Adapt: Global Image Processing with the Split and Merge Model

Jon A. Webb

July 22, 1991

CMU-CS-91-129

(Revised to reflect new Adapt syntax 22 July 1991.)

Abstract

Adapt is a simple architecture-independent language for both local and global image processing operations based on the *split and merge* programming model. In the split and merge model, the image is divided into portions, each portion is processed independently, and then the separate results are combined. The split and merge model is examined, and is found capable of computing any image processing operation that can be computed in forward or reverse order over a data strcture. Moreover, the split and merge model is amenable to efficient implementation on a wide variety of parallel architectures. The Adapt language is described, and Adapt programming techniques are presented. Several algorithms for important image processing operations, including image warping, image connected components, and two-dimensional fast Fourier transform, are presented in detail. Implementations of Adapt on several parallel computers are described. The performance of several algorithms on the Sun, the Carnegie Mellon Warp machine, and the Carnegie Mellon Nectar computer architecture are presented and compared to hand code.

Last revision August 6, 1991 10:02 am.
This file in /afs/cs/project/adapt/tr/tr.book.

This material is based upon work supported by a grant from Kodak Corporation, and by the National Science Foundation under Grant MIPS-8920420. The Government has certain rights in this material.

Table of Contents

List of Figures

List of Programs

1. Introduction

Image processing and low-level computer vision have several characteristics that make them suitable for implementation with parallel computers:

- The operations are regular: the same operation is performed repeatedly across an image, over and over. This makes load-balancing easy.

- The programs follow a stereotypical form: the image is scanned in raster order, and some result is produced by incrementally modifying a data structure. This means that a good parallelizing technique is likely to apply to a large class of algorithms.

- The algorithms are not "deep": they are not complex algorithms that can be implemented only with great difficulty. This means that reprogramming is not difficult, making it possible to take advantage of new architectures given suitable programming tools.

- There is no "dusty deck" syndrome: while code libraries exist (`GIPSY` [8] , `SPIDER` [16]) reprogramming is common; it is not a requirement to be able to compile and execute huge `FORTRAN` programs written decades ago, as in scientific computing. This means that we do not have to confront the very difficult problem of automatically parallelizing existing programs.

- The algorithms are stable: the software development work needed for parallel processing need not occur in an environment of constantly changing algorithms.

- There is a natural source of parallelism: the image. Parallel algorithms can often be created by partitioning the image into sections. Images are large, giving a vast source of parallelism.

- The speedups that parallel processing offers are needed: in many robotics tasks, image processing is a critical bottleneck that affects the speed of a system. In remote sensing applications, the enormous size of images and their quantity make image processing time a limiting factor in what we can learn.

All of these characteristics mean that image processing and low-level computer vision are ripe for the application of parallel processing, providing we can solve the software problem. But the software problem is a serious limitation. Currently, it is not possible to automatically and efficiently parallelize serial computer code, especially when the target machine is a distributed memory parallel computer, which offers the most cost-effective means of increasing program speed through the application of parallel processing. Instead, we have a variety of computer languages, most of which are specialized to a particular parallel computer or at most a small class of parallel machines.

The current situation with parallel computers is analogous to the situation in the early 1960s with serial computers. Each had its own assembly language, which would efficiently run on one computer only. Emulators were available that could run that assembly language on other machines, but only at a great loss of speed. Computer users were locked into the computers of a particular manufacturer since they could not move their programs to another computer without enormous difficulty. While higher level languages existed, these too were idiosyncratic to particular machines (though far less so than the assembly languages). More important, using these high-level languages involved great sacrifices in efficiency; programmers still wrote their inner loop bodies in the assembly language for particular computers, trading portability for speed. It was impossible to build libraries of code that could be shared among users, greatly increasing the cost of programming.

We can predict the future of parallel processing by examining the history of serial computing. Higher-level languages were developed that hid the underlying machine architecture and implemented programming models for different applications areas: scientific computing (`FORTRAN`), and business (`COBOL`). Compiler technology improved with time, until the best compilers could compete with the best human programmers; high quality assembly language programming was no longer a valuable skill. Eventually, the quality of compiler technology fed back into processor design, and led to new simplified processor architectures such as RISC.

The higher level languages that were developed for serial computers incorporated a general programming model (the variable modification model) and could be applied to any problem, although they were specialized for particular areas. It was possible, but inconvenient, to use `COBOL` to do scientific computing, for example.

This level of generality is not yet possible for parallel computing. We simply do not know enough to create compilers that can present the programmer with a completely general programming model and efficiently map it onto a wide variety of parallel computers. However, we can do something almost as good: we can develop specialized languages for different applications areas, and those languages can be architecture independent.

We can define the levels of architecture independence from least independent to most:

1. **Processor Architecture.** The first is independence from the underlying processor architecture: for example, the number of registers, the presence or absence of a math coprocessor, and so on. This is the degree of architecture independence provided by conventional serial processor languages, which is totally inadequate for parallel processing.

2. **Array Size.** The second level is independence from the size of the processor array: the array topology must be the same, as well as the intercommunication scheme, but the array size can change. This is the degree of architecture independence provided by the Connection Machine's "virtual processors," as well as various languages for transputer arrays, etc. This kind of independence is extremely useful—it allows the development of code on a small array of nodes, and production runs on large arrays, as well as upgrades of processor arrays to larger systems without changing software—but it does not allow the movement of code from one architecture to another.

3. **Array Topology.** The third level supports changes in the underlying processor array as well as its topology. For example, languages for transputer arrays support this: processes can be placed on whatever processors are convenient, and the programming system takes care of managing communication using whatever array topology is available. This, again, is useful, and it does provide some degree of independence from a particular architecture.

4. **Interprocessor Communication.** A fourth level completely hides interprocessor communication. For example, Linda manages interprocessor communication by supporting a "tuple" model in which processes place tuples in a common database and read them from their. This hides all details of process placement and communication from the programmer.

5. **Process Management.** Finally, we reach the highest level of architecture independence: this hides the existence of multiple processes from the programmer, and all communication. The programmer writes his or her program in a model that does not make any mention of multiple processes, and the compiler takes care of splitting the work into multiple processes, allocation of data, and so on.

It is this *last* model that will be discussed here. We achieve this remarkable degree of architecture independence by giving up on the notion that the programming language has to be general purpose. Instead, we develop a language that is targeted for a specific applications area.

This was done previously with the Apply language [7] . Apply was targeted to local, low-level vision operations like edge detection, smoothing, point operations, and so on. The programmer wrote his or her program as the "inner loop" of the local operation: the compiler then managed allocation of the image to different processors and all communication. The degree of architecture independence was remarkable, as can be seen from the wide range of architectures that have Apply compilers:

- Warp (10 processor one-dimensional systolic array) [19] , iWarp (4-1024 processor two-dimensional systolic array) [3] , FT Warp (two-dimensional systolic array with fault-tolerance) [13] .

- Sun and other UNIX architectures (serial architecture) [19] .

- Carnegie Mellon SLAP (one processor per column in an image—e.g., 512 processors for a 512×512 image.) [6]

- University of Massachusetts Image Understanding Architecture (one processor per pixel in the largest implementation) [15] .

- Meiko Computing Surface (transputer-based, reconfigurable) [4] .

• Hughes HBA, a video bus-based machine [18] [19]

Adapt addresses the issue of architecture independence in the larger context of both local and global image processing operations: that is, operations such as histogram, Hough transform, and image warping will be included as well as Apply-type operations. Architecture independence will be preserved by hiding from the programmer all details of the existence of multiple processes, allocation of data to processes, and inter-process communication, as well as anything having to do with multiple processors. This will be done by implementing a new programming model in Adapt, as will now be explained.

2. The Split and Merge Programming Model

In the split and merge model an operation is computed by splitting the data into parts, computing a result on each part independently, and then merging the two results.

Split and merge is a specialization of a more general algorithm design technique, called *divide and conquer*, that is used both in serial and parallel computing. In serial computing, the divide step is usually applied recursively until a trivial problem is reached; all the real work is in the merge step. This can lead to efficient algorithms, for example for multiplying long integers, sorting, and search [1] . Divide and conquer has been proposed before as a model for implementing image processing operations on parallel computers [17]

In divide and conquer, the data set is usually partitioned in a special way; for example, in Quicksort the dataset is split according to whether the data are larger or smaller than a given element [9] . In split and merge, the partitioning is done blindly, according to data position; this is because many parallel computers have facilities for dividing data according to position, while few have facilities for data dependent division.

Split and merge is an extremely common model for parallel computing of global operations. I will now consider two important characteristics of split and merge: what class of computations can be done with it, and what kinds of distributed memory parallel computers are suited for the use of split and merge.

2.1 The Class of Computations of Split and Merge

In the split and merge model, we assume that a data structure is being processed, and the computation is split up over this data structure. Without loss of generality, we can assume that this data structure is a linear sequence $[a_1, \ldots, a_n]$. (Other data structures, for example an image, can be mapped onto such a sequence by processing them in a particular order, for example raster order). We divide this sequence into sections $[a_1, \ldots, a_{i_1}]$, $[a_{i_1+1}, \ldots, a_{i_2}]$, $\ldots$, $[a_{i_m+1}, \ldots, a_n]$, process each sequence independently, and then *combine* the partial results. It will now be shown that the class of computations that can be done with such a model is exactly the class of computations that can be done in forward or reverse order over $[a_1, \ldots, a_n]$.

First we show that any computation that can be done in this way can be done in forward or reverse order. If we can do a computation in this way, then we can do it by dividing the input sequence down into n sequences of length 1. After doing this, we process each sequence independently, and then *combine* them either from the beginning to the end, or from the end to the beginning. We can do the combining in either way, so that we can process the entire sequence in either forward or reverse order.

Now we show that if we can process the sequence in either forward or reverse order, then we can process it with the split and merge model. To do this, we must employ a little more mathematical jargon than we have used up until now. Let the serial function, which processes the data structure one element at a time in order, be called A. We will write $A[a_i, \ldots, a_j]$ for $A(a_j, A(a_{j-1}, \ldots, A(a_i)))$ —that is, the serial function applied to all the elements in the sequence, starting at the first and working forward to the *last*. We will write A^R for the reversed version of the serial function. The function that *combine*s the independently computed results from two subsequences will be called C; the point of this part of the proof is to define C so that $C(A[a_i, \ldots, a_j], A[a_{j+1}, \ldots, a_k]) = A[a_i, \ldots, a_k]$ for any two subsequences $[a_i, \ldots, a_j]$ and $[a_{j+1}, \ldots, a_k]$.

We do this by “guessing” a sequence $[b_1, \ldots, b_l]$ such that $A[b_1, \ldots, b_l] = A[a_i, \ldots, a_j]$. (We must guess this sequence because we do not, at this point, have the actual values of the $[a_i, \ldots, a_j]$ available to us; all we have is $A[a_i, \ldots, a_j]$. We can always guess the sequence by simply enumerating all possible sequences and testing which of them gives the same result as $[a_i, \ldots, a_j]$. Therefore, this step is computable.) We then define

$$C(A[a_i, \ldots, a_j], A[a_{j+1}, \ldots, a_k]) = A^R(b_1, A^R(b_2, \ldots, A^R(b_l, A[a_{j+1}, \ldots, a_k])))$$

We will now show that C satisfies the requirement above, using the reversibility of A:

$$\begin{aligned}
A[a_i, \ldots, a_k] &= A(a_k, A(a_{k-1}, \ldots, A(a_i))) \\
&= A(a_k, A(a_{k-1}, \ldots, A(a_{j+1}, A(a_j, A(a_{j-1}, \ldots, A(a_i)))))) \\
&= A(a_k, A(a_{k-1}, \ldots, A(a_{j+1}, A(b_l, A(b_{l-1}, \ldots, A(b_1)))))) \\
&= A^R(b_1, A^R(b_2, \ldots, A^R(b_l, A^R(a_{j+1}, A^R(a_{j+2}, \ldots, A^R(a_k)))))) \\
&= A^R(b_1, A^R(b_2, \ldots, A^R(b_l, A(a_k, A(a_{k-1}, \ldots, A(a_{j+1})))))) \\
&= A^R(b_1, A^R(b_2, \ldots, A^R(b_l, A[a_{j+1}, \ldots, a_k]))) \\
&= C(A[a_i, \ldots, a_j], A[a_{j+1}, \ldots, a_k])
\end{aligned}$$

This completes the proof.

The proof confirms a natural intuition about the split and merge model: it is very general, but it cannot be applied when there is a dependency in the serial computation that “matters.” For if the serial computation has such a dependency, it will be impossible to compute the same result while taking the data in reverse order. Some examples of this from image processing are error-diffusion halftoning, in which an error is propagated through an image in a particular order, and some region segmentation algorithms in which regions are merged while scanning the image in a particular order. However, this is not a fatal problem for split and merge; in many cases, equivalent algorithms can be found that eliminate the dependency while producing the same or even better results—recent research has shown this for halftoning, for example [20] .

The generality of the split and merge model is encouraging; the *next* question to consider is what class of parallel computers can efficiently implement this model. We now consider this question.

2.2 The Class of Parallel Machines of Split and Merge

The split and merge model can clearly *be implemented* on almost any parallel computer. What we mean by “the class of parallel machines of split and merge” is the class of parallel machines where the split and merge model will run efficiently; efficiently, that is, as compared with the same program running on a serial computer—a program that processes the sequence of data as a whole without ever combining results from independent subsequences.

2.2.1 Processor Number

We consider what size processor arrays can implement the split and merge model. To do this, we will first develop formulae for execution time under split and merge.

In the split and merge model we take an existing serial computation function A and augment it with a combining function C that allows results to be computed independently and then *combine*d. Clearly, any loss of efficiency in the parallel computation comes from the function C (and from data transmission between processors, which we can include in C.)

The efficiency of the computation depends on

1. The number of processors P.

2. The method of *combining* results produced by different processors: they may be *combine*d serially or in a binary tree, for example.

3. The time needed for combining two independently produced results as compared with the time needed to produce them. To simplify the discussion, we will assume that the time needed to process a single datum with A is constant (call it a), as is the time needed to *combine* two results with C (call it c). We will write N for the number of data to be processed.

For example, on a serial computer the execution time is $T_s = aN$. On a parallel computer where results are *combine*d sequentially, the execution time is $T_1 = a(N/P) + cP$. If we *combine* results in a binary tree, the execution time is $T_b = a(N/P) + c\log P$. If we do the *combine*s as we might on a two-dimensional mesh, first combining all rows into one column and then combining that column, the execution time is $T_2 = a(N/P) + 2c\sqrt{P}$.

Typical execution time behavior under the split and merge model is shown in Figure 1. Initially, execution time decreases linearly as we add more processors. The execution time curve passes through a "knee" where the decrease in execution time starts to taper off as the overhead of more and more combining operations starts to build up. Eventually, the combining overhead equals the speedup from using more processors, and the curve reaches a minimum. Thereafter, adding more processors actually increases execution time.

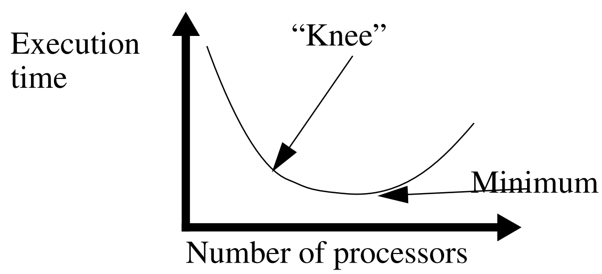

Figure 1. Execution time with the split and merge method

We clearly do not want to use so many processors that the decrease in execution time as we add another processor is near zero, as it is near the minimum—doing so is not cost-effective. Instead, we would rather be near the knee of the curve. The precise location of this point can be found [5] . It depends only on how partial results are *combine*d and on the number $\alpha = (aN)/c$. For sequential combining, the knee occurs at $\sqrt{\alpha/3}$; with combining as on a two-dimensional mesh, the knee occurs at $(\alpha/4)^{2/3}$. With binary tree combining, the knee occurs at the P that satisfies $P = (\alpha\log 2)/(\log P + 2)$.

In image processing, the number α depends on the number of pixels in the image, which ranges from the tens of thousands in industrial vision applications to hundreds of millions in reconnaissance applications, and the ratio of execution time of the A and C functions a/c. This ratio varies, but is usually quite small: less than 0.01. For example, in histogram the A function increments a histogram element, while the C function must add two histograms together, so this ratio is 0.004 or less. In the connected components algorithm discussed later, the ratio is less than 0.001.

Table 1 gives the optimal processor numbers for two image sizes and three algorithms under various combining methods. We see from this table that the number of processors that can be effectively used under the split and merge model is on the order of hundreds, not thousands, except with enormous images and algorithms in which the combining function is trivial.

Algorithm	$\frac{a}{c}$	Image Size	α	Combining Method: Serial	Two-Dimensional	Binary Tree
Image Sum	1	512×512	260,000	300	160	16,000
		$10K \times 10K$	100,000,000	5800	85,000	4,000,000
Histogram	0.004	512×512	1000	18	40	106
		$10K \times 10K$	390,000	360	2100	22,000
Connected Components	0.001	512×512	260	9	16	32
		$10K \times 10K$	10,000	41	120	430

Table 1. Optimal Processors for Split and Merge Model

2.2.2 Memory size

The split and merge model *combine*s the results from different processors into a single final result. As implemented in Adapt, there is no ordering restriction placed on the accesses to the data structures being combined. Therefore, for efficient implementation, the data structures being combined must be placed in a memory that can be efficiently accessed by a processor, such as the processor's local memory or a common shared memory. As a result, in the distributed memory case, the memory at each processor must be large enough to contain this final result. This means that split and merge processor arrays must have relatively large memories at each processor, or must share access to a relatively large common memory.

For example, to compute the histogram operation with split and merge, two histograms must be stored at each processor during the combining phase. This means at least 512 words of memory must be available. For the union-find based connected components algorithm described later, memory must be large enough to store two copies of the equivalence table; this means at least 20K words in typical applications. In an implementation of image warping using split and merge, two output images must be stored at each processor, meaning about 1/2 MB of memory must be available in a typical application.

2.2.3 Control

In the split and merge model, computation is done independently at each processor, and then these results are *combine*d. This creates opportunities for the exploitation of either SIMD or MIMD control. With SIMD, control for the independent computations in the first phase can be factored out, with some decrease in hardware cost. With MIMD, each processor must maintain its own instruction memory and controller, but it can take advantage of independent branching in conditional operations within the first phase, and reduce overall computation time correspondingly.

2.2.4 Interprocessor Bandwidth

Interprocessor bandwidth is critical in the split and merge model. The numbers in Table 1 indicate a severe limitation in the number of processors that can be effectively exploited, even assuming favorable ratios like 0.001 between the A and C functions. With low interprocessor bandwidth, this ratio increases rapidly, because results must be exchanged between processors during the combining phase. Because the combining phase is so critical for use of more than a few processors, interprocessor bandwidth must be very high.

2.2.5 Interprocessor Latency

Interprocessor latency is not as critical for the split and merge model as it is, say, in systolic applications. The input data can be distributed to all processors, local computation can be done independently, and then the combining phase can be done with its exchange of variables between processors. Nothing in this implies that one processor must wait repeatedly for another to complete before it can do its computation, as in a systolic computation. It is quite possible for the split and merge model to be implemented efficiently on a processor with relatively high interprocessor latency, so long as interprocessor bandwidth is high.

2.2.6 Array Topology

We have discussed briefly three different ways of implementing the combining operation, according to the underlying array topology: serial combining on a one-dimensional processor, two-level merging on a two-dimensional processor, and binary tree merging on a processor with hypercube or similar high-dimension interconnections. There are other possibilities. The important thing for split and merge is not the underlying topology, but rather how that topology interacts with the other characteristics, such as processor number and interprocessor bandwidth. Low-dimension interconnections can be made with high bandwidth and low latency, making it possible to use more processors because of a faster combining function; but high-dimension interconnections make it possible to implement a more efficient combining structure, again making it possible to use more processors. The formulae developed in this section for the usable processors given different values of α and different combining structures allow precise statements about the interaction of these characteristics, given specific processors, array topologies, and interprocessor bandwidths.

2.3 Summary

In this section we have developed a sense of what kinds of parallel computers can implement the split and merge model efficiently: they are those with tens or hundreds of processors, with fairly large memories, and with high interprocessor bandwidths. They can have various array topologies and either SIMD or MIMD control.

We now discuss an implementation of the split and merge model in a programming language for global image processing operations, the Adapt programming language.

3. The Adapt Language

The split and merge method is implemented in the Adapt language, which is a simple language for architecture independent local and global image processing. There are two kinds of parameters processed by Adapt programs: *global variables* and *images*. Global variables are broadcast to all processors (if input) or processed separately at different processors and then *combine*d (if output). Images are split among processors. To allow this, the Adapt programmer must divide an image processing function into four parts:

First An initialization function, which may be run at the beginning of any row of the image.

Next A raster-order function, which is applied in raster-order across the image (wrapped around at the borders of the image). Each execution of the raster-order function is guaranteed to be preceded either by another execution of the raster-order function (for the previous pixel), or by the initialization function at the beginning of a row. As in Apply, images processed by in the *Next* function are referenced relative to the current pixel position.

Combine A merging function, which *combine*s the outputs of any two image regions to produce an output for the concatenation of the two regions. To make programming easier, the combining function will be applied only to adjacent "swaths" (groups of consecutive rows) of the image [18] . The *Combine* function can reference any variable used elsewhere in the program. Its output is also expressed in terms of these variables. Within the *Combine* function, variables preceded by an underscore refer to the values of variables in the lower image region, and variables not preceded by an underscore refer to values from the upper image region. The *Combine* function result is to assign to variables not preceded by an underscore the correct values for the merged region.

Last A termination function, which is applied once after the output of the entire image is computed.

It is not necessary to have all these parts in an Adapt program; only the *Next* function is required if only images or images and global input variables are being processed, while the *Next* and *Combine* functions must be defined if images and global output variables are processed.

Because Adapt allows the programmer to reuse results from the previous pixel in the *Next* function and restricts the *First* function to be run only at the beginning of a row, Adapt is fundamentally oriented towards partitioning the image by rows. The reason, as discussed in Section 2, is that split and merge does not make enough parallelism available to use more than a few hundred processors, so that finer partitioning than by rows is useless. Moreover, giving the programmer the convenience of processing row-wise is a great advantage in making programs easier to write and more efficient.

Adapt uses many concepts from Apply to make it easier to write Adapt programs [7] . In particular, the programmer defines a rectangular window in each input image around the current pixel that can be accessed by the *Next* function; the elements in this window are indexed just as they are in Apply, relative to the current pixel position. Adapt also uses Ada-like syntax, as in Apply, and allows the programmer to describe subsampling and image expansion operations. Thus, Adapt includes all of the Apply concepts while adding the ability to reuse intermediate results generated at a previous pixel and the ability to merge results from regions processed separately.

There are four types of parameters in Adapt programs: input and output *images*, which are iterated over by the *Next* function (in either raster order or transposed raster order), and other input and output parameters, which are called *global*. Input global parameters are "broadcast" to all instances of *Next* (they are also available during *First*, *Combine*, and *Last*); their values cannot be changed. Output global parameters are given values as a result of processing in *First* and *Next*; values from two different regions of the image are combined within the *Combine* function; and the *Last* function can also modify the values.

A simple Adapt program for histogram, shown in Program 1, illustrates these concepts. The procedure header (lines 1 through 2) declares that there is one input image, and one output global variable, the histogram. The final histogram resulting from this program is a frequency of each pixel, rather than a pixel count, so it is declared to be a floating point value. An internal global program variable, `count`, is declared in line 4; it is available to all sections of the program. The *First* function (lines 5 through 10) zeroes an integer histogram called `count that is` used internally in the program to avoid expensive floating point computation. The *Next* function (lines 11 through 13) increments the `count` element for the current pixel. The *Combine* function (lines 14 through 19) adds the `count` arrays from the upper and lower image regions and stores the result into the histogram for the merged region; note the presence of the underscore (line 17) before the version of `count` that comes from the lower region. Finally, the *Last* function (lines 20 through 30) adds all the histogram elements together to get the total number of pixels in the image, and divides the frequency count table by this to get the final frequency count.

```
   procedure histogram(
1   im     : in image byte,
2   hist: out array(0..255) of float)
3  is
4   count: array(0..255) of integer;
5  first
6  begin
7   for i in 0..255 loop
8     count(i) := 0;
9   end loop;
10 end first;
11 next begin
12  count(im) := count(im) + 1;
13 end next;

14 combine
15 begin
16  for i in 0..255 loop
17    count(i) := count(i) + _count(i);
18  end loop;
19 end combine;

20 last
21  pixels: integer;
22 begin
23  pixels := 0;
24  for i in 0..255 loop
25    pixels := pixels + count(i);
26  end loop;
27  for i in 0..255 loop
      hist(i) := float(count(i))/
28 pixels;
29  end loop;
30 end last;
31 end histogram;
```

Program 1. Adapt Histogram.

The exact semantics of Adapt can be understood in terms of invariants applied over the input and output images. Let $I[(r_1, c_1), (r_2, c_2)]$ be an invariant that is true for the image region starting at pixel position (r_1, c_1) and up to but not including pixel (r_2, c_2). An invariant can be any statement involving any of the global program variables or parameters; it cannot involve any of the program variables declared in the individual program sections (like `i` and `pixels` in Program 1.) For example, an informal statement of the invariant for might be "`count(x)` is the number of pixels having value `x`." Keeping this in mind, the formal semantics of Adapt programs are:*

$$\{\mathtt{col} = 0\}\ \mathit{First}\ \{I[(\mathtt{row}, 0), (\mathtt{row}, 0)]\}$$

$$\{I[(\alpha, 0), (\beta, \gamma)] \wedge \mathtt{row} = \beta \wedge \mathtt{col} = \gamma\}\ \mathit{Next}\ \{I[(\alpha, 0), (\beta', \gamma')]\}$$

$$\{I[(\alpha, 0), (\beta, 0)] \wedge \_I[(\beta, 0), (\gamma, 0)]\}\ \mathit{Combine}\ \{I[(\alpha, 0), (\gamma, 0)]\}$$

$$\{I[(0, 0), (\mathtt{Rows} + 1, 0)]\}\ \mathit{Last}\ \{\text{Result}\}$$

In English, the semantics of Adapt are: The *First* function establishes the invariant for the null region starting and ending at the first pixel of the current row. Given that the invariant is true from some starting row up until the current pixel, the *Next* function extends it by one pixel. The *Combine* function takes two versions of global program variables,

*The names `row` and `col` refer to the current row and column of the image; the image has `Rows` rows and `Cols` columns. These names can be used as reserved variables in Adapt programs, although `Rows` and `Cols` cannot be used in declaring the size of an array. (β',γ') in the definition of *Next* refers to the pixel following the pixel at (β,γ) in raster order—that is, the pixel to the right if there is one, otherwise the first pixel of the next row. $\_I$ in the semantics of *Combine* refers to a version of *I* in which all global variable names have been preceded by underscores; in the Adapt program, these variables refer to the global variable resulting from the lower of the two regions being combined.

from two adjacent regions of rows, and makes the invariant true for the two regions merged. Given that the invariant is true for the entire image, the *Last* function establishes the final result of the program.

We can illustrate these semantics by once again examining the histogram program. The invariant is "`count(x)` is the number of pixels having value `x`" To establish this for a null region, `count(x)` must be set to 0 for all `x`; the *First* function does this. The *Next* function extends the invariant by incrementing the appropriate element of `count`. The *Combine* function must add the two versions of `count` together to keep the invariant true; and *Last* uses `count` to compute the final result of the program, the pixel frequency distribution.

We now illustrate the features of the Adapt language and its use by presenting several example Adapt programs.

3.1 Adapt Minimum Bounding Rectangle

An Adapt program for minimum bounding rectangle is given in Program 2. The invariant for this program is "Either `tr(x)=-1` or `(tr(x),tc(x))` is the top left pixel with value `x`, and `(br(x), bc(x))` is the bottom right pixel with value `x`." We make this invariant true trivially in the *First* function by setting all `tr(x)` to `-1` (lines 8 through 10). We then extend the invariant by one pixel in the *Next* function (lines 12 through 27). Note that we take advantage of raster-order processing in this section, by setting the top row and column of the rectangle without testing in lines 14 through 17 and by always extending the bottom row of the rectangle in line 25. There is no need to test because the first pixel we encounter in a raster-order scan must be the top left corner, and the last pixel seen is on the bottom row. The *Combine* function merges two rectangles; if there is none in the top region, the bottom rectangle applies; otherwise the bottom row and column come from the bottom rectangle, and the top column is the minimum of the columns of the two rectangles.

```
1   procedure mbr(im : in image byte,
2     tr : out array(0..255) of integer,
3     tc : out array(0..255) of integer,
4     br : out array(0..255) of integer,
      bc : out array(0..255) of integer)
5   is

6   first
7   begin
8     for i in 0..255 loop
9       tr(i) := -1;
10    end loop;
11  end first;

12  next begin
13    if im /= 0 then
14      if tr(im) = -1 then
15        tr(im) := row;
16        tc(im) := col;
17        bc(im) := col;
18      else if col > bc(im) then
19          bc(im) := col;
20        end if;
21        if col < tc(im) then
22          tc(im) := col;
23        end if;
24      end if;
25      br(im) := row;
26    end if;
27  end next;

28  combine
29  begin
30    for i in 0..255 loop
31      if tr(i) = -1 then
32        tr(i) := _tr(i);
33        tc(i) := _tc(i);
34        br(i) := _br(i);
35        bc(i) := _bc(i);
36      elsif _tr(i) /= -1 then
37          br(i) := _br(i);
38          bc(i) := _bc(i);
39          if _tc(i) < tc(i) then
40            tc(i) := _tc(i);
41          end if;
42      end if;
43    end loop;
44  end combine;

45  end mbr;
```

Program 2. Adapt Minimum Bounding Rectangle

3.2 Fast Uniform Convolution

A fast algorithm exists for $M \times N$ uniform convolution (that is, convolution with a rectangular filter with constant values). An array of sums of the last M elements of each column is maintained. The sum of the last N of these is also maintained—this is the “box sum,” which becomes the convolution result when multiplied by a constant. The image is scanned in raster order, and the next pixel is processed as shown in Figure 2. First the column sum for the next column is updated; the image pixel from the row *M* previous is subtracted, and the current image pixel is added. Next the box sum is updated: the column sum from the column *N* to the left is subtracted, and the current column sum is added. This results in uniform convolution being calculated in only four addition operations, instead of the MN needed if the box sum were calculated independently at each pixel.

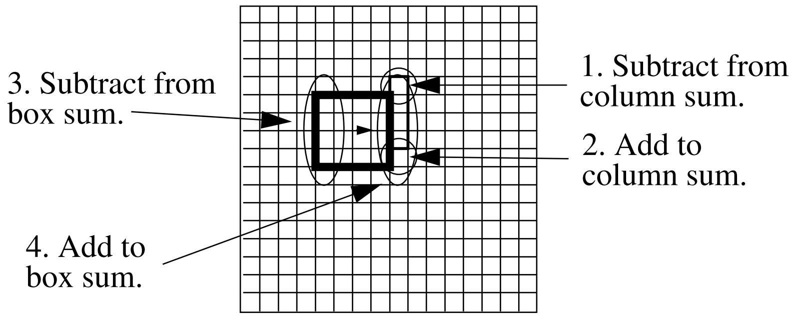

Figure 2. Uniform Convolution

If we use the array `colsums` to store the column sums and `boxsum` to store the box sum, and `im` is the image, then a natural invariant for uniform convolution is:

For region $[\,(\alpha, 0)\,,\,(\beta, \gamma)\,]$:

1. $c \le \gamma + N/2 \Rightarrow \left(\texttt{colsums}\,(c) = \sum_{r = \beta - M/2}^{\beta + M/2} \texttt{im}\,(r, c) \right)$
2. $c > \gamma + N/2 \Rightarrow \left(\texttt{colsums}\,(c) = \sum_{r = \beta - M/2 - 1}^{\beta + M/2 - 1} \texttt{im}\,(r, c) \right)$
3. $\texttt{boxsum} = \sum_{c = \gamma - N/2}^{\gamma + N/2} \texttt{colsums}\,(c)$

The problem with this invariant is that it cannot be established efficiently. The natural place to establish Parts 1 and 2 is in the *First* function; but the entire image cannot be accessed in this function, only a window around the first pixel in the row can be read, so Part 2 cannot be established. Hence, we must instead establish the invariant in the *Next* function, which leads to MN add operations per pixel.

However, we can overcome this problem by modifying the invariant and establishing Part 2 while the *Next* function scans the first row. We simply establish Parts 1 and 3 of the invariant above in the *First* function, and note which row we are processing; Part 2 of the invariant is established as we process this row.

The invariant then becomes:

$$1.\ c \le \gamma + N/2 \Rightarrow \left(\texttt{colsums}\,(c) = \sum_{r = \beta - M/2}^{\beta + M/2} \texttt{im}\,(r, c) \right)$$

$$2.\ \texttt{startrow} = \alpha$$

3. $\beta \ne \texttt{startrow} \wedge i > \gamma \Rightarrow \left(\texttt{colsums}\,(c) = \sum_{r = \beta - M/2 - 1}^{\beta + M/2 - 1} \texttt{im}\,(r, c) \right)$
4. $\texttt{boxsum} = \sum_{c = \gamma - N/2}^{\gamma + N/2} \texttt{colsums}\,(c)$

The Adapt program for uniform convolution is shown in Program 3. Lines 10 through 18 establish all four parts of the invariant for the first column in a region. Lines 23 through 26 establish Part 1 for columns in the first row, while line 28 establishes this step for other rows. Lines 31 through 34 establish Part 3 for the first column in a row (unless already established by *First*.) The update to `boxsum` for the next pixel is in line 36. The total number of adds per pixel is only four, with *N* extra adds required for each row, and *MN* extra adds per image. On a parallel computer where each processor takes one swath of rows of the image, the execution time in add operations is $MN + \text{PRows}\ (N + 4 \times \texttt{Cols})$, where each processor is given a swath of PRows rows.

```
   procedure uconv(
     inimg   : in image
       array(-M/2-1..M/2, -N/2..N/2)
1      of byte border mirrored,
2    weight  : in float,
3    outimg  : out image byte)
4  is
     colsums : array(-N/2..Cols+N/2)
5                      of integer;
6    boxsum    : float;
7    startrow : integer;
8  first
9  begin
10   startrow := row;
11   boxsum := 0;
12   for c in -N/2..N/2 loop
13     colsums(c) := 0;
14     for r in -M/2..M/2 loop
         colsums(c) := colsums(c) +
15                   inimg(r,c);
16     end loop;
       boxsum := boxsum
17 +colsums(c);
18   end loop;
19 end first;

20 next
21 begin
22   if startrow = row then
23     colsum(col+N/2) := 0;
24     for i in -M/2..M/2 loop
         colsums(col+N/2) := colsums(col+N/2) +
25                            inimg(i,N/2);
26     end loop;
27   else
       colsums(col+N/2) := colsums(col+N/2)
28       -inimg(-M/2-1, N/2)+inimg(M/2,N/2);
29   end if;
30   if col = 0 and row > startrow then
31     boxsum := 0;
32     for c in -N/2..N/2 loop
33       boxsum := boxsum + colsums(c);
34     end loop;
35   else
       boxsum := boxsum - colsums(col-N/2-1)
36                     + colsums(col+N/2);
37   end if;
38   outimg := boxsum * weight;
39 end next;
40 end uconv;
```

Program 3. Adapt Uniform Convolution.

3.3 Adapt Run-Length Encoding

Run-length encoding is a simple image compression operation. An image is scanned in raster order, and each run of identically valued pixels of length *n* and value *g* is changed into an ordered pair $(g, n-1)$ (the length is represented as $n-1$ instead of *n* because any run is of length at least 1, and this allows the representation of runs one pixel longer than would otherwise be the case.)

An invariant for run-length encoding is:

> After processing pixel k in raster order, the run-length encoded representation is $\{(g_1, n_1), (g_2, n_2), \ldots, (g_m, n_m)\}$ if, and only if, the image is $g_1^{n_1+1} g_2^{n_2+1} \ldots g_m^{n_m+1}$, where a^b represents a sequence of b a's, and $g_i \neq g_{i+1}$ for any i.

An Adapt program that establishes and preserves this invariant is given in Program 4. The invariant is established for the null region in line 3. Line 6 checks to see if the invariant can be extended simply by incrementing `n(m)`; if not, `m` is incremented and a new run is started. Merging two runs is complicated because the end of one run may be merged with the beginning of the next; line 19 check for this case. The second run is copied onto the end of the first in lines 32 through 37.

```
   procedure rle1(
     im : in image byte,
     g   :out array(0..MAXSIZE-1) of byte,
     n   :out array(0..MAXSIZE-1) of byte,
 1   m   : out integer) is

 2 first begin
 3   m := -1;
 4 end first;

 5 next begin
     if m = -1 or im \= g(m) or
 6       n(m) = 255 then
 7     m := m + 1;
 8     g(m) := im;
 9     n(m) := 0;
10   else
11      n(m) := n(m) + 1;
12   end if;
13 end next;

14 combine
15   pos, sum : integer;
16 begin
17   if m >= 0 and _m >= 0 then
18     pos := 0;
       while g(m) = _g(pos) and
19           pos \= _m+1 loop
20       sum := n(m) + n(pos) + 2;
21       if sum <= 256 then
22         n(m) := sum - 1;
23         pos := pos + 1;
24       else
25         n(m) := 255;
26         m := m + 1;
27         g(m) := g(m-1);
28         n(m) := pos - 256;
29         pos := pos + 1;
30       end if;
31     end loop;
32     while pos \= _m+1 loop
33       m := m + 1;
34       g(m) := _g(pos);
35       n(m) := _n(pos);
36       pos := pos + 1;
37     end loop;
38   end combine;

39   end rle1;
```

Program 4. Run-length encoding, first version.

A problem with this program, or rather the data structure that it produces, is that it is difficult to invert in parallel. It is not possible to determine the value of a pixel in the run-length encoded representation without stepping through all the data in the representation up to that point. We therefore cannot do run-length decoding in parallel. However, a small change to the data structure allows this.

All we have to do is maintain a table of the starting positions of each row of data in the original image. To do this, we will have to split runs of data across the image boundaries. Doing so is unlikely to decrease the compression by much, since pixels at opposite borders of the image are likely to be unrelated. We will maintain an array, called `Start`, that indicates the position in the run-length encoding where each row begins. To define such an array, runs must be split across image borders. An invariant for such a program is the invariant above combined with:

The *i*th row in the region starts at position $n_1 + n_2 + \ldots + n_{\texttt{Start(i)} - 1}$.

The modified program is shown in Program 5. Note that it is shorter and simpler than Program 4. The simplification comes because we no longer need to merge runs in the *Combine* section, because runs are not merged across image borders. Note also that we simply append the bottom region's `Start` array in the *Combine* section, instead of noting the actual beginning row of the region and storing the row offsets in the appropriate place in `Start` because we know that eventually, *Combine* will be applied with the top region containing the first row of the image, so that the append will cause the entire array to be filled from the beginning.

Given this program, with its invariant, the run-length decoding program is simple, and is shown in Program 6.

```
procedure rle2(
  im     : in image byte,
  g   :out array(0..MAXSIZE-1) of byte,
  n   :out array(0..MAXSIZE-1) of byte,
  Start : out array(Rows) of integer,
  m      : out integer) is
  NRows : integer;

first begin
  m := -1;
  NRows := 0;
end first;

next begin
  if col = 0 or im \= g(m) or
      n(m) = 255 then
    m := m + 1;
    g(m) := im;
    n(m) := 0;
    if col = 0 then
      Start(NRows) := m;
      NRows := NRows + 1;
    end if;
  else
    n(m) := n(m) + 1;
  end if;
end next;

combine
  offset : integer;
begin
  if NRows > 0 then
    offset := Start(NRows-1);
  else
    offset := 0;
  end if;
  for i in 1.._NRows loop
    NRows := NRows + 1;
    Start(NRows) :=
      _Start(i-1) + offset;
    offset := offset + _Start(i-1);
  end loop;
end combine;

end rle2;
```

Program 5. Run-length encoding, second version.

```
procedure rld(
  g   : in array(0..MAXSIZE-1) of byte,
  n   : in array(0..MAXSIZE-1) of byte,
  Start : in array(Rows) of integer,
 im     : out image byte) is

 offset, val, length : integer;

first begin
  if row > 0 then
    offset := Start(row);
  else
    offset := 0;
  end if;
  val := g(offset);
  length := n(offset) + 1;
  offset := offset + 1;
end first
next begin
  if length > 0 then
    length := length - 1;
  else
    val := g(offset);
    length := n(offset) + 1;
    offset := offset + 1;
  end if;
  outimg := val;
end next;

end rld;
```

Program 6. Run-length decoding.

3.4 Image Warping*

In image warping an input image is transformed into an output image according to a mapping function. The mapping function can be specified either in terms of the mapping from the input to the output, or the output to the input. The target pixel location (whether input or output) is in general not an integer, and must be calculated either by taking the closest position or by an interpolation method, for example bilinear or cubic convolution.

Perhaps the most straightforward way of implementing image warping in Adapt is to map each input pixel into the appropriate output pixel. In this case we want the transformation to be specified in terms of the input-to-output mapping.

If the transformation shrinks the image, multiple input pixels may map onto a single output. These pixels must be

*This section is based on joint work with George Gusciora, which has been funded by Kodak Corporation.

handled carefully if we are to produce a correct Adapt program. We cannot simply assign the output pixel the value of the “most recent” input pixel that mapped onto it, for this would make the program dependent on the order of processing of the input image—and ultimately, would result in the program giving different results when implemented on different architectures.

Instead, we will sum all input pixels that map to an output, and average the result in a second step. This produces the desirable effect of averaging, rather than subsampling, the input image when it is reduced in size. The invariant for the first step is:

> Each element of `op` is the sum of all input pixels in the region that map onto it; the corresponding value of `count` is the number of input pixels that mapped onto it.

A program that implements image warping is shown in Program 7. We do not show the program that divides `op` by `count` to produce the final output, which is trivial. Note that this program takes advantage of raster order processing to reduce the difficulty of computation of the affine transformation at each pixel. Note also that this program requires any processor implementing *Next* to have access to the entire output image; this is quite a large memory requirement. While this requirement can be reduced arbitrarily by producing a portion of the output image in each pass over the input, this results in a linear increase in execution time since each pass will perform redundant computation over the input as it calculates the target addresses for input pixels that are not mapped in this pass.

```
procedure affinewarp(
  ip    : in image byte,
  op    : out array(0..Width-1,
              0..Height-1) of byte,
  count : out array(0..Width-1,
              0..Height-1) of integer,
  warp  : in array(0..1,0..2) of
float)
is

first
begin
  for r in 0..Rows-1 loop
    for c in 0..Cols-1 loop
      op(r,c) := 0;
      count(r,c) := 0;
    end loop;
  end loop;
end first;

next
  x, y: float;
  ix, iy: integer;
begin
  if col = 0 then
    x := row * warp(0,0) + warp(0,2);
    y := row * warp(1,0) + warp(1,2);
  end if;
  ix := integer(x);
  iy := integer(y);
  if ix >= 0 and ix < Rows and
     iy >= 0 and iy < Cols then
   op(ix,iy) := op(ix,iy) + ip;
   count(ix,iy) := count(ix,iy)+1;
  end if;
  x := x + warp(0,1);
  y := y + warp(1,1);
end next;

combine
begin
  for r in 0..Rows-1 loop
    for c in 0..Cols-1 loop
      count(r,c) := count(r,c) +
                    _count(r,c);
      op(r,c) := op(r,c)+ _op(r,c);
    end loop;
  end loop;
end combine;

end affinewarp;
```

Program 7. Image Warping—input partitioning.

We can also calculate the input pixel that maps onto a given output and assign it this value. This is the traditional method of image warping. The invariant is simply:

> Each output pixel has the value of the nearest input pixel that maps onto it.

A program that implements image warping using this invariant is shown in Program 8. It is simpler than Program 7 because it can be implemented as a purely local operation (treating the input image as a global variable). Once again, this program has large memory requirements: the entire input image must be accessible to any processor executing the *Next* function. Just as in Program 7, we can reduce these memory requirements by processing only a portion of the input image in each pass over the output; doing so once again results in a linear increase in execution time.

```
procedure affinewarp(
  ip   : in array(0..Height-1,
                  0..Width-1) of byte,
  op   : out image byte,
  warp : in array(0..1,0..2) of
float)
is

next
  x, y : float;
  ix, iy : integer;
begin
 if col = 0 then
  x := row * warp(0,0) + warp(0,2);
  y := row * warp(1,0) + warp(1,2);
  end if;
   ix := integer(x);
   iy := integer(y);
   if ix >= 0 and ix < Rows and
      iy >= 0 and iy < Cols then
     op := ip(ix,iy);
   end if;
   x := x + warp(0,1);
   y := y + warp(1,1);
 end next;
```

Program 8. Image Warping—output partitioning.

Note that Program 7 is appropriate if we are producing a small output image from a large input; in that case, the output may indeed fit in an individual processor. Program 8 is appropriate if a small input is being mapped onto a large output.

For a shrinking transformation, Program 8 subsamples the input to produce the output; this may result in aliasing and other undesirable effects. Program 7, on the other hand, averages each image region that maps onto an output pixel, which will produce a better result. However, Program 7 produces an incorrect result when the transformation expands the input image; it is possible for output pixels to be missed when they map onto a non-integral location in the input image. Program 8 correctly assigns some value to all output pixels that map into the interior of the input image. Neither program really produces the “correct” result in all cases.

It is possible to modify Program 7 so that the correct output image is produced in all cases. All that is necessary is to update all pixels that an input pixel maps onto, instead of only one. Such a program is shown in Program 9. An array `pixels` stores the coordinates of the parallelogram that a single pixel maps onto. When processing an input pixel, all pixels in that parallelogram are updated. Note that just as in Program 7, the entire output image must be accessible at a processor executing *Next*, but that it is possible to reduce memory use by processing the output image in multiple passes, though with a linear increase in total execution time.

Both of these programs require that an entire image be accessible within the context of the *Next* section. This can impose prohibitive memory requirements, since usually all variables that are accessed within a single execution of *Next* must be stored in the memory of a processor. It is possible to significantly reduce memory requirements using a scan-line image warping algorithm [22] .

```
procedure affinewarp(
     im : in transposed image
          array(0..Width-1, 0..0)
                              of byte
          sample(0..Width-1, 0..0),
     a, b, c, d : in float,
     om : out image array(1, Cols)
          of byte)
is
 intx, inty: integer;
 x, y, tmpx, tmpy, cst : float;

next begin
  y:= float(row);
  cst:= (y * b) + c;
  for i in 0..Cols-1 loop
    x:= float(i);
    tmpx:= (x - cst) / (d * a);
    intx:= integer(tmpx);
    if intx >= 0 and intx < Cols then
      om(0, i):= im(intx, 0);
    else
      om(0, i) := 0;
    end if;
  end loop;
end next;

end affinewarp;
```

Program 10. Image warping using scanline method.

Such algorithms work by first skewing the image in two orthogonal directions. The first skew aligns all the columns of the skewed image with their ultimate destination, and the second skew completes the transformation by lining up

```
procedure affinewarp(
  ip   : in image byte,
  op   : out array(0..Height-1,
                   0..Width-1) of float,
  count : out array(0..Height-1,
                   0..Width-1) of byte,
  warp : in array(0..1,0..2) of float)
is
  pixelcount : integer;
  pixels  : array(0..MAXEXPAND-1, 0..1)
                            of float;

first
  horiz, vert : integer;
  addx, addy  : float;
begin
  horiz := max(abs(warp(0,0)),
                abs(warp(1,0)));
  for t in 0..horiz loop
    pixels(t,0) := warp(0,0)*horiz/t +
                   warp(0,2);
    pixels(t,1) := warp(1,0)*horiz/t +
                   warp(1,2);
  end loop;
  pixelcount := horiz;
  vert := max(abs(warp(0,1)),
               abs(warp(1,1)));
  for t in 1..vert loop
    addx := warp(0,1)*vert/t;
    addy := warp(1,1)*vert/t;
    for s in 0..horiz loop
      pixels(pixelcount + s, 0) :=
                   pixels(s,0) + addx;
      pixels(pixelcount + s, 1) :=
                   pixels(s,0) + addy;
    end loop;
    pixelcount := pixelcount +
horiz;
  end loop;
end first;
next
  px, py  : integer;
  x, y    : float;
begin
  if col = 0 then
    x := row * warp(0,0);
    y := row * warp(1,0);
  else
    x := x + warp(0,1);
    y := y + warp(1,1);
  end if;
  for i in 0..pixelcount-1 loop
    px := x + pixels(i,0);
    py := y + pixels(i,1);
    if px >= 0 and px < Rows and
       py >= 0 and py < Cols then
      op(px, py) := op(px, py) + ip;
      count(px, py) :=
        count(px, py) + 1;
    end if;
  end loop;
end next;

combine
begin
  for r in 0..Rows-1 loop
    for c in 0..Cols-1 loop
      count(r,c) := count(r,c) +
                    _count(r,c);
      op(r,c) := op(r,c) + _op(r,c);
    end loop;
  end loop;
end combine;

end affinewarp;
```

Program 9. Image Warping—input partitioning, second version

all the rows similarly.

An Adapt program for affine image warping using this method is shown in Program 10. The same Adapt program is applied twice, with different values of the `a`, `b`, `c`, and `d` parameters. The input image is transposed before being processed in each pass (by the `TRANSPOSED` keyword applied to the `im` parameter) so that the two transposes necessary to complete the image warping are evenly distributed over the passes. In the first pass, `a` is the cosine of the angle through with the image is being rotated, `b` is the negative of the sine of the angle, `c` is the translation in the X direction, and `d` is the scaling factor in the X direction. In the second pass, `a` is the inverse of the cosine of the angle, `b` is the tangent of the angle, `c` is the translation in the Y direction, and `d` is the scaling factor in the Y direction.

3.5 Image Connected Components*

In image connected components, an input binary image is labelled so that output pixels have the same label if, and only if, they belong to the same region in the input image. This is an important transitional algorithm between low- and mid-level image processing since it allows the measurement of features related to the shape of two-dimensional

*This section is based on joint work with James Kistler, which is reported in more detail elsewhere [11] , including a discussion of other algorithms.

objects in a scene. After image connected components, image features are often summarized in terms of the connected regions, and an image is reduced to a small number of regions and their associated features.

Two things make image connected components an especially interesting target for the application of parallelism. First, the output is an image, as in image warping; this makes it necessary to consider methods in which the output is stored in a distributed fashion. Second, the algorithm is global, and highly sensitive to local events; the presence or absence of a single pixel can affect the labelling of all pixels in the image. This makes it impossible to treat image connected components as an essentially local operation with some global mapping transformation, as we did with image warping in Program 7 and Program 9.

We will discuss three algorithms for connected components:

1. Nearest neighbor: each significant pixel is assigned a unique label, and then each pixel repeatedly receives the maximum of the adjacent labels. This process is repeated until no pixels change.

2. Shrink-expand: image regions are shrunk using Levialdi's morpohological shrink operation [12] , so that each image component is eventually shrunk down to the lower-right pixel of its bounding box (before disappearing). The value at each iteration is saved in a vector for each pixel. Using the vectors, the shrinking process is then inverted, so that eventually all components have reappeared, now properly labelled.

3. Union-find: The algorithm is in three phases. In the first, each pixel gets a unique label. In the second, the image is scanned, and an equivalence table is created that maps every pixel in the image to its final label pixel. This table is updated when it is discovered that two image components, formerly thought to be separate, are connected. In the third phase this table is applied to the image get the final image labels.

3.5.1 Connected components via nearest neighbor

The invariant for the step that assigns the initial label, called `init`, is

> Each pixel has the value `row*Rows+col+1`.

This is accomplished by a trivial Adapt program, not shown here.

The second step, called `prop`, is only slightly more complex. To terminate the program when the propagation is done we introduce a variable `changed` which records whether a pixel has changed:

> Each pixel gets the maximum of the labels of the pixels near it. `changed` is 1 if, and only if, a pixel has changed.

A program for this step is shown in Program 11. The *First* function initializes `changed` to 0, since no pixel is changed in the null region. The *Next* function does the maximization in lines 10 through 30, taking care to treat image pixel 0 (no label) differently. The *Combine* function combines the two values for `changed` in the obvious way.

While this program is very simple, it is also inefficient. Each pass over the image does useful computation only at pixels which change; this is approximately the number of pixels on the boundary of the region. In the worst case, the number of passes can be proportional to the size of the image. However, the program makes minimal use of interprocessor communication (nearest neighbor only) and requires only a small amount of memory at each processor.

3.5.2 Connected components via shrink-expand

The `shrink` step in this algorithm is nothing more than an implementation of Levialdi's morphological operation in Adapt, including a test for termination. The program is shown in Program 12. The program constructs a variable `SMix` that is an encoding of the relevant pixels surrounding the current pixel; this is used as an index into a lookup table that encodes the morphological operation. It makes use of Adapt's raster-order processing to avoid completely recomputing this variable at every pixel. Instead, it computes it once for the first column of the image in lines 9 through 21. Then, when processing a new pixel, it simply shifts the variable (line 23) and uses bitwise or to compute the

```
procedure prop(
 imagein:in image array(-1..1,-1..1)
         of float border 0,
 imageout:  out image float,
1 changed:   out byte) is

2 first begin
3   changed := 0;
4 end first;

5 next
6   current: float;
7 begin
8   current := imagein(0, 0);
9   if current /= 0 then
10    if imagein(-1, 1)>current then
11      current := imagein(-1, 1);
12    end if;
13    if imagein(0, 1)>current then
14      current := imagein(0, 1);
15    end if;
16    if imagein(1, 1)>current then
17      current := imagein(1, 1);
18    end if;
19    if imagein(-1,0)>current then
20      current := imagein(-1, 0);
21    end if;
22    if imagein(1, 0)>current then
23      current := imagein(1, 0);
24    end if;
25    if imagein(-1,-1)>current then
26      current := imagein(-1, -1);
27    end if;
28    if imagein(1,-1)>current then
29      current := imagein(1, -1);
30    end if;
31    if current /= imagein(0,0) then
32      changed := 1;
33    end if;
34  end if;
35  imageout := current;
36 end next;

37 combine begin
38   if _changed /= 0 then
39     changed := 1;
40   end if;
41 end combine;

42 end prop;
```

Program 11. Propagate step in nearest neighbor propagation.

updated value in lines 24 through 29.

```
procedure shrink(
  InImage:   in image
             array(-1..0, -1..0) of
             byte border 0,
  ShrinkMap: in array(0..15) of byte,
  OutImage:  out image byte,
1 Finished:  out byte) is
2 SMix:      integer;

3 first begin
4   Finished := 1;
5   SMix := 0;
6 end first;

7 next begin
8   if col = 0 then
9     SMix := 0;
10    if InImage(0, 0) /= 0 then
11      SMix := SMix lor 1;
12    end if;
13    if InImage(-1, 0) /= 0 then
14      SMix := SMix lor 2;
15    end if;
16    if InImage(0, -1) /= 0 then
17      SMix := SMix lor 4;
18    end if;
19    if InImage(-1, -1) /= 0 then
20      SMix := SMix lor 8;
21    end if;
22  else
23    SMix := (SMix lsh 2) land 15;
24    if InImage(0, 0) /= 0 then
25      SMix := SMix lor 1;
26    end if;
27    if InImage(-1, 0) /= 0 then
28      SMix := SMix lor 2;
29    end if;
30  end if;
31  OutImage := ShrinkMap(SMix);
32  if OutImage /= 0 then
33    Finished := 0;
34  end if;
35 end next;

36 combine begin
37   if _Finished = 0 then
38     Finished := 0;
39   end if;
40 end combine;

41 end shrink;
42
```

Program 12. Shrink step in shrink-expand.

The program that calls `shrink` must save the images it produces (parameter `OutImage`) and supply them in reverse order to the `expand` procedure. This procedure is shown in Program 13. It simply examines the next image produced by `shrink` and propagates the label appropriately, creating new labels as needed using the same numbering scheme as in nearest neighbor propagation.

```
procedure expand(
  InImage : in image byte,
  LabelsIn:in image array(0..1,0..1)
               of float border 0,
  LabelsOut : out image float) is

next begin
  if InImage /= 0 and
      LabelsIn(0, 0) = 0 then
    if LabelsIn(0, 1) /= 0 then
      LabelsOut := LabelsIn(0, 1);
    else if LabelsIn(1, 0) /= 0 then
      LabelsOut := LabelsIn(1, 0);
    else if LabelsIn(1, 1) /= 0 then
      LabelsOut := LabelsIn(1, 1);
    else
      LabelsOut := float(row) * DIM +
                   col + 1;
    end if; end if; end if;
  else
    LabelsOut := LabelsIn(0, 0);
  end if;
end next;

end expand;
```

Program 13. Expand step in shrink-expand

Note that many images are produced by `shrink` and they all must be stored until the `expand` step. This can use an enormous amount of memory; as much as 1 gigabit for a 512×512 image. However, since the operation on these images is local, on a large processor array these images can stay at each processor, so that the memory requirements per processor are small. On a processor with more limited memory, the images must be unloaded after each `shrink` pass and reloaded for each `expand` pass, leading to prohibitive I/O requirements.

3.5.3 Connected components via union-find

As in nearest-neighbor, union-find works by assigning initial labels to pixels, then changing these labels once it is found that two pixels are connected. One way to implement union-find, then, is to assign each pixel a unique label, then scan the image and perform a union operation between all two touching pixels. This allows the algorithm to be implemented in only two passes; one assigns initial labels and creates the equivalence table; the second applies the unified table. In fact, union-find has been implemented in this way on serial computers (see, for example, routine `CLAB` in).

However, a parallel implementation of this algorithm in the split and merge model does not work well, because of memory requirements. In assigning the initial labels, there is no *a priori* knowledge telling how many labels must be used in each region assigned to a processor. Therefore, it is necessary to assume the maximum; one-quarter the number of pixels. This means that the equivalence table has labels extending over a wide range, meaning that either a hash storage mechanism must be used to implement it, or the equivalence table will be large and wasteful of memory.

A better scheme is to make a first pass over the image and create serial labels for the pixels; no label is skipped. This results in a much smaller equivalence table in the usual case, but results in a three pass algorithm.

Such labels can be assigned by exploiting the raster-order processing feature of Adapt. Within each row, we assign labels sequentially, starting at 1. (At the same time, we reduce the total label set by assigning horizontal runs of pixels a unique label.) We also maintain an array `RowOffset` that gives a number that must be added to a label within a row to get its “real” value. The invariant is:

> The horizontal runs of pixels are assigned unique labels, starting with 1. The label for a pixel is `RowOffset(row)+outimg(row,col).FirstLabel(row)` is the label for the first labelled pixel in each row.

Given this invariant, the program is shown in Program 14. Two auxiliary variables, `FirstRow` and `LastRow`, record the starting and ending rows of each region. `NumLabels` is used to keep track of the next label (minus 1) to be assigned to a pixel.

```
    procedure cc1(
      inimg:   in image array(0..0, -1..0)
               of byte border 0,
      outimg:  out image integer,
      FirstLabel:out array(-1..Heigh+1) of
                 integer,
      RowOffset: out array(-1..Height+1) of
1                integer)

2   is
3     NumLabels:          integer;
4     FirstRow, LastRow:  integer;

5   first begin
6     FirstRow := Row;
7     LastRow := Row - 1;
8     NumLabels := 0;
9   end first;

10  next begin
11    if col = 0 then
12      LastRow := Row;
13      FirstLabel(Row) := NumLabels+1;
14      RowOffset(Row) := 0;
15    end if;
16    if inimg(0,0) /= 0 then
17      if inimg(0,-1) = 0 then
18        NumLabels := NumLabels+1;
19      end if;
20      outimg := NumLabels;
21    else
22     outimg := 0;
23    end if;
24  end next;

25  combine
26  begin
27    for i in _FirstRow.._LastRow loop
        RowOffset(i) := _RowOffset(i) +
          NumLabels +
28  RowOffset(FirstRow);
        FirstLabel(i) :=_FirstLabel(i) +
29        NumLabelsRowOffset(FirstRow);
30    end loop;
31    LastRow := _LastRow;
      NumLabels :=
32  NumLabels+_NumLabels;
33  end combine;

34  last begin
35    RowOffset(-1) := 0;
36    FirstLabel(-1) := 1;
37    RowOffset(Rows) := 0;
38    Firstlabel(Rows) := NumLabels + 1;
39  end last;
```

Program 14. First pass of union-find.

`FirstRow` and `LastRow` are initialized in lines 6 and 7. (The null region starts at the current row, and ends at the previous row.) Labels are assigned starting at 1, so `NextLabel` gets 0 in line 8. For the null region, `RowOffset` and `FirstLabel` have no value.

At the beginning of each row, we must update all variables. This is done in lines 11 through 15. `LastRow` now includes this row, and `FirstLabel` and `RowOffset` can be given values for this row. Within a region, no offset is necessary, so `RowOffset` gets 0 in line 14. Assigning a label to a pixel is trivial; we assign the current label if we are continuing a run, and a new label otherwise.

The *Combine* section must merge the two `RowOffset` arrays by adding to the second array the offset of the first, and the number of labels in the first array, and similarly for `FirstLabel`. This is done in lines 27 through 30. The update to `LastRow` and `NumLabels` is trivial.

In the *Last* section we extend the definition of `RowOffset` and `FirstLabel` slightly to include rows above and below the image.

The second pass of union-find is complex because the merge operation has several special cases and the implementation of the union operation requires multiple steps. The invariant is simple:

`Eqtbl(i)=Eqtbl(j)` if, and only if, the pixels with labels `i` and `j` are connected.

In implementing this invariant in Program 15, we use two macros (defined using the M4 macro facility [10] in lines 4 through 18). `CFIND(source, temp, table)` performs a compressing find in the equivalence table `table` starting at the label `source` and leaving the result in `temp`. `CSET(target, source, table)` sets all labels at the equivalence chain starting at label `target` in the equivalence table `table` to `source`.

In the scan of the region we always unify labels with the labels from the row above, starting with the first row. As a result it is possible, and even likely, for a label to be mapped into a label from the row above the region. Therefore,

```
procedure cc2(
  inimg:in image array(-1..1,-1..1) of
         integer border 0,
  Eqtbl:  out array(0..MAXLABELS-1) of
         integer,
  FirstLabel:   in array(-1..Rows) of
                 integer,
  RowOffset: in array(-1..Rows) of
               integer)

is
 FirstRow, LastRow : integer;

define('CFIND','
  $2 := $1;
  while $2 /= $3($2) loop
    $2 := $3($2);
  end loop;
  CSET($1, $2, $3)')
define('CSET','
  tmp1 := $1;
  tmp2 := $3(tmp1);
  $3(tmp1) := $2;
  while tmp1 /= tmp2 loop
    tmp1 := tmp2;
    tmp2 := $3(tmp1);
    $3(tmp1) := $2;
  end loop;')

first
  i: integer;
begin
  FirstRow := Row;
  LastRow := Row - 1;
  for i in FirstLabel(row-1)..
    FirstLabel(row)-1 loop
    Eqtbl(i) := i;
  end loop;
end first;

next
  i, tmp1, tmp2: integer;
  pattern: integer;
  source: integer;
begin
  if col = 0 then
    LastRow := Row;
    for i in FirstLabel(row)..
      FirstLabel(row+1) - 1 loop
      Eqtbl(i) := i;
    end loop;
    pattern := 0;
    if inimg(0,0) /= 0 then
      pattern := pattern lor 8;
    end if;
   if inimg(-1,0) /= 0 then
      pattern := pattern lor 2;
    end if;
    if inimg(-1,1) /= 0 then
      pattern := pattern lor 1;
    end if;
  else
    pattern := (pattern lsh 1) land 22;
    if inimg(0,0) /= 0 then
      pattern := pattern lor 8;
    end if;
    if inimg(-1,1) /= 0 then
      pattern := pattern lor 1;
    end if;
  end if;
  if pattern >= 12 and
      pattern <= 15 then
    CFIND(inimg(-1,-1) +
      RowOffset(Row-1), source,
Eqtbl)
    CSET(inimg(0,0) + RowOffset(Row),
      source, Eqtbl)
  end if;
  if pattern = 10 or pattern = 11 then
    CFIND(inimg(-1, 0) +
      RowOffset(Row-1), source,
Eqtbl)
    CSET(inimg(0,0) + RowOffset(Row),
      source, Eqtbl)
  end if;
  if pattern = 25 or pattern = 13 or
    pattern = 9 or pattern = 29 then
    CFIND(inimg(-1, 1) +
      RowOffset(Row-1), source,
Eqtbl)
    CSET(inimg( 0, 0) + RowOffset(Row),
      source, Eqtbl)
  end if;
end next;

combine
  i, tmp1, tmp2: integer;
  source, target: integer;
begin
  for i in FirstLabel(_FirstRow)..
    FirstLabel(_LastRow+1) - 1 loop
    CFIND(i, source, _eqtbl)
    eqtbl(i) := source;
  end loop;
  for i in FirstLabel(_FirstRow - 1)
    FirstLabel(_FirstRow) - 1 loop
    CFIND(i, source, eqtbl)
    target := i;
    while_eqtbl(target)/=target loop
      CSET(target, source, eqtbl)
      target := _eqtbl(target);
    end loop;
    CSET(target, source, eqtbl)
  end loop;
  LastRow := _LastRow;
end combine;

last
```

Program 15. Second pass of union-find.

the equivalence table must have a value for these labels. We establish this in the *First* section in lines 24 through 26.

As the first row is processed, the equivalence table must be initialized so that labels in this row map into the identity. This is done in lines 35 through 38.

`pattern` is used to efficiently keep track of the binary pattern of the five pixels, at, to the left, above and to the left, above, and above and to the right of the central pixel. It is these pixels that may contribute their label to the central pixel, or which may have their labels unified. The value in `pattern` is a bit encoding of whether these pixels are non-zero or not, in the manner shown in Figure 3. Creating `pattern` at the first column is simplified because there are no pixels to the left; this is done in lines 39 through 48. Updating `pattern` for the next pixel is simplified because the old value can be reused; this is done in lines 50 through 56.

bit position 2	bit position 1	bit position 0
bit position 4	bit position 3	

Figure 3. Bit encoding of pixels in `pattern`.

There are several special cases for which pixels must be unified. These were derived by hand, through exhaustive analysis of the thirty-two possible arrangements of the relevant pixels, and are shown in Figure 4. Lines 58 through 61 check for and do the unification with the pixel to the upper left, lines 62 through 65 do the unification with the pixel above, and lines 66 through 69 do the unification with the pixel to the upper right.

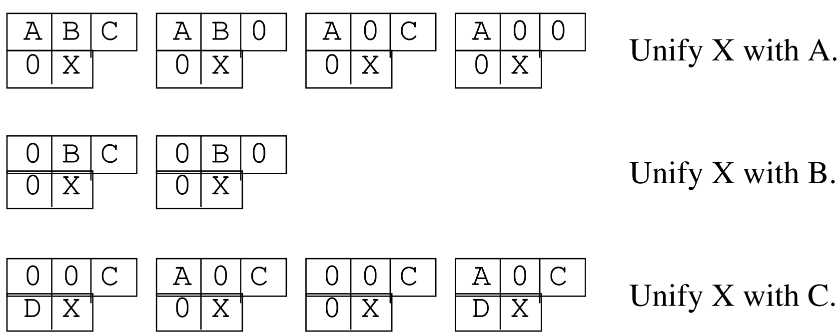

Figure 4. Case-by-case analysis for union-find.

The *Combine* section merges two equivalence tables. This section is considerably simplified because the two equivalence tables are for adjacent regions of rows; the only labels they can have in common occur along the bottom row of the upper region. Accordingly, we simply copy all other labels from the bottom equivalence table into the top table in lines 75 through 78, and then consider the possibly shared labels in lines 79 through 87. For each such label, we find its current value in the combined equivalence table, and then scan the chain of equivalences for this label in the lower equivalence table in lines 82 through 87. Each label in this chain is set to the value of the label in the combined table.

In the next and final step of connected components we will apply the equivalence table to the image in a trivial Adapt program. The *Last* section prepares for this pass by performing a compressing find on all the labels in the table.

This version of the connected components algorithm is the most like a sequential algorithm, and it is likely to be the fastest on a parallel computer with a relatively small number of processors. However, it makes the most demanding memory requirements of any of the algorithms since the equivalence table must be stored locally.

3.6 Two-dimensional Fast Fourier Transform*

The two-dimensional fast Fourier transform is calculated in two passes over the image, in orthogonal directions. In the decimation-in-time algorithm presented here, the data within each row or column is first rearranged in bit-reversed order, and then $\log_2 N$ passes are made over the data, where N is the number of data in the row or column.

Adapt makes it easy to write such an algorithm, because an entire row of column of data can be accessed as a unit within *Next* by correctly applying `SAMPLE` and the array parameter to the input images. `SAMPLE(r,c)` causes `r-1` rows and `c-1` columns to be skipped between each invocation of *Next*; therefore, `SAMPLE(1,Cols)` causes *Next* to be applied once per row. If the `ARRAY` parameter applied to the input image is also `(1, Cols)` then the entire row is available for processing within each *Next*, and is processed once only. If the input image is `TRANSPOSED`, as in Program 16, then the above applies to columns of the image, not rows.

```
   procedure fft(
     inr  : in transposed image
          array(0..Height-1,0..0) of float
          sample(Height, 1),
     ini  : in transposed image
          array(0..Height-1,0..0) of float
             sample(Rows, 1),
     outr:out image array(0..0,0..Rows-1)
             of float,
     outi:out image array(0..0,0..Rows-1)
             of float,
1    sign: in integer) is

2  multiplier : float;

3  first begin
4    multiplier := sqrt(1.0/Rows);
5  end first;

6  next
7    i, j, m, mmax : integer;
     theta, wtemp, temp, tempr, tempi,
8      wpr, wpi, wr, wi : float;
9  begin
10   for i in 0..Rows-1 loop
11     outr(0,i):=inr(i,0)*multiplier;
12     outi(0,i):=ini(i,0)*multiplier;
13   end loop;
     -- Do bit reversal
14   j := 0;
15   for i in 0..Rows-1 loop
16     if j > i then
17       SWAP(outr(0,j),outr(0,i))
18       SWAP(outi(0,j),outi(0,i))
19     end if;
20     m := Rows rsh 1;
21     while m >= 1 and j >= m loop
22       j := j - m;
23       m := m rsh 1;
24     end loop;
25     j := j + m;
26   end loop;
     -- Danielson-Lanczos loop
27   mmax := 2;
28   while Rows >= mmax loop
29     theta := 6.2831853/(mmax*sign);
30     wtemp := sin(0.5*theta);
31     wpr := -2.0*wtemp*wtemp;
32     wpi := sin(theta);
33     wr := 1.0;
34     wi := 0.0;
35     for m in 0..(mmax rsh 1)-1 loop
36       i := m;
37       while i < Rows loop
38         j := i + (mmax rsh 1);
           tempr := wr*outr(0,j) -
39                  wi*outi(0,j);
           tempi := wr*outi(0,j) +
40                  wi*outr(0,j);
41         outr(0,j) := outr(0,i)-tempr;
42         outi(0,j) := outi(0,i)-tempi;
43         outr(0,i) := outr(0,i)+tempr;
44         outi(0,i) := outi(0,i)+tempi;
45         i := i + mmax;
46       end loop;
47       wtemp := wr;
48       wr := wr*wpr - wi*wpi + wr;
49       wi := wi*wpr + wtemp*wpi + wi;
50     end loop;
51     mmax := mmax lsh 1;
52   end loop;

53 end next;

54 end fft;
```

Program 16. Two-dimensional fast Fourier transform.

Program 16 is one pass of the fast Fourier transform. Since the input image is transposed in line 1 and the output image is not, the output image is stored in transposed order relative to the input; therefore, the second pass of fast Fou-

* This section is based on some ideas due to Bruce Greer of Intel Corporation.

rier transform is accomplished by another execution of this routine. The code in the body of *Next* is taken from [14] , pages 411-412; the only changes have to do with storing the complex input and output values as two separate arrays, rather than shuffling the real and complex values together, and the code in lines 10 through 13 that divides each element of the transform by the square root of the size of the transform, so that the inverse transform gives back exactly the original.

3.7 Summary

In this section we have defined the Adapt language and given examples of Adapt programs and programming technique. In Adapt programs, to a greater extent than in other languages, the invariant plays a key role. Since it is possible for the Adapt compiler to execute parts of Adapt programs in different orders depending on the target machine, it is necessary to develop the program with an invariant in mind, which will guarantee the program correct regardless of the execution order.

4. Adapt Implementations

On a serial machine, an Adapt program can be implemented as shown in Figure 5. In this implementation, the *First* function is executed once, at the beginning of the image, and then the *Next* function is executed all across the image; finally, the *Last* function is executed. The *Combine* function is not executed at all.

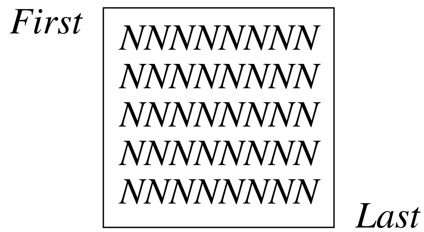

Figure 5. Adapt Serial Implementation

On a parallel machine, the input image can be divided either by rows or by columns. If it is divided by rows one possibility is shown in Figure 6. Here each processor takes a horizontal swath of the image, and executes the *First* function at the beginning of the swath. The *Next* function is executed by each processor across its swath. Then *Combine* functions are used (here serially) to combine the results computed independently by the processors. Finally, the *Last* function is executed.

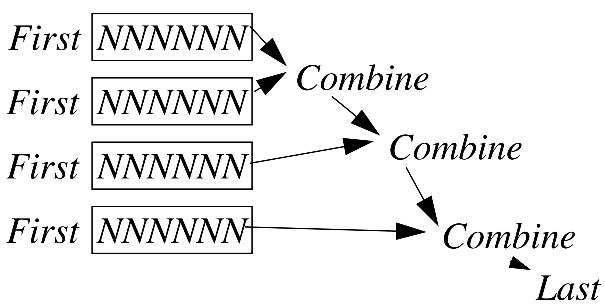

Figure 6. Row-Partitioned Implementation of Adapt

If the image is partitioned by columns, a possible implementation is shown in Figure 7. For each row, a processor executes the *First* function, then runs *Next* functions across its portion of the image. Then it transfers its intermediate variables to another processor. This second processor runs *Next* functions on its portion of the image, then transfers its intermediate results to a third processor. The last processor in this chain executes *Next* functions on its portion of the image and then executes a *Combine* function to combine the results just computed with the results for the previous portion of the image. After the entire image is processed, the *Last* function is executed.

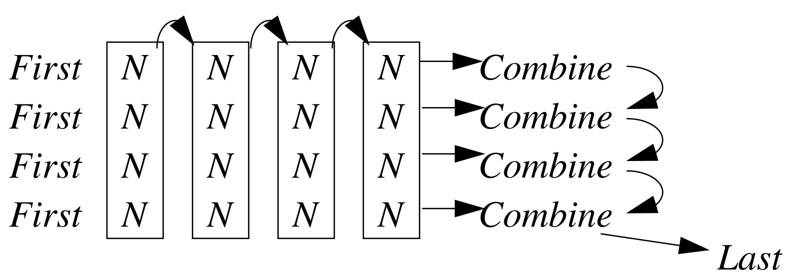

Figure 7. Column-partitioned implementation of Adapt.

The column-partitioned implementation is less efficient than the row-partitioned implementation; intermediate results must be passed between processors, and each processor must wait for the previous processor to complete before it can begin. (Of course, the processors may still be working all at the same time, in a pipeline.) The tight coupling between processors and the need for a lot of interprocessor communication may seriously reduce performance on many parallel computers. However, this implementation has some advantages: The amount of input image stored at each processor is smaller, allowing it to be used when the processors have small memories; The *Combine* function is run in parallel with the *Next*, while the row-partitioned implementation executes all the *Combine* functions after all the *Next* functions.

On a two-dimensional processor mesh, the executions of the *Combine* function can be modified to be more efficient.Figure 8 illustrates this. The image is still partitioned by rows, but now the *Combine* function is first executed horizontally across the processor array in parallel, and then it is executed vertically. This method increases the number of processors that can be applied to an image processing problem because the overhead for the *Combine* step is reduced. This is important because processor meshes can be implemented much more efficiently, and with much higher interprocessor bandwidth, than binary tree arrays.

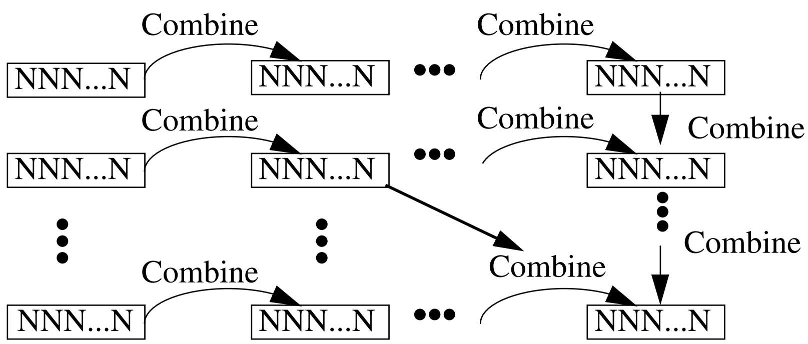

Figure 8. Implementation of Adapt on a two-dimensional processor array.

Adapt has been implemented on Unix serial architectures using the serial implementation method, on the Carnegie Mellon Warp machine [2] in two separate implementations, one using row partitioning and the other using column partitioning, and on the Carnegie Mellon Nectar computer using column partitioning. We now discuss these implementations.

4.1 Serial implementation

The serial implementation of Adapt is extremely straightforward: the Adapt compiler generates a C program that takes as arguments the images and other parameters to the Adapt program. Images are stored as simple two-dimensional arrays, and are accessed using ordinary array indexing. No optimization of array access is done. The compiler simply generates a two-level loop over the rows and columns of the image; the innermost portion of the loop is the *Next* section from the Adapt program. The loop is preceded by the *First* section and followed by the *Last* section.

This C program is generated in a single pass over the Adapt program; a yacc-based parser scans the input and generates the result with no intermediate processing step. Image borders are handled by a single handwritten subroutine that takes as input an image and the border description, and returns a pointer to the image which has been extended with the appropriate border. This subroutine is inserted in the generated code and appropriate calls to it are generated by the Adapt compiler.

4.2 Column-partitioned implementation on Warp

The column-partitioned method was the first implementation of Adapt on Warp. This method was used because the existing Apply compiler used this method, and an Adapt compiler could be easily developed from it.

Warp includes an input and output host processor, which supply and receive data from the linear ten cell Warp array. The host processors have access to large memories, but are far less computationally powerful than the Warp cells. As a result, we chose the implementation shown in Figure 9. The image is fed to the Warp array row-by-row, in column-raster order, by the input host. A row of the input is distributed in a `GETROW` operation [7] to all cells. The first cell then executes the *First* section and then *Next* operations over the first tenth of the columns of the image. It then passes its intermediate results on to the second cell, which executes *Next* over its columns of the image, and passes its intermediate results on to the next cell, also doing a `PUTROW` operation to output any resulting images. This process continues across the Warp cell array, until the last Warp cell executes its *Next* operations and passes the final results to the output host, which executes *Combine* between the results for all previous rows and this row. After all the rows have been processed, the output host executes *Last*.

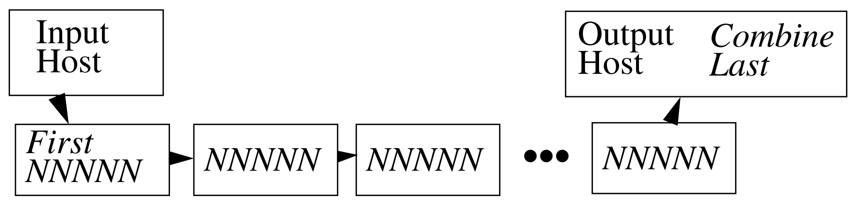

Figure 9. Column-partitioned implementation on Warp

This implementation method makes it unnecessary to allocate storage for variables used only in the *Combine* section on the Warp cells. This is important because memory on the Warp cells is quite limited: only 32Kwords are available for data.

The input and output queues between Warp cells are 512 words long. So long as all intermediate data will fit in these queues, a pipeline of processing is set up. However, there are many programs for which the intermediate results are larger than 512 words; for example, minimum bounding rectangle produces arrays which total 1024 words in length. For such programs, only one Warp cells is active at any time. Even when pipelined processing is possible, the resulting program may be inefficient since it must input and output all intermediate variables after every tenth of the columns of the image are processed.

4.3 Row-partitioned implementation on Warp*

The row-partitioned implementation on Warp divides the image into ten swaths, and then interleaves rows from different swaths of the image as it passes them to the Warp array. For example, for an image of height 512, the image rows are passed to Warp in this order: 0, 52, 104, 156,... Each cell executes *First* on its portion of the image (once only), and then executes *Next* all across its region. No intermediate results are exchanged until the entire image has been processed, when *Combine* operations are executed serially across the Warp array: the first cell sends its variables to the second, which executes *Combine*, and then sends its variables to the third cell, which does likewise, and so on. The last cell also executes *Last*, passing the final result to the output host.

This implementation results in much faster programs than the previous, for three reasons. First and second, as noted, the column-partitioned implementation does excessive interprocessor communication and results in unnecessary coupling between Warp cells; this implementation does not. Third, in this implementation the loop of *Next* operations is ten times longer than in the column-partitioned implementation. This is important on Warp, which has a deep floating point pipeline, which results in significant start-up overheads for each loop.

4.4 Row-partitioned implementation on Nectar

The Carnegie Mellon Nectar computer is a distributed network of workstations, connected by a high-bandwidth, low-latency crossbar network implemented via fiber optic links attached to a CMOS crossbar and connected to the

*This implementation was done by George Gusciora.

workstations via I/O processors called CABs.

The Nectar implementation of Adapt operates with a master-slave model. All input data begin, and all results end up, in a workstation called the master. The master first sends all slaves the non-image input parameters. Then the master divides the image into swaths, the number of swaths depending on the number of slaves, but generally at least twice as many swaths as slaves. It then sends each slave a swath. Each slave then executes *First* and *Next* operations over its swath. When a slave reaches the middle row of its swath, it sends a message to the master requesting more data; the master then sends the requesting slave another swath. When a slave completes its processing, it reads more data if available. Once the entire image has been processed, the master sends all slaves a map of which slaves have which swaths. The slaves then execute *Combine* operations independently of the master, according to the following strategy: the swath results are combined as in a binary tree. If a slave has both the left and right corresponding leaves, it combines them, creating a new leaf. If a slave has only the right leaf, it sends this leaf to the slave holding the left leaf. If a slave has the left leaf, it does nothing, because eventually another slave will send it the right leaf. Once the entire image has been combined, the slave sends the result back to the master, which executes *Last*.

A note on time measurement in Nectar is in order. It is much harder to measure performance of a Nectar program than a Sun or a Warp program. On the Sun, special operating system calls make it possible to isolate just the time allocated to a process; by running at times of low system load (easy to do in a workstation environment), it is relatively easy to measure execution time consistently. This is even easier on the Warp machine; it is a single user machine, and the time measured there varies only slightly (due to interactions between the host processors).

On Nectar the only meaningful number to measure is total wall clock time from the start to the end of execution on the master. This is, after all, the time the user would have to wait for his or her program to complete; it is not possible to separate system time (which might be minimized under the right conditions) since this varies from host to host. It is not possible to guarantee that the load on all of the hosts will be low (although this can be minimized by running at times of low overall load) since they are freely available for use in offices. We have therefore had to estimate execution times on Nectar by running several experiments and using model-based data analysis methods.

Our results from these experiments have established that the current Nectar implementation of Adapt suffers from the relatively low bandwidth available across the workstation bus. Every image is transmitted twice across this bus, once in the master and once in the slave. This bottleneck means that Adapt Nectar programs perform poorly compared to their counterparts running on a single workstation, which has a special-purpose high speed bus to its memory. For this reason, and because of the difficulty of measuring Nectar performance in the first place, we do not include Nectar performance figures here.

4.5 Performance

Table 2 gives the execution times of Adapt programs on Warp and the Sun 4/330, and, in cases where they exist, the execution times of equivalent Warp hand-coded programs. In most cases, the Warp hand-coded programs were carefully optimized for execution time.

Most of the programs in this table are discussed in this paper. The median filter programs are discussed elsewhere [21] . The halftoning program was developed as part of an approach to high quality parallel halftoning [20] . The pattern generation programs are routines `PGEN1` through `PGEN5` from the SPIDER library [16] .

Comparing the Warp hand-coded times with the Warp row-partitioned times we see that in general, Adapt does better than hand-coded programs. The reasons for this are: (1) In some cases, the hand-coded programs were designed under assumptions that became invalid with the evolution of the W2 compiler (the base language for Warp); as the compiler matured, certain special-purpose features were eliminated, which made the hand-coded programs less efficient. The Adapt compiler's program mapping is more recent, and corresponds more exactly to the current W2 compiler. Of course, this validates the Adapt approach since the Adapt programs can be kept up to date (by changing only one program, the Adapt compiler) much more than the hand-coded programs. (2) In some cases, the mapping technique used in the row-partitioned Adapt is better than the hand-coded programs, which generally use a column-partitioning of the image (although they do not create the same problems with interprocessor communication as in the Adapt column-parititoned mapping.) (3) In some cases, for example "Diamonds," the Adapt algorithm is faster.

One reason that the Adapt programs are occasionally slower is that the hand-coded W2 programs have an image size specified at compile time, while in all Adapt implementations the image size is specified at run time. This intro-

Program	Warp hand-coded	Warp Adapt column-partitioned		Warp Adapt row-partitioned		Sun 4/330 Adapt	
Histogram	0.143	3.43		0.119		0.163	
Minimum bounding rectangle	1.25	8.46		0.143		1.30	
Run-length encoding, second version		2.91		0.207		0.216	
Run-length decoding		0.271		0.189		0.166	
3×3 median filter	0.779	1.85		0.779		1.83	
Halftoning using Gibbs sampler		2.50		0.576		1.80	
Pattern generation:							
Checkerboard	0.106	0.253		0.178		0.200	
Stripes	0.151	0.334		0.280		1.30	
Bull's-eye	0.140	0.310		0.274		4.37	
Diamonds	0.926	0.395		0.338		2.53	
Grid	0.131	0.234		0.178		0.183	
Connected components:	4.51 (total)						
Size of regions in image:		Small	Large	Small	Large	Small	Large
Nearest neighbor propagation:							
Initialize		0.416	0.333	0.232	0.257	0.316	0.333
Propagate		3.14	102	1.69	59.5	2.57	73.3
Shrink-expand:							
Shrink step		2.92	149	1.23	64.4	3.28	169
Expand step		3.18	162	1.90	97.3	5.28	265
Union-find							
Step 1		6.50	6.50	0.347	0.356	0.853	0.769
Step 2		55.5	67.5	0.526	0.523	1.22	0.783
Step 3		0.528	0.549	0.428	0.429	0.456	0.354

Table 2. Execution times, in seconds, of Adapt programs for 512×512 image processing.

duces some extra overhead and eliminates some possibilities of optimization in the W2 compile stage for the Adapt programs.

Comparing the Warp column-partitioned times with the Warp row-partitioned times we see small differences created by the shorter length loops in the column-partitioned implementation, and large differences created by the excessive interprocessor communication in that implementation for some programs.

A comparison of the Warp row-partitioned times with the Sun 4 times shows that the Sun4 performs nearly as well as Warp except for algorithms in which there is a lot of floating point computation.

The results for connected components show the general superiority of the nearest-neighbor algorithm to the others on images with very small regions (only a few pixels each) and the superiority of the union-find algorithm on other images, because of its near constant execution time independent of the image regions. The union-find Adapt algorithm is faster than the hand-coded and optimized W2 version, which is also a union-find algorithm, because the Adapt algorithm is superior and because the machine mapping method is better.

5. Summary and Conclusions

Image processing is amenable to parallel implementation. What is needed to realize this promise is parallel languages that will provide a stable base for the development of code libraries that run on the continually improving, short-lived parallel computers that are available today.

Adapt is just such a language. It is based on a simple, powerful programming model: the split and merge programming model. Split and merge allows any computation that can be computed in forward or reverse order over a data structure to be programmed. It is a simple model to understand, and it can be implemented efficiently on a wide variety of parallel computers.

In programming in Adapt, a key idea is the invariant that applies over the image. By keeping in mind the idea of the invariant, the programmer can write programs that are guaranteed to be correct regardless of the underlying implementation of the Adapt program.

Several programs for a wide variety of image processing algorithms have been presented here. Such a wide variety of programs have been demonstrated that it is clear that Adapt can be used to program complete vision systems, with all image processing being done in Adapt.

Adapt can be implemented in several different ways, depending on the underlying architecture, as has been demonstrated by existing implementations of Adapt on the Sun, Warp, and Nectar. Performance from the implementations shows that Adapt is efficient, even when compared with hand code.

Acknowledgments

As noted in the paper, the results in this paper are in part collaborative work with George Gusciora and James Kistler. I gratefully acknowledge their contributions to the Adapt project. I also appreciate the contributions of Intel Corporation, in particular Bruce Greer and Brent Baxter, who contributed ideas to the development of Adapt.

This material is based upon work supported by a grant from Kodak Corporation, and by the National Science Foundation under Grant MIPS-8920420. The Government has certain rights in this material. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

6. Bibliography

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. Data Structures and Algorithms. Addison-Wesley, 1983.

2. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A. “The Warp Computer: Architecture, Implementation and Performance”. *IEEE Transactions on Computers C-36*, 12 (December 1987), 1523-1538.

3. Baxter, B., Cox, G., Gross, T., Kung, H. T., O'Hallaron, D., Peterson, C., Webb, J., and Wiley, P. Building Blocks for a New Generation of Application-Specific Computing Systems. International Conference on Application Specific Array Processors, Princeton, NJ, September, 1990.

4. Dew, P. M., Wang, H. and Webb, J. A. Implementation of Apply on a Transputer Array. To appear, Concurrency: Practice and Experience, 1991.

5. Eager, D. L., Zahorjan, J., and Lazowska, E. D. “Speedup versus efficiency in parallel systems”. *IEEE Transactions on Computers 38*, 3 (1989), 408-423.

6. Fisher, A., Highnam, P. T., and Leon, J. Design and Performance of a SIMD Optimizing Compiler. Third Symposium on the Frontiers of Massively Parallel Computing, 1990.

7. Hamey, L. G. C., Webb, J. A., and Wu, I-C. “An Architecture Independent Programming Language for Low-Level Vision”. *Computer Vision, Graphics, and Image Processing 48* (1989), 246-264.

8. Haralick, R. M. Gipsy: General image processing system. Product description brochure, Intelligent Systems Laboratory, Department of Electrical Engineering FT-10, The University of Washington, Seattle, WA 9819.

9. Hoare, C. A. R. “Partition (Algorithm 63) and Quicksort (Algorithm 64)”. *Comm. ACM 4*, 7 (July 1961), 321.

10. Kernighan, B. W. and Ritchie, D. M. The M4 Macro Processor. In *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, NJ 07974, 1979.

11. Kistler, J., and Webb, J. A. Connected Components with Split and Merge. International Parallel Processing Symposium, Anaheim, Ca, 1991.

12. Levialdi, S. “On Shrinking Binary Pictures”. *Comm. ACM 15*, 1 (1972), 7-10.

13. Menzilcioglu, O. *Using Powerful Processors in a Configurable Systolic Array Architecture.* Ph.D. Th., Carnegie Mellon University, October 1988.

14. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, 1988.

15. Scudder, M. An Apply Compiler for the CAAPP. University of Massachusetts at Amherst, 1990.

16. Electrotechnical Laboratory. SPIDER (Subroutine Package for Image Data Enhancement and Recognition). Joint System Development Corp., Tokyo, Japan, 1983.

17. Stout, Q. F. Properties of divide-and-conquer algorithms for image processing. Workshop on Computer Architectures for Pattern Analysis and Image Database Management, IEEE Computer Society, 1985, pp. 203-209.

18. Wallace, R. S. and Howard, M. D. HBA Vision Architecture: Built and Benchmarked. Computer Architecures for Pattern Analysis and Machine Intelligence, IEEE Computer Society, Seattle, Washington, December, 1987.

19. Wallace, R. S., Webb, J. A. and Wu, I-C. Architecture Independent Image Processing: Performance of Apply on Diverse Architectures. 1989, pp. 265-276.

20. Webb, J. A. Accurate Parallel Digital Halftoning. Society for Information Display, Anaheim, CA, 1991.

21. Webb, J. A. Architecture-Independent Global Image Processing. Tenth International Conference on Pattern Recognition, International Assocation for Pattern Recognition, Atlantic City, NJ, June, 1990, pp. 623-628.

22. Wolberg, G. Digital Image Warping. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA, 90720-1264, 1990.

I. Adapt Syntax and Semantics

Adapt syntax and semantics are based as much as possible on the syntax and semantics of Ada. This has been done to reduce the burden of memorizing a set of different operators for a new language or, worse, remembering a set of operators that are similar to, but slightly different from, an existing language. In some cases it has not been possible to exactly imitate Ada's semantics, for example in the semantics of **out** parameters to Adapt procedures. However, usually the syntax and semantics have been followed, and any deviation from what is expected should be reported to the Adapt maintainers.

I.1 Top-level syntax

An Adapt file consists of a number of procedures, functions, and procedure or function declarations. Procedures can be of two types: *ordinary* or *Adapt*. Functions and ordinary procedures are similar to those in Ada, while Adapt procedures provide the data parallel abstraction that allows Adapt to efficiently apply parallel computers to image processing.

I.2 Adapt Pocedures

An Adapt procedure consists of a procedure header followed by global declarations for the procedure, followed by (up to) four *sections*, as shown in Figure 1.

```
procedure program(parameters) is
  global declarations

first
  first declarations
begin
  first statements
end first;

next
  next declarations
begin
  next statements
end next;

combine
  combine declarations
begin
  combine statements
end combine;

last
  last declarations
begin
  last statements
end last;

end [program];
```

Figure 1. Adapt procedure syntax.

The parameters to the program are images and other variables. The Adapt compiler maps the computation described within the program over the images.

Variables declared in the global declaration section are common to all sections of the code, and can be accessed from any section. Variables declared in the **first**, **next**, **combine**, and **last** sections of the code are local to those sections.

The **first** section of the code is executed before any computation is done. It is intended that this section of the code be used to initialize variables before processing in the **next** section. Depending on how the image is mapped over the parallel processors, it may be executed once at the beginning of the image (that is, at row and column 0), or it may be executed once for each section of the image processed by a processor (that is, at column 0 of the first row processed), or it may be executed once per row. Whenever it is executed, it will be executed at column 0 of the image.

The **next** section of the code is executed once per pixel, in raster order, across the image. Any execution of **next** is preceded by either another **next** or a **first**. Variables retain their values from one execution of **next** to another; this makes it possible to make operations more efficient by taking advantage of raster-order processing of the image. Execution of **next** wraps around from the **last** pixel of a row to the first pixel of the **next** row, without necessarily executing **first** again at the beginning of the row.

The **combine** section of the code is executed to combine results from two independent regions of the image. The two regions are always adjacent blocks of rows. These results are computed by previous executions of **next** or **combine** over the regions. The two sets of results are referred to by special names: the variables from the upper region receive the usual names, and the variables from the lower region receive the variable name preceded by an underscore. The function of the **combine** section is to take these two versions of variables and reduce them to one, with names not preceded by underscore, by performing appropriate calculations. The calculation should be such that the same results would have been computed by repeated execution of **next** over the entire region.

The **last** section of the code does any final computation on the data, perhaps discarding intermediate results. It is executed once, after the entire image has been processed by **first**, **next**, and **combine**.

If the Adapt program is a pure image processing program, that produces only variables of type **out image**, then only the **next** section is required. If the program produces variables of type **out** then both the **next** and **combine** sections must be supplied.

The images can be accessed only with the **first** and **next** sections. Only input images can be accessed within **first**.

I.2.1 Parameters to Adapt procedures

Parameters may be input or output images, or ordinary input or output variables. The syntax is shown in Figure 2.

```
parameters ::= parameter {, parameters}                                   1
parameter ::= variable {, variable} : {in|out}
                                              [transposed] [image]
     [array(range, range) of]
          {signed_byte | unsigned_byte | byte | signed_integer | unsigned_integer |
                                              integer | float | long_float]
                                  [border <value> | replicated | mirrored]
                                  [sample (rows, cols)                    2
range ::= expression..expression                                          3
```

Figure 2. Parameter Syntax

The correspondence between Adapt and C types is shown in Table 1.

The semantics of a parameter differ greatly depending on whether or not it is an image.

Ordinary variables (non-images) may be scalars or arrays. The **transposed**, **sample**, and **border** keywords do not apply to them. They are treated as normal **in** or **out** parameters as in Ada.

Adapt Type	C Type
signed_byte	**char**
unsigned_byte	**unsigned char**
byte	**unsigned char**
signed_integer	**int**
unsigned_integer	**unsigned int**
integer	**int**
float	**float**
long_float	**double**

Table 1. Correspondence between Adapt and C types

Parameters of type **image** are two-dimensional arrays. Their size is not declared within the Adapt program; it is not determined, in fact, until the Adapt program is called. Each Adapt program is passed two extra parameters, which are the height and width of the "iteration" image. These height and width parameters are used to determine the height and width of all of the images in the program.

This is done by multiplying the iteration height and width by the sample rate (if the image is of type **in**), or by the size of the array in the **array** parameter (if the image is of type **out**). If neither of these parameters is supplied, then the image height and width are the same as the iteration height and width.

For example, suppose the iteration size is 480×512, and there are three images: in1, an input image with a sample rate of (1,2); in2, an input image with a sample rate of (2,2); out1, an output image with no **array** parameter; and out2, an output image with an **array** parameter of (-2..1,0..1). Then the actual size of in1, in2, out1, and out2 are 480×1024, 960×1024, 480×512, and 1920×1024.

Transposed images are read in transposed order; that is, when (row,col) of a non-transposed image is being processed, (col,row) of a transposed image will be processed. This means that the image size is also transposed: that is, if in1 in the previous paragraph had been declared transposed, it would be of size 512×480. Currently, **transposed** is implemented only for input images.

The **sample** parameter states that processing is not to be done at every pixel of the image, but after skipping a certain number of pixels. For example, in2 in the previous example would be processed at pixels (0,0), (0,2), (0,4), ...,(2,0), (2,2), etc. This allows images to be reduced in size (sub-sampled) in Adapt programs.

The meaning of **array** changes depending on whether an input or an output image is involved. For input images, **array** states that a "window" of the input image is processed; the window is centered around the (0,0) pixel in the window. The corresponding parameter within **first** and **next** is referred to with array subscripts, and any pixel within that window can be accessed. This allows the Adapt program to perform operations that depend on a small local region surrounding the current pixel, for example, edge detection operations. If the **array** parameter of in1 was (-1..1, -1..1), for example, then the situation would be as shown in Figure 3. The first window to be processed would be centered at (0,0); pixels from (-1,-1) to (1,1) could be accessed (the pixels "off the edge" of the image would be filled in by the **border** parameter, if present.) The second window to be processed would be centered at (0,2) (because of the sample parameter); the pixels that could be accessed in this iteration would run from (-1,1) to (1,3)—note that pixels from (-1,1) to (1,1) could be accessed within both windows.

For output images, **array** implies that the image is being expanded. The corresponding parameter in **first** and **next** is an array parameter. These output arrays tile the output image, without overlapping (unlike input images). The situation for out2 is shown in Figure 4. The first iteration produces pixels (0,0) to (3,1), which are referred within the Adapt program as pixels (-2..1,0..1). The second iteration produces pixels (0,2) through (3,3). Note that the base offset of the range in the **array** parameters makes no difference in the image produced.

The **border** value describes what to do when windows fall off the edge of the image (for input images only), as in Figure 3. **Border mirrored** means that image pixels are "mirrored" around the edge of the image; i.e., pixel (-1,0) has the same value as pixel (0,0), pixel (-2,0) has the same value as pixel (1,0), and so on. **Border replicated** means that the first and last rows and columns of the image are replicated, so that pixels (-1,0), (-2,0), (-3,0), etc. have the same

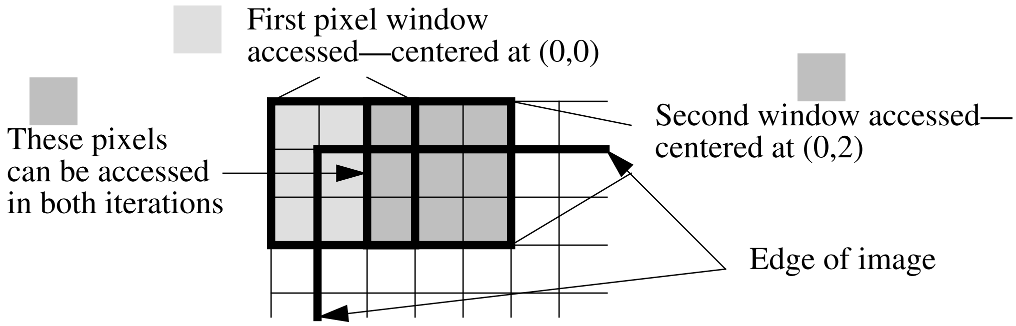

Figure 3. Window and sample characteristics for in1.

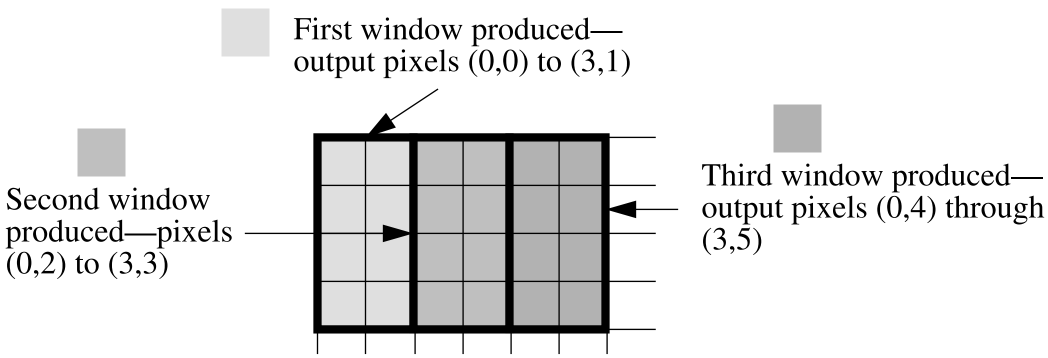

Figure 4. Array characteristics for out2.

value as pixel (0,0). **Border** <value> uses the given constant value for pixels outside the image.

I.3 Ordinary procedures and functions

Ordinary procedures and functions are like Ada procedures and functions, except that Adapt does not implement Ada's default parameter values or named parameters. The syntax is shown in Figure 5.

```
procedure program(parameters) is
  declarations
begin
  statements
end [program];
```

```
function program(parameters) return scalar-type is
  declarations
begin
  statements
end [program];
```

Figure 5. Adapt ordinary procedure and function syntax

Procedures and functions may also be declared in Adapt programs without providing the body of the procedure or function. This results in Adapt assuming that the procedure or function will be provided when the program files are linked. This makes it possible to include calls to procedures and functions written in languages other than Adapt, for example C, within Adapt programs.

Parameter syntax for ordinary procedures and functions is shown in Figure 6.

```
parameters ::= parameter {, parameters}                                  1

parameter ::= variable {, variable} : {in|in out}
     [array(range, range) of]
          {signed_byte | unsigned_byte | byte | signed_integer | unsigned_integer |
                                   integer | float | long_float]         2
```

Figure 6. Ordinary procedures and function parameter syntax.

Ordinary procedures in Adapt are based very closely on C procedures. This leads to a number of restrictions:

- Images cannot be passed as parameters to ordinary procedures and functions, except as fully-referenced scalars. To pass the current window of an image as an array to a procedure or function, copy it to an array first, and pass that.

- Ordinary procedure can have parameters of type **in** and **in out** only; **out** parameters are not allowed. As in Ada, functions can take parameters only of type **in**.

- Adapt does not do type checking of procedure or function arguments. It is possible, for example, to pass an array as an argument for a scalar parameter.

- Although Adapt translates its procedure and function calls into corresponding C procedure and function calls, it does keep track of the types of parameters to procedures and functions, unlike C. It is therefore unnecessary to cast parameters to the correct type except for **in out** parameters.

I.4 Declaration syntax

Variables within a procedure or function (ordinary or otherwise) are declared with a sequence of ordinary variable declaration statements (separated by semicolons), using the syntax of Section I.2.1, except that the **in**, **out**, **border**, and **sample** keywords do not apply.

Variables declared between the first **is** and the initial section of the program can be accessed within any section; they retain their values between accesses. Variables declared between a section name and its **begin** can be accessed only within that section. If the section is **next** or **combine**, those variables retain their values between executions.

I.5. Statements

The statements allowed in procedures and functions are shown in Figure 7. The correspond in form and semantics to a subset of Ada's statements. The semantics will be briefly explained here; they are intended to correspond to Ada's semantics as closely as possible, and any deviation should be reported to the Adapt maintainers.

Assignment has the same semantics as in Ada, and, indeed, in most programming languages; a single memory location is modified, and the expression on the right hand side is completely evaluated before the assignment is made. The **if** statement is also as in Ada.

The loop statement is implemented as in Ada. Note that in Ada, the variable in a **for** iteration clause is implicitly declared within the context of the loop. It does not retain its declaration outside the loop. Therefore, if a variable of the same name is declared in the Adapt program, this variable cannot be accessed within the loop, but retains its value after the loop.

The **exit** statement allows premature exit from a loop. If no loop identifier is given, the nearest enclosing loop is terminated immediately if the boolean expression is true; with a loop identifier, the named loop is terminated. Note that

```
statement ::= null_statement
            | assignment_statement
            | if_statement
            | loop_statement
            | exit_statement
            | case_statement
            | procedure_call                                        1

null_statement ::=   ;                                              2

assignment_statement ::=    variable := expression;                 3

if_statement ::=     if boolean then statements
                       {elsif boolean then statements}
                       [else statements]
                     end if;                                        4

loop_statement ::=   [loop_identifier:]
                   [iteration_clause] loop statements end loop;     5
loop_identifier ::=   identifier                                    6
iteration_clause ::=    for simple-variable in [reverse] range
                     |  while boolean                               7

exit_statement ::=    exit [loop_identifier] when boolean;          8

case_statement ::=    case expression is
                      {when choice {| choice} => statements}
                      end case;                                     9

procedure_call ::= procedure(expression{,expression})              10
```

Figure 7. Adapt Statement Syntax

exit can be used only within a loop, and the loop identifier the name of an enclosing loop.

The **case** statement is just as in Ada. The expression given must yield an integer value, and the choices listed must be constant integer expressions. Note that the character "|" separates the choices in the list of choices; this is not a meta-character. The choice **others** may be given; it gives a default case if no choice is satisfied. Note that, as in Ada, **others** must be given last if it is given at all.

I.6 Expressions

Adapt expression syntax is shown in Figure 8. The syntax and semantics are taken from Ada. The **land**, **lor, lxor, lnot**, **lsh**, and **rsh** operators (which are not in Ada) are used to manipulate integers and bytes as strings of bits: they stand for bit-wise and, or, exclusive or, not, left shift, and right shift, respectively. Note that, as in Ada, a boolean expression cannot include both **and** and **or** in the same sequence; parentheses must be used. Adapt implements Ada's **and then** and **or else** logical operators, which evaluate their right-hand argument only as necessary.

A type name can be used to convert an expression to that type. For example, **float(i)** converts i to floating point type.

Identifiers in Adapt programs may include letters, the underscore, and numbers, as in Ada. An underscore cannot appear as the first or last character of a name. Underscore is itself an operator in Adapt programs; within the **combine** section of a program, it denotes variables that are results from the lower region that is being combined.

Adapt operator precedence is shown in Figure 9. It is taken from Ada, and is slightly different from C operator precedence.

Certain Adapt identifiers have special values. The variables **row** and **col** refer to the current row and column being

```
expression ::=   expression operator expression
               | (expression)
               | simple_expression
               | array_reference
               | type_name(expression)
               | function(expression{,expression})
               | type_name(expression)                              1

operator ::=  + | - | * | / | ** | mod | rem                        2
          | land | lor | lxor | lnot | lsh | rsh                    3

simple_expression ::= identifier | integer | float | 'char'         4
identifier ::=  [a-zA-Z][a-zA-Z_0-9]*                               5

array_reference ::= simple_variable(expression {,expression})       6

boolean_expression ::=     conjunction
                       |   disjunction
                       |   simple_boolean_expression
                       | not boolean_expression                     7
                       | (boolean_expression)                       8
simple_boolean_expression ::=     expression > expression
                              |   expression < expression
                              |   expression = expression
                              |   expression /= expression
                              |   expression <= expression
                              |   expression >= expression         9
conjunction ::=   simple_boolean_expression and conjunction
               | simple_boolean_expression and then conjunction
               |  simple_boolean_expression                        10
disjunction ::=   simple_boolean_expression or disjunction
               |  simple_boolean_expression or else disjunction
               |  simple_boolean_expression                        11
```

Figure 8. Adapt Expression Syntax.

```
or xor                  1
lor land lxor           2
= /= < <= > >=          3
lsh rsh                 4
+ - not lnot            5
* / mod rem             6
**                      7
_                       8
```

Figure 9. Adapt Operator Precedence.

processed (in the iteration image). The variables **rows** and **cols** refer to the size of the iteration image. **Rows** and **cols** cannot be used in integer expression to declare arrays.