose to extract the lowest common denominator based more on essential functional capabilities rather than specific hardware details. This approach also emphasizes the commonality of parallel array processing whether it be on true multiple processor architecture or embodied in an efficient pipeline.

The second major design goal was to investigate the variations on the local-neighbor interconnections of the above array processors to include *power of 2*[Klette, 1981] and/or *pyramid*[Tanimoto, 1981; Dyer, 1982] topologies now being proposed. By augmenting the local interconnection topology with these addition links the diameter of the network is reduced, reducing the time complexity of several region-level image processing algorithms from O(n) to O(log n)[Dyer, 1979]. This is possible with a pipeline because neighbor addressing is performed by scrolling the image plane relative to the processor (since the relative scroll can easily be set dynamically to any distance).

The implementation of the above ideas requires several fundamental capabilities:

(1) Being able to apply the necessary arithmetic and logical operations to a two-dimensional array of data.

(2) Being able to locally freeze, or inactivate, instruction execution.

(3) Being able to perform computations involving data at different relative spatial locations.

Several commercially available pipelined image processors provide the capabilities (although some difficulties are encountered, see the section on the implementation).

The next section describes the overall organization of the array processor and its instruction set. Following this is a section containing several example programs illustrating some common parallel image processing algorithms concluding with an outline of a low-level two-dimensional pattern-matcher based on probabilistic productions or transforms used in a recognition cone[Uhr, 1972; Schmitt, 1981]. Lastly, a short description of the implementation is included for the curious.

3. Overall Organization

The emulator provides the user with what is conceptually a large two dimensional array, 512 by 512, of one bit processing elements all of which execute the same instruction (the so-called *single-instruction multiple-data stream* or *SIMD* architecture). At the global level we can think of the machine as a series of planes; a plane of processors along with a set of memory planes. Instructions operate simultaneously over an entire plane of data because each processor is executing the same instruction. Normally, each processor receives data from the location in the plane immediately under it, but optionally, by specifying a relative x, y scroll an entire memory plane is shifted relative to the natural state and therefore the processor receives data from a neighboring location.

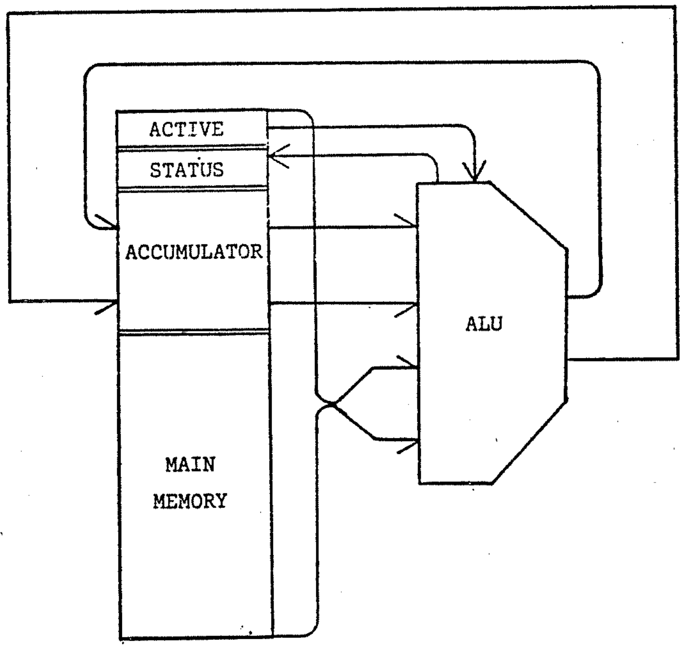

Figure 1. – Local processing element.

At the cellular level each location has a 1-bit processing element and 24 1-bit local memory cells (see Fig 1). The processing element is essentially an arithmetic logic unit or alu since the entire machine uses the same control unit (in fact, for all intents and purposes the control unit is the host computer). The memory locations are numbered from 0 to 23. Eight of these location, numbers 16-23, by default belong to the *accumulator*. The number of bits in the accumulator is greater than one to speed arithmetic calculations in the array.[1] All unary operators operate on the accumulator and all binary operators derive one of their inputs from the accumulator. The results of these operations are then output back into the accumulator. Arithmetic operations involving the accumulator treat them in n-bit 2's complement form. In addition to the 8 memory locations devoted to the accumulator location 0 is set aside as a temporary scratchpad register. It is not wise to store any data in this cell since certain instructions will destroy its contents.

In addition to the local memory each processor contains three special registers, the *status* register, the *active* register and the *display* register. The contents of the active register determines whether the processor at this location in the array is "on" and hence whether the next instruction will be executed at this position in the array. If the value in this register is "true" then the instruction will be executed, if it is "false" than the instruction will not be executed.

The status register contains the results of comparison operations. More on this later.

[1]In fact the number of bits in the accumulator is variable. Note the **||bits** instruction.

Finally, because the emulation is being performed on a display terminal it is possible to see the contents of a memory plane. At any time three planes are continuously being shown; these are coded in different colors. The contents of the active register are shown in blue. The contents of the status register are shown in green. The display register is set to a memory location whose contents are then shown is red. Any of the 24 main memory locations can be stored in this register.

4. Instruction Set

The basic instruction set is divided into several major classes. These include *transfer* instructions, *binary* operators, *unary* operators, *comparison* operators and a set of miscellaneous instructions.

4.1. Transfer instructions

The simplest operations are the group of transfer instructions. These instructions all operate independently of the contents of the active register. The complete source plane is copied to the destination plane, as a result the contents of the destination are completely overwritten, the contents of the source remain unaffected.

The most general instruction is the **||tra**[8] instruction which transfers a source plane to a destination plane with a relative scroll. The most common is the **||mov** instruction which is the same as **||tra** but without the scroll. Two special instructions, **||lda** and **||sta**, load and store the lowest bit in the accumulator. **||frz** transfers its source to the active plane, **||iff** transfers the contents of the status plane to the active plane and **||ffi** transfers true to the active plane.

This group of instructions is unique in that they operate irregardless of whether the processor is active or inactive.

4.2. Binary operators

These operators are all of the form:

accumulator <op>= <source> <dx> <dy>

The accumulator is combined with the source plane scrolled by the amounts "dx" and "dy". The directions of the scrolls are in raster coordinates, with positive x moving rightward and positive y moving downward on the display. "Source" may be any memory location. These operators, unlike the transfer instructions, act in the areas of the image which have the corresponding active bit set to "true". If this bit is "false", then the contents of the accumulator are unaffected, or protected, by the operation. All operations use the current precision of the accumulator, by default 8-bits. This, however, can be changed with the **llbits** instruction. It takes as its argument the size of the accumulator, that is number of bits which ranges from 1 to 8. On the other hand the source is always only 1 bit. Thus before source can be combined with the accumulator it is bit extended to match the precision of the accumulator. During this process all the extended bits are set to 0. The exceptions to this rule are the immediate mode operations. With these instructions the operand value is either a prespecified constant or 0, depending on whether the source is true or false, respectively.

One of the simplest operators is the assignment operator, **||asg**, which assigns the source to the accumulator. This is very similar to a **||lda** instruction ex-

[8] Double vertical bars are used to represent a parallel instruction. This nice mnemonic derived from elementary geometry texts was originally suggested by Len Uhr in PascalPl.

cept that the assignment occurs only in the active regions. The logical operators are **||or** (logical or), **||orn** (logical or with the negation of the source), **||and** (logical and), **||adn** (logical and with the negation of the source) and **||xor** (exclusive or).

The final two instructions are the arithmetic operators, **||adi** and **||sbi.** These operators are the only two that use immediate mode addressing. They can be used to add or subtract a constant depending on the contents of the source. One unfortunate problem with the current implementation is that there is no way to test for an overflow or underflow condition.

4.3. Unary operators

The unary operators affect only the accumulator. As with the binary instructions these instructions act only in the active regions.

Currently there are the following unary instructions. **||clr** clears (sets to zero) the accumulator; **||set** sets (i.e. assigns one) to the accumulator and **||com** forms the one's complement of the accumulator. In addition the following operators are being completed: **||shl** shifts the contents of the accumulator left one bit, and **||shr** shifts the accumulator right one bit.

4.4. Comparison operators

The comparison instructions compare the contents of the accumulator to a given value using the named relational operator. As mentioned previously the accumulator contains a n-bit (by default 8) 2's complement signed number. Comparisons can either be *signed* or *unsigned*. If the result of this comparison is true then the bit in the status plane is set to 1, otherwise the status plane is set to 0. The comparison takes place only at those positions that are active. The status plane is set to false wherever the active plane is false.

The following comparisons are possible: **||eq** (equal), **||ne** (not equal), **||ge** (greater than or equal), **||le** (less than or equal), **||gt** (greater than), **||lt** (less than), **||ugt** (unsigned greater than), **||uge** (unsigned greater than or equal), **||ult** (unsigned less than), and **||ule** (unsigned less than or equal).

4.5. Miscellaneous instructions

The last group of instructions perform a variety of functions. The most important is the **boot** instruction, which initializes the system. This instruction must be executed before any other instruction.

A very useful instruction is the **||tst** instruction. This tests whether the entire operand plane is 0. If there is a bit set at any location of the plane then the result of this is false. This instruction is different than all the others in that it is a function which returns a boolean value.

Another very useful instruction is the **||shw** (show or display) instruction. This instruction sets the display register to indirectly point to an operand memory plane. The contents of this plane are then visible in red on the display monitor.

The entire instruction set is summarized in the table in Appendix A.

5. Example Programs

Perhaps the most common operation in low-level image processing is the computation of a local function of pixels surrounding a center pixel (often the neighborhood is restricted to immediate 4, 6 or 8 connected neighbors by the actual physical links between processing elements, in this emulation this restriction is not enforced). The power of these spatially dependent functions lies in their ability to filter images, or equivalently correlate or match features in an image with features we wish to detect. Such an operation is mathematically called shift-invariant, since the function computed does not depend on the location in the array. The mathematical property of shift invariance means simply that we need to search the entire image for the feature – notice that such an operation is explicitly parallel.

5.1. An Edge Detector

As an example, consider the following non-linear function which returns the result of looking for "edges".

```
procedure edge( src, dst : memory );
 begin
  ||clr;
  ||orn(src, 1, 0);
  ||orn(src,-1, 0);
  ||orn(src, 0, 1);
  ||orn(src, 0,-1);
  ||and(src, 0, 0);
  ||sta(dst);
 end;
begin
 boot;
 edge( 1, 2 );
 ||shw(2);
end.
```

In all the examples in this paper we will show the array instructions embedded in a high level pascal-like language. This reflects the fact the array processor is controlled by a single serial control unit – this unit is in fact a general purpose computer with a much more powerful set of instructions then the array processor. This interplay between serial and parallel code is very powerful, since it permits the use of a high-level programming language to control the execution of the parallel statements. Because there is only a single serial control unit it can be much more powerful. This arrangement justifies in some sense the simplicity of the individual array processing elements since they inherit complexity from the host computer.

This is an example of a very simplistic edge detector. Edges are signalled whenever there is a on-off transition along any of the four principal directions. First notice that we start the system using the **boot** instruction and can see the results by displaying the output of the procedure "edge" using **||shw**. Memory locations are simply referred to by their number, in the above example we assume the image has already been input into location 1 (Procedures to input and output memory planes are described in Appendix B). The edge function is computed using a sequence of logical operators. The complement (negation) of the four neighbors are "or"ed together by shifting each along the horizontal and vertical directions. At this point the accumulator will be set if any of these neighbors is 0. To complete the computation we "and" the accumulator with the center pixel.

Notice that the algorithm is executed in time proportional to the size of the local neighborhood and is independent of the size of the array (providing the array to be processed fits within the limits of machine). On a more traditional serial computer this algorithm would require $O(kn^2)$ time, where n is the dimension of the array and k is the size of the neighborhood.

5.2. A Binary Region Filling Procedure

Region filling or determination of connectivity is another common use of an array processor. The goal of this procedure is to label all points that share a certain property, which without loss of generality we can denote as having the value 1, that are connected to a seed point. This algorithm is embodied in the following procedure.

```
procedure flood( input, label : memory );
 begin
  repeat
   ||mov(label,old);
   ||lda(label);
   ||or (label,-1, 0);
   ||or (label, 0, 1);
   ||or (label, 0,-1);
   ||or (label, 1, 0);
   ||and(input, 0, 0);
   ||sta(label);
   ||xor(old, 0, 0);
  until ||tst;
 end;
```

The "input" plane contains 1's wherever the image has the desired property. The "label" plane contains isolated "seed" points. The algorithm above dilates the "label" plane by setting points that neighbor, that is, are connected to labelled points and are part of the input region. The procedure terminates when no additional points can be added to the labelled region. This condition is tested for by "exclusive-or"ing the label plane before and after a single iteration and testing for the changes indicated by 1's in the result. As mentioned above this can be done with the function **||tst** since it will return false if a 1 exits anywhere in the accumulator. This algorithm, unlike the last example, cannot be formulated to run on array processor with $O(n^2)$ speed improvements over the conventional serial computer.

5.3. A Parallel Program for the Game of Life

As mentioned in the introduction, the accumulator contains more than a single bit to speed arithmetic operations. These arithmetic results can be further manipulated using the comparison operators. The use of these instructions is shown in the following example, which implements John Conway's game of Life.

```
procedure Life;
 const
  world = 1; eq2 = 2; eq3 = 3;
 function alive : boolean;
  begin
   alive := ||tst
  end;
 procedure generation;
  procedure neighbors;
   begin
     ||clr;
     ||adi(world,1,-1, 1);
     ||adi(world,1, 0, 1);
     ||adi(world,1, 1, 1);
     ||adi(world,1,-1, 0);
     ||adi(world,1, 1, 0);
     ||adi(world,1,-1,-1);
     ||adi(world,1, 0,-1);
     ||adi(world,1, 1,-1);
   end;
  begin
   neighbors;
   ||eq( 2 ); ||mov(status,eq2);
   ||eq( 3 ); ||mov(status,eq3);
   ||lda(world);
   ||and(eq2,0,0);
   ||or (eq3,0,0);
   ||sta(world);
  end;
 begin
  boot;
  ||shw(world);
  while alive do generation;
 end;
```

The rules of Life can be succinctly stated with the logical expression

w := (w and (neighbors = 3)) or (neighbors = 2)

where "w" is the world and "neighbors" is a function which counts the 8-adjacent neighbors that are alive. The procedure "neighbors" adds 1 to the accumulator when any of the surrounding neighbors is set, the maximum number of nearest neighbors is 8. The fate of the center cell depends on the actual number of neighbors at each point in the plane "world". Individual cell neighbor counts are determined by the **||eq** comparison operator. After each comparison the status plane contains the result and can be used like any other plane of memory to produce the new life forms in the next generation.

When computing functions which fetch from neighboring locations there exists an ambiguity at the border cells along the edges of the array. One possibility is to treat the two-dimensional array as an isolated square and when a memory reference extends past the edge either fetch a constant value or the value of the closest border cell. Another possibility, and what is done in this emulation, is to simply scroll or wrap around to the other side of the array. In this way the square is joined, left edge to right edge and top edge to bottom edge. Topologically, this makes the plane equivalent to a torus; so the creatures in the world of Life exist on a doughnut!

The above examples use the parallel array processor to compute nearest-neighbor, either 4- or 8-connected, functions over large data arrays. On most binary array processors the local neighborhood is "hard-wired", and in fact the *Clip4* allows these neighbors to all be fetched with a single instruction. Enhancing the neighborhood of a pixel by allowing fetches of data a different distances can be viewed in two ways. One interpretation of these additional structured fetches is that we are embedding a two-dimensional data structure into the array. Alternatively, we can view the process as adding interconnecting links between the individual processing elements. These additional links decrease the length of the best paths between pairs of processors improving the running time of several interesting algorithms. In the next two examples we take advantage of the abilities *(i)* to fetch data at varying distances and *(ii)* to locally freeze a processor to simulate two different enhanced array architectures; the first a two-dimensional shuffle network, the second a pyramid machine.

5.4. A Two-Dimensional Shuffle

Consider the following shuffle procedure whereby a recursive data rearrangement is done by selectively transferring data between distant processors by using a combination of the active register and power-of-2 shifts.

```
procedure mask( level : integer );
  const scratch = 15;
  { set a single position in a 2d array }
  procedure seed(x, y : integer; location : memory);
    external;
  { expand a pattern by doubling it }
  procedure double( delta : integer );
    begin
      ||lda( scratch );
      ||or ( scratch, delta, 0 );
      ||sta( scratch );
      ||or ( scratch, 0, delta );
      ||sta( scratch );
    end;
  begin
    { set the bit in the upper right hand corner }
    seed( 0, 0, scratch );
    for i := 1 to 7 do
     if i < level
      then {form block} double(pow2(i))
      else {fill array} double(pow2(i+1));
  end;

{copy from(sx,sy) to(dx,dy)}
procedure copy( sx, sy, dx, dy : integer );
  begin
    ||tra( scratch, active, -dx, -dy );
    ||asg( source, sx-dx, sy-dy );
  end;

procedure rotate-ccw( source : memory );
  var step, level : integer;
  begin
    for level := 1 to 9 do
      begin
          mask( level );
          step := pow(level-1);
          ||clr;
          copy( step, 0, 0, 0 );
          copy( step, step, step, 0 );
          copy( 0, step, step, step );
          copy( 0, 0, 0, step );
          ||sta( source );
      end;
  end;
```

At each level we apply the same 2x2 permutation. To rotate a 2x2 square each element is moved clockwise one position. (This same scheme works irregardless of the 2x2 permutation. Other examples are transpositions and reflections.) One pass of the permutation is done by sequentially mapping each subsquare from its original position to its destination position. The entire process is repeated log$n$ times. At each pass the size of the subsquares increase by a power-of-2 so successively more global transfers are occurring. Different block patterns are created by the procedure "mask" and loaded into the active register to control the pattern of transfers.

5.5. Parallel Pattern Matching

To conclude we will show a fragment of an interpreter for a pattern matching language used to implement the *transforms* used in a recognition cone. This interpreter combines many of the architectural features discussed above. The motivation and overall structure for this approach is outlined in Uhr[1972]. Briefly the recognition cone consists of a series of converging layers; associated with each is a set of transforms. An image enters the cone at the lowest and the largest level. Subsequent layers contain the results of applying pattern matches to the previous layer to yield a converged symbolic description of the image. The transforms themselves consist of two major parts, the first the *if* or *relational* part contains a series of *lookfors* which comprise the pattern. These are each tested for and if found, a weight is added, increasing the probability that this transform will succeed. Success is signalled if after all features are tested the accumulated probability exceeds a threshold. And finally when a transform succeeds features are output into the next layer.

The ideal architecture for such a system is a true hardware pyramid machine since the organization of the processors (and their memory) and the interconnections between neighboring processors mirror the above organization. Instead, the present implementation views the recognition cone as a special data structure embedded into the array processor. Each layer is stored in a single plane of memory. As the layer number increases we decrease the effective spatial resolution of the layer and increase the number of features in that layer. The first layer then consists of a 512 by 512 array of a single feature. The second layer is converged by 2 to a 256 by 256 array, but now to fill in the entire 512 by 512 plane we add 4 features to each cell (a cell corresponds to an active processor at that layer). This continues at each level, decreasing the resolution and active processors by 4 while increasing the number of features by a corresponding factor of 4.

To impose this pyramidal data structure on a simple array processor we use a set of variable resolution masks similar to that used in the above shuffle procedure. The masks are generated by the following procedure:

```
procedure masks;
 begin
  { The array mask points to a memory location }
  seed( 0, 0, mask[8] );
  for level := 8 downto 0 do
    double( mask[level], pow2(level) );
 end;
```

The procedure "double" is the same as that given above modified to take an source and a destination channel. The masks are what create a particular network topology. Notice the differences between these masks and the previous masks.

The key "match" procedure applies a transform at the present level in the recognition cone and outputs the results to the next highest level.

```
procedure match( template : transform );
 begin
  lower := template.level;
  upper := lower + 1;
  ||frz( mask[lower] );
  ||clr;
  for i := 1 to template.maxlookfor do
   with template.lookfor[i] do
      {feature and weight are fields in lookfor}
      begin
        {features are implicitly stored at (x,y)}
        x := (dx * pow2(lower))+(feature mod pow2(lower));
        y := (dy * pow2(lower))+(feature div pow2(lower));
        {array layer points plane location}
        ||adi( layer[lower], weight, x, y );
      end
  with template.implied do
   {feature and threshold are fields in implied}
   begin
      { check against threshold }
      ||ge( threshold );
      ||mov( status, operand );
      {convergence function, others are possible}
      ||frz( mask[upper] );
      ||clr;
      ||or( operand, 0, 0 );
      ||or( operand, pow2(lower), 0 );
      ||or( operand, pow2(lower), pow2(lower) );
      ||or( operand, 0, pow2(lower) );
      ||sta( operand );
      {features are implicitly stored}
      x := feature mod pow2(upper);
      y := feature div pow2(upper);
      { assign the result of the transform }
      ||tra( layer[upper], accumulator,  x,  y);
      ||asg( operand, 0, 0 );
      ||tra( accumulator, layer[upper], -x, -y);
   end
  end;
```

At each level in the pyramid there are a maximum of 2^{level} by 2^{level} different features. These are stored as a contiguous square. The various features are numbered up to the maximum; from their number they are implicitly assigned a particular (x,y) location. When testing for "lookfors" it is possible to test for a feature in a neighboring cell which requires that we shift in units of 2^{level}. Once all the "lookfors" have been tested, the accumulated result is thresholded and converged into the "implied" feature cell in the next higher level. This layer will be a factor of 4 smaller then the previous layer so the implied feature is in fact some function of potentially 4 different applications of the transform to the previous layer. For simplicity we "or" the four results together but more generally could be achieved by providing different convergence functions.

The hardware has limited somewhat the generality of the patterns that can be tested for. Ideally the output of a transform should contain multiple "implieds" each with different certainties. Although compact, this procedure illustrates many subtle aspects of this emulation and array processing in general. Despite this and other limitations this compact procedure illustrates most of the design features of the emulator. It require the ability to perform single bit logical as well as multiple bit arithmetic operations. It also requires the ability to locally inactivate processor execution and fetch and store data at varying distances.

6. Micro-instruction format

This section describes the implementation of the instruction interpreter on a Stanford Technology Corporation (STC) color display terminal. The configuration in the Image Processing Laboratory has 3 refresh channels (512x512 by 8-bits) and a graphics overlay (512x512 by 4-bits). In addition to the display hardware this terminal also has a pipelined arithmetic logic unit that operates at video rates. The system is programmed by sending commands to specific subunits using a dma transfer. Procedures exist to format subunit commands and execute them on the terminal. The instructions described above require setting most of the different subunits in the display terminal, analogous to microprogramming a computer. The following microinstruction record contains fields for all the necessary sub-operations required for a single instruction (a few miscellaneous instructions do not follow this format):

```
microinstruction =
 record
  pathenable : lutmap;
  pathfunction : lut;
  relativeoffset : scroll;
  operations : alu;
  go : feedback;
 end;
```

(Each type controls a specific subunit command. For the purposes of this discussion the details of these records are not important but can be found in the programmers' manual [Hanrahan and Schulz, 1981].) The fields perform the following functions: "Pathenable" sets the internal data bus to route data from the relevant refresh memory to the appropriate pipeline. There are three different pipelines which serve different functions in the emulation. The most important is the *transfer* pipe which routes data for binary operations; another is the *arithmetic* pipe which performs immediate mode substitution and the last is the *display* pipe which is used by the show register. Pipeline functions are determined by programming their *lookup tables* under control of the "pathfunction" field in the microinstruction. The alignment of bits stored in different positions in the 8-bit refresh buffers with the accumulator is performed with a lookup table. The "relativeoffset" command implements shifting of a memory plane relative to the processing element. This involves simply setting the scroll registers to the specified values. The "operation" field selects the opcode for the arithmetic logic unit. The different binary and unary instructions differ only in this field. Actually two opcodes are passed to the alu, one opcode for the active processors and another for the inactive processors. Generally the inactive processors are set to copy data from source to destination. The "go" routes data from the output of the alu back into the accumulator and begins the execution of each instruction. Execution is triggered when the *vertical retrace* signal starts the next scan (every 30 milleseconds). At this point the device performs the instruction, but at the same time, in parallel the host cpu is freed to perform other computations. Usually this time is spent formatting the next micro-coded instruction. This can be done within the allotted time so no refresh cycles are missed and the computation proceeds at the maximum rate.

During this implementation several problems were encountered that are particular to the STC. As much as possible these have been hidden from the user but at the expense of making certain operations comparatively more expensive than otherwise necessary. Most problems revolve around the fact that the refresh memory is organized to handle 8-bit data. Thus, different sequence of operations may be required depending on whether two memory location are contained in the same or different channels. For example, it is not possible to perform feedback operations with the same channel as input and output, if it is also being scrolled! [The solution was to create a temporary scratchpad register in one channel and perform single channel transfers by first loading the scratchpad register.] Originally we had also hoped to be able to reconfigure the memory to various widths, that is, instead of 24 1-bit locations we could have 6 4-bit locations, or 8 3-bit locations. The major difficulty with this scheme is that logical memory locations will in general fall across physical memory boundaries necessitating many awkward and time consuming shuffles when transferring data.

7. Summary and Conclusions

We have embedded an emulation of a binary array processor onto a pipelined display terminal. This allows us to code programs using powerful instructions which operate on a 1/4 million byte data stream. As a result for many simple image processing tasks a rather large speed up is achieved over traditional serial computer architectures. We have also begun to form a testbed for the development of parallel algorithms for the much more complicated tasks of image understanding and visual pattern recognition. Eventually, as more powerful arrays are designed and built it may be possible to achieve a "real" time implementation of these algorithms/processes.

8. Acknowledgements

I would like to thank Randy Schulz for programming the interface to the Stanford Technology Terminal. In particular, if he had not done a complete rewrite to include a structured command format this emulation could not have been written. I would also like to thank Len Uhr for numerous discussions about array processing and its potential in artificial intelligence.

This work was undertaken while the author had a University of Wisconsin Graduate Fellowship.

9. References

Batcher, K.E., Design of a massively parallel processor. *IEEE Transactions on Computers* 29:836-840, 1980.

Duff, M.J.B., Review of the CLIP image processing system., *Proc. AFIPS NCC*, pp 1055-1060, 1978.

Dyer, C.R., Augmented Cellular Automata for Image Analysis., Ph.D. Thesis, University of Maryland, 1979.

Dyer, C.R., Pyramid algorithms and machines., In: K. Preston and L. Uhr, Eds., *Multi-Computers and Image Processing* pp 409-420, 1982.

Hanrahan, P. and Schulz, R., STC programmers manual, (unpublished report), 1981.

Klette, R., Parallel operations on binary images., *Comp. Graphics and Image Processing* 14:281-292, 1980.

Reddaway, S.F., DAP - a flexible number cruncher., *Proc. 1978 L.A.S.L. Workshop on Vector and Parallel Processors*, pp 233-234, 1978.

Schmitt, L., The ICON perception laboratory user manual and reference guide., University of Wisconsin Computer Sciences Technical Report #421, 1981.

Stansbury, J., Clip4 and Dap simulators., (personal communication), 1980.

Stansbury, J. and Uhr, L., Extensions to parallel pascal., (personal communication), 1980.

Tanimoto, S.L., Programming techniques for hierarchial parallel image processors., In: K. Preston and L. Uhr, Eds., *Multi-Computers and Image Processing*, pp 412-429, 1982.

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

Uhr, L., A language for parallel processing of arrays, embedded in Pascal. In: M.J.B. Duff and S. Levialdi, Eds., *Languages and Architectures for Image Processing.*, London: Academic Press, pp 53-87, 1981.

Uhr, L., Thompson, M., and Lackey, J., A 2-layered SIMD/MIMD parallel "array/net"., *Proc. on Computer Architecture for Pattern Recognition and Image Data Base Management.*, IEEE Press, pp 209-216, 1981.

Appendix A — Table of Instructions

Transfer Operations

mov	- transfer	<src> <dst>
lda	- load the accumulator	<src>
sta	- store the accumulator	<dst>
frz	- load the active register	<src>
iff	- transfer status to active	
ffi	- transfer true to active	
tra	- transfer	<src> <dst> <dx> <dy>

Unary Operations

clr	- clear accumulator
set	- set accumulator
com	- complement accumulator
shr	- shift accumulator right
shl	- shift accumulator left

Binary Operations

asg	- assign	<src> <dx> <dy>
adi	- add value if true	<src> <value> <dx> <dy>
sbi	- subtract value if true	<src> <value> <dx> <dy>
and	- logical and	<src> <dx> <dy>
adn	- logically and the negation	<src> <dx> <dy>
or	- logical or	<src> <dx> <dy>
orn	- logically or the negation	<src> <dx> <dy>
xor	- logical exclusive or	<src> <dx> <dy>

Comparison Operations

Signed Comparisons

eq	- true if equal	<value>
ne	- true if not equal	<value>
ge	- true if greater or equal	<value>
gt	- true if greater	<value>
le	- true if less or equal	<value>
lt	- true if less	<value>

Unsigned Comparisons

ult	- true if less	<value>
ule	- true if less or equal	<value>
ugt	- true if greater	<value>
uge	- true if greater or equal	<value>

Miscellaneous Operations

boot	- initialize the emulator	
bits	- number of accumulator bits	<value>
tst	- true if all bits are zero	
shw	- show the operand	<src>

Appendix B — Usage

Currently, the above package of procedures can be executed from either a C or Pascal language program. All that is necessary is to include the appropriate type definitions and constants and then to link the appropriate libraries when loading.

When using C include the file "llc.h" somewhere near the head of your file. Because vertical bars cannot begin a legal identifier (and are already tokens in C (or metacharacter in the shell (and basically because fixing all these problems is too much work))) they are replaced by two "l"s, that is the letter el. Observing this rule load the program with:

```
% llc test.c
```

Similarly, when using Pascal include the file "llp.h" in the procedure section of the programs declarations. And then compile the program with:

```
% llp test.p
```

The biggest practical problem encountered is storing, retrieving and viewing image data. Data can be viewed with the program:

```
% llshw <memory>
```

which displays the appropriate bit plane just like the inline instruction.

Reading and writing data to the processor is not quite as easy. At this point the only way to do this is to dump an image into the processor memory and vice versa. (Note that the emulator does not change the contents of the refresh memories when starting or exiting. This approach is not as awkward as it might seem because the refresh memory is part of a display terminal and there are many programs which have been written to view, inspect and extract statistical information from images stored in the terminal. To do this however requires a memory map showing the correspondences between the array processor memory locations and the physical memory layout in the terminal.

Memory Map		
location	channel	bit-plane
0	3	7
1	3	6
2	3	5
3	3	4
4	3	3
5	3	2
6	3	1
7	3	0
8	2	7
9	2	6
10	2	5
11	2	4
12	2	3
13	2	2
14	2	1
15	2	0
16	1	7
17	1	6
18	1	5
19	1	4
20	1	3
21	1	2
22	1	1
23	1	0
active(-1)	graphics	0
status(-2)	graphics	1

Notice that memory locations are numbered from 0 through 23 and that the active and status locations are referred to as -1 and -2, respectively.