APPLY: Machine Independent Image Processing Language and its Implementation on a Meiko Computing Surface

Peter M Dew [1], Han Wang [1] and Jon A Webb [2]

[1] School of Computer Studies, The University of Leeds, Leeds, UK

[2] Department of Computer Science, Carnegie Mellon University, Pittsburgh, USA

APPLY is a machine independent, low level image processing language for expressing local window operations. It has two main advantages: (1) it significantly reduces the programming effort, and (2) it can be mapped onto a wide range of parallel computers. In this paper, we report an our recent experience on implementing APPLY on a Meiko Computing Surface (Transputer array machine) using a farmer/gather model. The performance of Meiko implementation on a number of edge detection algorithms including the popular Canny operator is analysed.

1. Introduction

Low level image to image processing is an important component of a computer vision system. There is now a wide range of equipments for performing these computations in real and reasonable time and the major problem is porting software between these machines. APPLY is machine independent image processing language for expressing local windowing operations, such as the Sobel edge detector. The vision researchers express the image processing computations in APPLY and these codes are then automatically mapped on particular image processing equipment. Another advantage of the APPLY approach is that it is easy to extract parallel tasks and implement the computation on a processor array machine such as a Meiko Computing Surface.

The purpose of this paper is to outline the APPLY for edge detection algorithms and discuss its implementation on a Transputer array machine using a farmer/gather model. The Canny Edge Detector has been implemented in APPLY and results suggest that it runs faster than hand coded algorithms reported in the literature. Finally, the APPLY language is compared with LATIN [Crookes1987].

2. Low Level Image Processing

Let I_{in} denote the input image and I_{out} denote the output image, low level local windowing image processing can be defined as the mapping: $W : I_{in} \rightarrow I_{out}$, where the function W is amenable to output data partition [Dew1988]. That is,

$$I^i_{out} = W(I^i_{in}), \text{ where } I_{out} = \bigcup_i I^i_{out} \quad \text{and} \quad I_{in} = \bigcup_i I^i_{in}$$

The abstract machine model for supporting output data partitioning is the farmer/gather model (or processor farm) shown in Figure 1. The abstract machine model can be mapped into a wide range of parallel and serial computers [Sleigh1988].

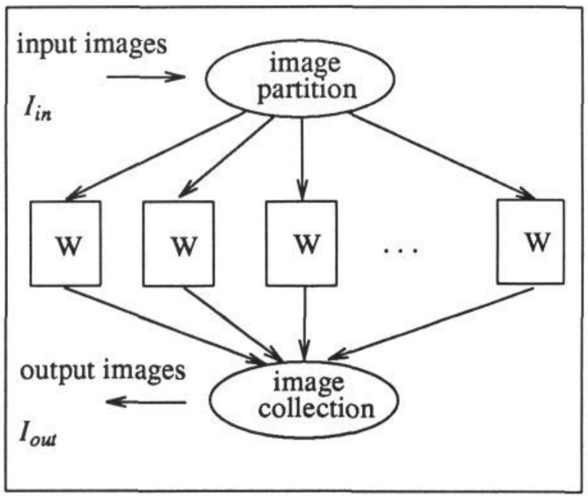

Figure 1. The Abstract Machine Model for Local Windowing Image Processing

3. The APPLY Language

The APPLY language was designed to automate the process of low level image processing [Wallace1988]. It provides the programmer with a machine independent interface for local window image-to-image computations. The user specifies the algorithm by the language and the compiler and its supporting environment will translate and map this algorithm into the physical machine. It has two main advantages: (1) It reduces the programming effort by hiding image handling routines and the machine architecture, and (2) It is machine independent. It uses the abstract machine model shown in Figure 1 and generates automatically the object codes for the worker processors. So far, it has been implemented on the Warp [Annaratone1987], Hughes HBA, UNIX machines and the

Meiko Computing Surface.

APPLY has an ADA-like syntax. It defines a local window operation for each input image, conducts an arbitrary computation over the window, and outputs the processed image. For example, the Sobel edge detector is defined as two template convolutions, the horizontal template (horz) and the vertical template (vert):

$$\text{horz}=\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline & & \\ \hline -1 & -2 & -1 \\ \hline\end{array} \qquad \text{vert}=\begin{array}{|c|c|c|}\hline -1 & & 1 \\ \hline -2 & & 2 \\ \hline -1 & & 1 \\ \hline\end{array}$$

where the output is a scalar value, either $\sqrt{horz^2+vert^2}$ or $|horz| + |vert|$. The Sobel edge detector expressed in APPLY is shown below.

```
procedure sobel(
  imagein : in array (-1..1, -1..1)
            of byte border mirrored,
  switch  : const integer,
  imageout: out byte)
is
   horz, vert : integer;
begin
  horz := imagein(-1,-1)+2*imagein(-1,0)
          +imagein(-1,1)-imagein(1,-1)
          -2*imagein(1,0)-imagein(1,1);
  horz := abs(horz);
  vert := imagein(-1,-1)+2*imagein(0,-1)
          +imagein(1,-1)-imagein(-1,1)
          -2*imagein(0,1)-imagein(1,1);
  vert := abs(vert);
  if switch = 1 then
     imageout := integer(sqrt(horz*horz
                 + vert*vert));
  else imageout := horz + vert;
  end if;
end sobel;
```

The procedure argument of `sobel` specifies the 3x3 input window, `imagein` and the output window elment, `imageout`. Each element of the window is of type byte and the elements of the window are referred in the normal array notation. The parameter `switch` is used to select the value of the output window element `imageout`. APPLY applies the input window to all pixels of the image and the phrase `border mirrored` signifies that border pixels for input images are obtained by mirroring.

The compiler generates object codes for the host that sends and gathers images and for worker processors that conduct local window operations. The object codes generated by the compiler is machine dependent. For example, three languages are supported: OCCAM, C, and W2. In a worker processor, the computation has four steps: get row, compute, put row and update the input buffer. Only one window buffer is declared for each input image. The window operation is overlapped with the I/O. It is possible to improve an hand coding, for example, a cyclic scroll buffer is used which updates the input buffer under the cost of only one addition.

The object code will be loaded into the machine for execution together with the APPLY supporting environment which includes the system configuration information, the networking library, the image I/O library and the user interface.

4. APPLY Implementation on the Meiko Computing Surface

Experiments have been conducted on the Meiko Computing Surface using the parallel one-dimensional array (PODA). Figure 2 depicts the topology of PODA. "H" represents the host and "W" the worker processor.

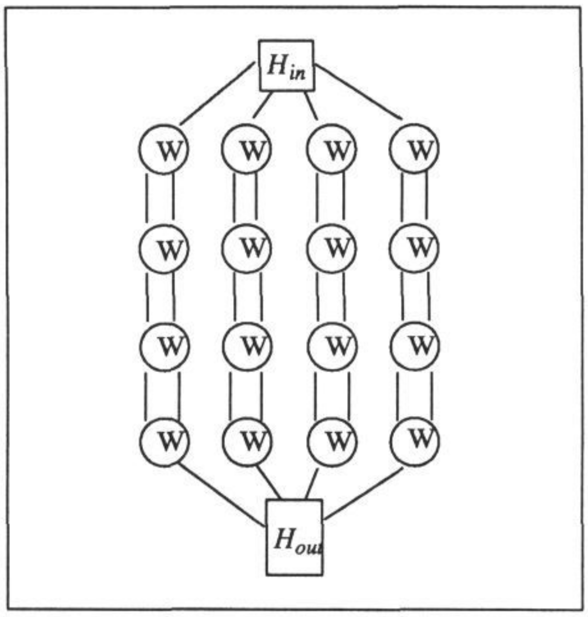

Figure 2. Parallel One Dimensional Array(PODA)

In the PODA approach, the H_{in} process will receive images from the disc file and deliver them to the worker processors line by line. Each line is divided into N equal sections. Then these sections will be packed into data packets and sent down to the named processors. The H_{out} process collects the results from the worker processors and puts them back to the disc.

5. Performance of the Edge Detection Algorithms

One advantage of APPLY is that it is easy to compare the computational time for a range of local window operation algorithms. For example, two algorithms, "egpr" (edge preserving smoothing) and "egsb1" (Sobel edge detector) were chosen to investigate limitations of the network I/O capacity. The algorithm "egpr" is compute bound relating to "egsb1". The result for 8, 32 and 64 Transputers are shown in Figure 3. It can clearly be seen that there is not much advantage in exceeding 32 Transputers.

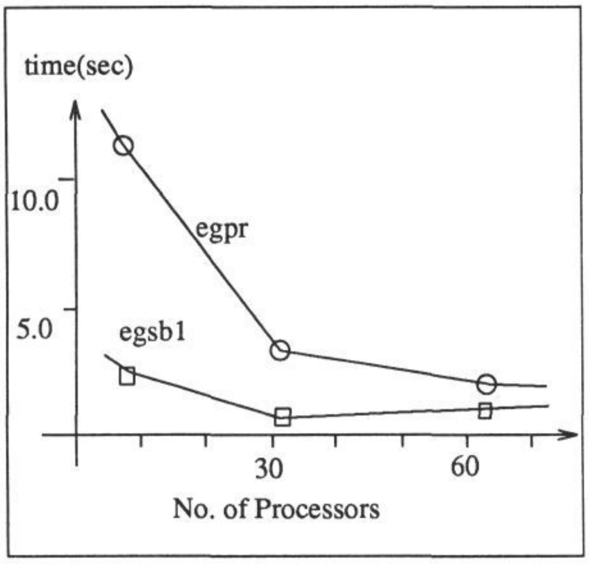

Figure 3. Performance of "egpr" & "egsb1"

6. Extension of APPLY to Pipelined Multiple Window Operations

Algorithms in the previous section all can be expressed by a single window operation. However, image processing algorithms often contain more than one window operation. For example, the Canny edge detector has four steps: (1) Gaussian smoothing, (2) 2D differentiation, (3) non-maxima suppression and (4) hysteresis (threshold). In the previous APPLY version, these steps of Canny operator would have to be compiled and executed individually, because the compiler can handle only one procedure at a time. It is an expensive process. For example, the procedure that computes the differentiation will receive the Gaussian smoothed image and output three images of gradients in X and Y dimensions and the magnitude which is the sum of absolute value of X gradient and Y gradient. The communications on the network involves input of one real image (4 bytes per pixel) and output of three real images (two for the gradients and one for the magnitude) which are then saved in as the disc files ready for the input of the next procedure. Then gradients and magnitudes will have to be loaded from the disc again for the input of the next procedure which is the non-maxima suppression. It is clear that intermediate results will have to be moved along the network of processor array twice. This is not practical particularly for Transputer arrays since they have relatively slow I/O links. To deliver one real image from the H_{in} process by a single Transputer would take 839 milliseconds in theory. In practice, the time for sending a real image is three times higher.

Experiments have been performed at Leeds to extend APPLY to the pipelined multiple window operation to solve this problem. This technique requires several window operations (procedures) to reside concurrently in one worker processor. One soft channel is used between two adjacent procedures. The first procedure receives inputs from the host and outputs intermediate results to the second one; the last procedure will input from previous one and output final results back to the host through the network. Advantages have gained from this improvement:

(1) Saving the Intermediate Results Moving Along the Network
Intermediate results are confined inside worker processor instead of outputting to the disc and loading in again. They move only along soft channels between two procedures. The speed of the soft channel is the bandwidth of memory copying, which it is 30 times faster than the hard link.

(2) Balancing Load for Worker Processors
Procedures in one algorithm are computationally uneven. For instance, the 2D differentiation takes three adds for one pixel, while the Gaussian smoothing takes 9 multiplications. In the implementation of the Transputer array, the former is I/O bound and the latter is compute bound. For the I/O bound procedure, if it is executed individually, the performance drops for the worker processor will have to wait when it finished computing one line until it gets the next input. In the pipelined implementation, there is no time wasted in waiting the input from the host, because I/O bound procedures act as FIFO data buffers.

Figure 4 depicts a pipeline chart of the Canny edge operator. All these three procedures stay in one worker processor.

The execution time of the pipelined Canny edge detector on the Meiko with 32 T800 processors is 0.86 seconds (0.66 seconds without I/O), which is a speed-up of more than 2 times compared with individually compiled procedures. The image is of size 512x512. The time is measured from when the data is ready at the host H_{in} to the moment that data are collected at the host H_{out}. The time of the Gaussian smoothing is not included.

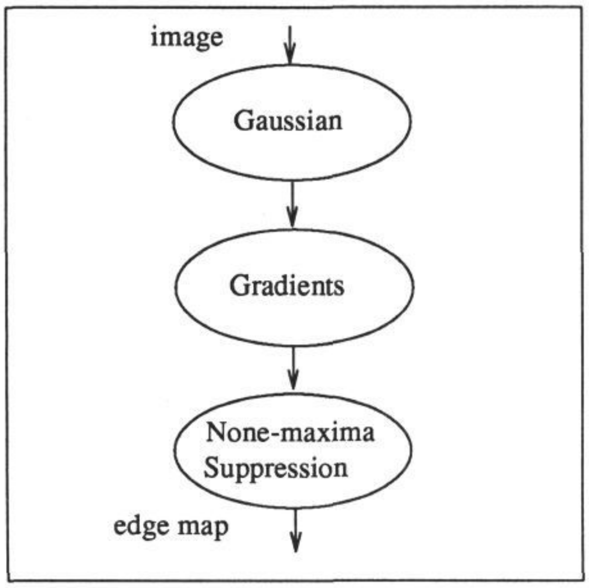

Figure 4. Pipeline Implementation of the Canny Operator

7. APPLY compared with LATIN

LATIN and APPLY satisfy different application requirements. The former is a general purpose parallel language and the latter is an application oriented language. However, they do share some similarities, for example, algorithms in LATIN may have multiple processes, APPLY can deal with multiple window operations as well; data path is transparent in LATIN, and APPLY has hidden network details to users. However, These two languages are quite different in some other aspects. Firstly, LATIN allows the inter-processor communication, but APPLY does not. The abstract APPLY machine model has defined that this is not necessary for localised window operations. Secondly, in LATIN, the programmer will have to specify the system configuration. This means when an application has more than one process, LATIN demands more processors than APPLY does. Finally, the implementation of APPLY has been concentrated on the efficiency of window operations, for instance, the window buffering and border handling techniques. Users can put efforts in image processing algorithms, rather than work on technical details, eg. networking and deadlock. So APPLY is more suitable for image processing.

8. Discussion

APPLY provides a programming tool for low level local windowing image processing, and it can also be used in real time. The pipelined implementation on the Meiko Computing Surface extended APPLY to deal with multiple window operations. However, APPLY cannot cope with data dependent algorithms, for example, the hysteresis and the Hough transform. This is because the abstract APPLY machine model is based on data parallelism which permits only window operations. Future work includes studying of new computational models.

REFERENCES

Annaratone1987.
M Annaratone, E Amould, T Gross, H T Kung, M Lam, O Menzilcioglu, and J A Webb, "The Warp Computer: Architecture, Implementation and Performance," *IEEE Trans. on Computers* C-**36,12** pp. 1523-1538 (Dec 1987).

Crookes1987.
D Crookes, P J Morrow, P Milligan, N S Scott, and P L Kilpatrick, "Notes on Implementing a Language for Transputer Networks," *Microprocessing and Microprogramming* **21** pp. 559-566 North-Holland, (1987).

Dew1988.
P M Dew and H Wang, "Data Parallelism and the Processor Farm Model for Image Processing and Synthesis on a Transputer Array," *Proceedings of SPIE Symposium* **977** pp. 212-220 (August 1988). Real Time Signal Processing XI, Society of Photo-Optical Instrumentation Engineers

Sleigh1988.
A C Sleigh, C J Radford, and G J Harp, "RSRE Experience Implementing Computer Vision Algorithms on Transputers, DAP and DIPOD Parallel Processors," pp. 133-154 in *Parallel Architectures and Computer Vision*, ed. I Page,Oxford University Press (1988).

Wallace1988.
R S Wallace, J A Webb , and I C Wu, "Machine-independent Image Processing: Performance of APPLY on Diverse Architectures," *Presented at the Third International Conference on Supercomputing*, (May 1988).

Acknowledgements The authors would like to thank Professor H T Kung, the Edinburgh Concurrent Supercomputer Centre and the MEIKO Limited for their support and allowing access to the frame grabber. Thanks also goes to Ms. J Ryan for her discussion on the early draft version of this paper.