0.95

IMPLEMENTING THE HOUGH TRANSFORM ON A DISTRIBUTED MEMORY MULTIPROCESSOR USING A PARALLEL FUNCTIONAL LANGUAGE

A. Underhill, M. Atiquzzaman and J. Ophel

Department of Computer Science and Computer Engineering,
La Trobe University, Bundoora, Vic. 3083, Australia.
{*underhil* | *atiq* | *johno* }*@latcs1.lat.oz.au*

ABSTRACT

The Hough Transform is a powerful tool for the recognition of patterns in images. However, its high computational cost both in terms of time and storage requirements restricts its usefulness on single processor machines. Multiprocessor implementations of the algorithm are therefore attractive. We have investigated algorithms for use on the Fujitsu AP1000 distributed memory multiprocessor, and implemented them using a 'mostly' functional language called *paraML*. Reasonable speedup results have been achieved for the Hough calculation using up to 15 processors, but results for larger numbers of processors are disappointing. We anticipate that our results could be improved by adding a broadcast facility to *paraML* to make use of the AP1000's broadcast network since communication time seems to be a major factor in our algorithms. Although there appear to be some limitations with *paraML*, its functional nature, ease of use and high-level methods of controlling parallelism suggest that this style of language could have a part to play in the programming of the highly parallel computers of the future.

1. INTRODUCTION

The detection of lines, circles and other parameterizable curves in images is a common task in intermediate level image processing. The Hough transform is a commonly used technique which allows one to extract the parameters of such curves. Real-time detection of parameterizable patterns is highly desirable in many image processing and computer or robot vision tasks. The importance of the Hough transform lies in the fact that it is relatively unaffected by noise in the image and discontinuities in edges, and it can cope well with occluded objects. The major disadvantage of the Hough transform is that it requires an enormous amout of computation and a large amount of storage capacity. Consequently, the Hough transform has been found to be an excellent candidate for multiprocessor implementation. We have a number of aims in this work:

- To investigate algorithms to partition the standard Hough transform for implementation on an MIMD (Multiple Instruction Multiple Data) machine.
- To implement those algorithms on the Fujitsu AP1000, a highly parallel, scalable, distributed memory computer consisting of three separate communication networks.

- To investigate the use of *paraML* [?], a parallel functional language designed for the AP1000 style of multiprocessor.

Our results show that we are able to gain reasonable speedup for one of our algorithms using up to 15 processors. We also comment on how the clarity, expressiveness and simplicity of paraML enable parallel programs to be written in a simple and effective manner.

1.1 The Hough Transform

The Hough transform is a mapping from an image plane to a parameter space. The transform turns the problem of finding patterns in an image plane to a problem of determining local maxima in a transform parameter space. Every feature point in the image plane votes for all the possible patterns passing through that point. These votes are accumulated in an accumulator array in the parameter space. The parameters of the patterns receiving the maximum votes are determined by locating peaks in the accumulator array.

This paper restricts its attention to the problem of finding straight lines only, which are most conveniently parameterized using the two parameters ρ and θ. Thus, transforming each feature point (x, y) in the image involves computing

$$\rho = x\cos(\theta) + y\sin(\theta), \qquad 0 \le \theta < \pi$$

in steps of π/θ_{res}, where θ_{res} is the desired resolution for the θ index of the accumulator array. The ρ index ranges from $-\sqrt{2}N$ to $\sqrt{2}N$ for an image of size N x N. In the case of a straight line, each feature point in the image defines a sinusoidal curve in the accumulator array. The peak in the accumulator array is located at the point where the largest number of these curves intersect, and defines the required parameter values of the line in question. When searching for multiple lines there will be a number of peaks, each of which defines a different line. In reality, there will also be noise and quantization errors which mean that a peak may not be well defined due to it being spread out over some larger area of the accumulator array. These cases require a more sophisticated approach than simply finding local maximas.

1.2 Parallel Implementations of the Hough Transform

The major disadvantage of the Hough technique is that it is highly compute intensive and requires a large storage capacity. In fact storage increases exponentially as the number of parameters increase and is of $O(m^p)$, where p is the number of parameters and m is the resolution of each parameter axis. Many methods have been proposed to limit both time and storage requirements [?]. The major advantage in implementing the Hough transform in a multiprocessor system is that each feature point in the original image can in fact be treated completely independently. Thus, parallel processing of all the points is possible. This inherent parallelism has attracted much attention. Implementations using various algorithms and architectures have been reported. Algorithms suitable for SIMD mesh architectures are described by Silberberg on the GAPP [?], and in Guerra & Hambrusch [?]. Kannan & Chuang use a mesh connected torus array [?], and Krishnaswamy et al. use a SIMD model on a Connection Machine [?]. Hierarchical variations suitable for pyramid architectures have been proposed by Princen et al. [?], Jolion & Rosenfeld [?], and Bongiovanni et al. [?]. A Hough transform algorithm is used to demonstrate a pyramid programming environment on a connection machine in [?]. Atiquzzaman has suggested a multiresolution algorithm [?] and its implementation on a pyramid architecture [?]. Pan & Chuang give three algorithms for SIMD hypercube architectures [?], while other hypercube implementations are discussed in [?, ?, ?]. Schemes for shared memory MIMD computers are described in both Choudhary & Ponnusamy [?], and Thazhuthaveetil & Shah [?], while Eghtesadi & Sandler have used a transputer network [?].

Section 2 describes the AP1000 architecture while Section 3 examines the *paraML* language. Section 4 describes our algorithms and implementation strategies, Section 5 gives some results and observations, before our conclusions in Section 6.

2. AP1000 ARCHITECTURE

The AP1000 [?] is a highly parallel, distributed memory computer - an MIMD machine. Although a major design goal was to create a machine which would execute fine-grain data-parallel programs efficiently [?], it has been found to be effective for both fine-grain and coarse-grain applications. It is a scalable machine with the ability to contain between 16 and 1024 separate processing elements (PE's) called cells. The cells are connected by three separate communication networks (see Figure 1). A two dimensional mesh connected Torus network (T-net) is used for point-to-point communication between cells. A Broadcast network (B-net) is used for 1-to-N communications, and a Synchronization network (S-net) is used for barrier synchronization. The cells are connected to a Sun-4/390 which acts as a front-end host to the AP1000. The machine used in this study contains 128 cells. Each of these cells contains a SPARC CPU, a floating point unit, message and routing controllers, and 16MB of memory.

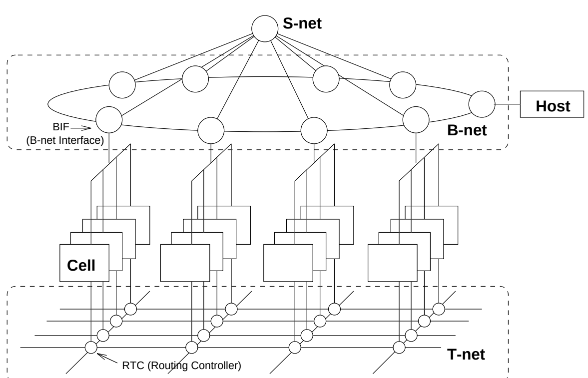

Figure 1: AP1000 System Configuration. (From [?]).

Major design goals for the AP1000 were to achieve a high throughput, low latency communication network; fast message handling; fast barrier synchronization, and efficient data distribution and collection. It is claimed [?] that the length of the communication path has a negligible effect on the communication time, and that communication time can be described as a linear function of message size. *ParaML* utilizes this fact by allowing the user to program in a way that lets them ignore the actual physical placement of processes on processors. This task is taken care of by the compiler.

3. PARAML

ParaML [**?**] is an extension to the ML language, a functional language with strong polymorphic typing that has attracted considerable attention in the programming language community. ML's advantages stem both from its functional nature, which encourages the writing of reliable, error-free programs, and from its type system, which provides the safety of static type checking with the flexibility of polymorphic types [**?**].

3.1 Programming Model

The *ParaML* extension has been specifically designed for programming the highly parallel computers of the future such as the AP1000. It aims to satisfy the requirements of a parallel language whilst retaining a functional style and all the inherent advantages of ML. Other functional languages [**?**] rely on implicit fine-grain parallelism, such as for the parallel execution of multiple arguments of a function call. This technique however, is widely viewed to be ineffective [**?**]. In contrast, *paraML* emphasizes explicit coarse-grain parallelism as the programmer has the knowledge to determine whether the speed increase offered by parallel execution outweighs the overhead of process creation in a given algorithm.

Consistent with this coarse-grained model of parallelism, there is no global memory and processes communicate with each other using message passing to typed channels. A channel can be viewed as a port associated with a particular process to which other processes send messages. A process can only get messages from its own channel, but can send messages to any channel associated with any other process provided it knows the actual channel name for the particular process it wishes to send to.

A *paraML* program consists of a top-level user function which runs on one cell, and a series of function definitions each of which uses the *create* or *vectorcreate* primitive to define a particular process form. Instances of these processes are invoked by calling these functions. Returned values from processes can be gained by using the *result* primitive which takes a process name as an argument and blocks until that process has finished and returned its result. During execution, data is passed between processes by using the *send* and *get* primitives.

The Hough transform is suited to this style of programming model since it requires explicit partitioning of the data and computation, yet can utilize the very low-latency cell-to-cell communication provided by the AP1000 for transmission of data between cells.

4. IMPLEMENTATION

In implementing a Hough transform algorithm suitable for use on a multiprocessor, one has to determine appropriate partitioning schemes for the image, the accumulator array and the computation amongst the individual cells. The standard

Hough transform approach consists of two main steps: first, the accumulation of votes in the accumulator array for each feature point in the image; and secondly, determination of the peak in order to find the required parameters. Two different partitioning schemes are described below. We assume a binary image, with 1's representing the feature or edge points of interest.

4.1 Method 1 : Distributing the Image

In this first method, the edge points of the image are distributed evenly among the cells by a top-level master function which runs on a single cell. Each cell contains its own accumulator array, and uses its own set of edge points to calculate the entire accumulator array. The master function carries out an initial preprocessing step by scanning through the image and counting the number of edge points, hence allowing the computational load to be balanced by sending approximately the same number of edge points to each cell. After the Hough calculation there exists a separate accumulator array in each cell and these must be combined together before the global peak can be found. This combining process is done in a binary tree fashion with one cell sending its accumulator to a neighbour, where it is added to its own accumulator. This combining process requires n stages, where there are 2^n cells. The peak finding process is then carried out on the single master cell.

The expected benefit of this method is a fast Hough calculation because of the relatively small number of edges on each cell, while the drawback is the time taken for the combining of the accumulators. The combining step also results in a low processor utilization whilst the combining takes place, since after the first combining stage, half of the cells are idling.

4.2 Method 2 : Distributing the Accumulator Array

Our second method divides the accumulator array amongst the cells. The entire set of feature points is sent to each cell while the accumulator is split among the cells in the θ dimension only so that each has its own range of θ values to compute. Using this scheme, there are two possibilities for the peak finding stage of the algorithm. Either each subportion of the accumulator can be sent back to the master process so that the peak can be found based on the entire accumulator array (as in method 1 above), or part of the peak finding work can be done within each cell by finding the peak in the local accumulator array and sending the local peaks to the master. The master then simply has the task of finding the maximum of the received local peaks. The latter method has been used in the results which follow. This works for finding only single lines but requires a slightly more sophisticated algorithm where multiple peaks (i.e. multiple lines) are to be detected. The expected benefit of this method is excellent processor utilization due to a considerable amount of the peak finding work being done in a distributed manner and the lack of the need to combine the separate accumulator arrays as in method 1.

ParaML is not a purely functional language because it does support some imperative features. In our implementations we make use of ML's updatable arrays as well as its powerful list abilities. In both of the above methods, the edge points are extracted from the image and stored as a list of coordinate pairs before being distributed. In method 2, the whole list is sent to every cell while in method 1 a different sublist is sent to each.

5. RESULTS

The results used here are based on images of size 64 x 64 with the number of feature points ranging from 64 to 1024. Speedup measurements refer to the time just before the feature points are sent to the cells until after the peak is found. Due to the nature of the prototype implementation of *paraML* there were some difficulties during development of the programs. An example is the failure of large messages to arrive properly at their destinations.

A number of observations shaped the development of the implementations. First, the repeated calculation of *sine* and *cosine* values is redundant and very expensive. Thus the required *sine* and *cosine* values were stored in a lookup table to speedup the calculation of ρ considerably. The storing of edge points as lists of coordinate pairs minimises the amount of communication when sending the image to the cells since only the coordinates of the feature points are sent. Similarly, the Hough calculation time is shortened since it is not necessary to scan through the image when calculating the ρ values. Although ML's strong typing is helpful when writing a program, it was found that if the image was stored as a list of integer coordinates, then it was necessary to convert the coordinate values into reals before being able to carry out a multiplication with a real number, since the operands must be of the same type. The calculation time for ρ could be halved if the feature points were already stored as a list of real coordinates. Thus the conversion to reals was done at the master process, and the list of feature points was sent as reals. However, the drawback with this approach is that sending a list of reals takes approximately twice as long as a list of integers.

figure=method2.2.ps,width=7.5cm

Figure 3: Speedup for method 2.

Figure 3 shows the speedup gained for method 2 is almost linear up to 4 processors, ranges from 8 to 10 for 15 processors, and decreases after that. The total time T_2 for method 2 is given by:

$$T_2 = t_{send} + P * t_{Hough} + t_{subpeak} + t_{comm} + t_{peak}$$

where t_{send} is the total time to send all feature points to all cells, $t_{subpeak}$ is for finding the peak of the local accumulator array, t_{comm} is the time to send the subpeak to the master at the same time, and t_{peak} is for the master to get the subpeaks and find the global peak. Figure 4 shows a Gantt chart for 4 processors working on 1024 points. As the number of processors increases, the time to send the feature points to all the processors becomes the dominating factor. The processors therefore spend a long time idling while waiting for their feature points to arrive.

Method 1 failed to show speedup (see Figure 2) because the time to send and combine the accumulator arrays was far greater than the time to calculate ρ. This time increases as the number of processors increases since more combinations are necessary. It also depends on the resolution of each parameter axis of the array. Using N processors for P feature points the total time T_1 is:

figure=method1.2.ps,width=7.5cm

Figure 2: Speedup for method 1.

$$T_1 = t_{send} + \lceil P/N \rceil * t_{Hough} + \log_2 N * t_{combine} + t_{peak}$$

where t_{send} is the time to send the feature points, t_{Hough} is the time to compute ρ for one feature point, $t_{combine}$ is for sending and combining an accumulator array, and t_{peak} is to find the peak in the final accumulator.

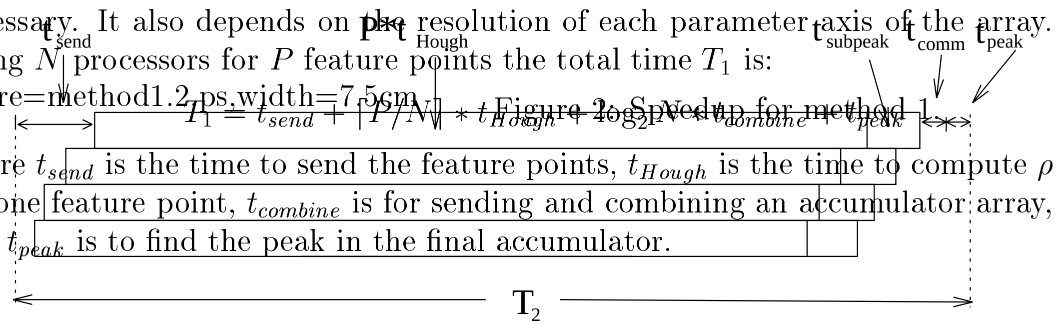

Figure 4: Method 2 timing sequence for 4 processors and 1024 feature points.

6. CONCLUSION

We have investigated the use of the 'mostly' functional language *paraML* for implementing the Hough transform on the AP1000 distributed memory multiprocessor. Although we have shown speedup for the Hough transform using various numbers of processors, it is clear that speedup for our algorithms is good for only a small number of processors, and any increase in the number of processors beyond