ELSEVIER

Microprocessors and Microsystems 22 (1999) 355–362

MICROPROCESSORS AND MICROSYSTEMS

Performance of the Hough transform on a distributed memory multiprocessor

Austin Underhill[a], Mohammed Atiquzzaman[b,*], John Ophel[a]

[a]*School of Computer Science and Computer Engineering, La Trobe University, Bundoora, Victoria 3083, Australia*
[b]*Department of Electrical and Computer Engineering, University of Dayton, Dayton, OH 45469-0226, USA*

Received 6 January 1998; received in revised form 24 July 1998; accepted 6 August 1998

Abstract

The Hough transform is a projection-based transform which can be used to detect shapes in images. One of the disadvantages of the transform is its requirement for large amounts of computing power. Parallel machines have given programmers the potential for incredible computing power. To obtain maximum performance from parallel machines, parallel algorithms should be designed to reflect the architecture of the parallel machine. The work reported in this paper compares the performance obtained in running several parallel versions of the Hough transform on a Fujitsu AP1000 distributed memory multiprocessor. © 1999 Published by Elsevier Science B.V.

Keywords: Hough transform; Distributed memory machine; Performance evaluation

1. Introduction

The advent of parallel machines has given programmers the potential for incredible processing power. Unfortunately, except in a few small domains where vector processing machines have produced good results, the overall impact of parallel machines has been small; having much potential but failing to deliver actual, scalable performance gains. Unlike single processor machines where the dominant architectural paradigm of the von-Neumann machine has given a common starting point, architectures of parallel machines vary greatly from machine to machine. For example, the alternative communication models of shared memory and message passing encourage different algorithms for the same problem. Differences in processing elements, topologies, etc. similarly affect algorithm design.

Large compute-intensive tasks can be partitioned into a number of subtasks, and can be distributed among the different processors [1,2] in a distributed memory multiprocessor system. The speedup obtained from such a system for a particular task depends on the partitioning algorithm and on the ratio of the computation to communication among the subtasks. The aims of this paper are to develop partitioning strategies for the Hough transform, implement the subtasks in a real multiprocessor system, determine the maximum speedup achievable, and show the effects of granularity (ratio of computation to communication) on the speedup.

The Hough transform is used to detect geometric patterns in images and is known to be very compute-intensive. There has been much work carried out on the Hough transform in the last two decades for single processor machines [3]. The nature of the Hough transform algorithm reveals that each feature point in the image can be treated independently. Thus, it is possible to process each of these points concurrently. This inherent parallelism has attracted much attention. Implementations of the transform using various algorithms and architectures have been reported. Algorithms suitable for SIMD mesh architectures are described by Silberberg on the GAPP [4] and Guerra [5]. Kannan and Chuang use a mesh-connected torus array [6], and Krishnaswamy et al. use a SIMD model on a connection machine [7]. Hierarchical variations suitable for pyramid architectures have been proposed by Princen et al. [8] and Jolion et al. [9]. A Hough transform algorithm is used to demonstrate a pyramid programming environment on a connection machine by Sher et al. [10]. Atiquzzaman [11] has suggested a multiresolution algorithm and its implementation on a pyramid architecture [12]. Pan et al. [13] give three algorithms for SIMD hypercube architectures, while other hypercube implementations are discussed by Ranka et al. [14]. Schemes for shared memory MIMD computers are described by Choudhary et al. [15].

Much of the previous work is theoretical in nature and

* Corresponding author.

0141-9331/99/$ - see front matter © 1999 Published by Elsevier Science B.V.
PII: S0141-9331(98)00093-3

relies on ideal theoretical machines. Although these algorithms without implementations have made valuable contributions to the theory, the practical aspects of developing working solutions must not be overlooked.

The objectives of this work are as follows:

- Develop parallel Hough transform algorithms of the Hough transform on the Fujitus AP1000 distributed multiprocessor system.
- Compare the performance of the different algorithms for varying numbers of processors and image sizes on a Fujitsu AP1000, a coarse-grained machine. It is to be mentioned that most previous real implementations are on fine-grained machines.
- Determine the optimal number of processors for which the speedup is maximum on the AP1000.
- Determine the optimal granularity at which the maximum speedup is obtained and to determine the optimum number of processors for maximum speedup.
- Determine the effects of the multicasting feature of the AP1000 on the parallel versions of the Hough transform.

This work attempts to address the issue of implementing the Hough transform on a real machine, the Fujitsu AP1000 multiprocessor. We have implemented four different partitioning strategies and implemented them on a Fujitsu AP1000 multiprocessor system using the paraML functional programming language. Speedup obtained from the parallel algorithms have been measured and compared. Comparisons of speedup using images of varying sizes in each of the parallel algorithms are also reported.

The paper has five main sections. The first describes the important features of the paraML language and the Fujitsu AP1000 distributed memory multiprocessor followed by a brief description of the Hough transform. Various parallel implementations of the Hough transform which have been developed and implemented are discussed along with their performance results. Finally, some conclusions of this work are given.

2. AP1000 Multiprocessor and paraML

We have implemented four versions of the parallel Hough Transform on the AP1000 multiprocessor using paraML. In this section, we describe the architecture of the Fujitsu AP1000 multiprocessor system and functionalities of paraML which are of interest to this work.

2.1. The AP1000

The AP1000 [16], originally known as CAP (cellular array processor), was developed by Fujitsu as a research machine and is thus not yet in commercial production. The AP1000 (see Fig. 1) is a powerful, highly-parallel, distributed memory, scalable computer. It can contain between 16 and 1024 processing elements, each with 16 MB of memory. The machine used in this study (at the Australian National University) contained 128 processing elements. Each processing element is termed a cell. The cells are connected by three separate communication networks. 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 one-to-many communication, and a Synchronization network (S-net) is used for barrier synchronization.

2.2. The paraML language

paraML is an extension of Standard ML [17], allowing parallel computation on a distributed memory multiprocessor. The general-purpose programming language SML is a mostly functional language with a sophisticated type system. paraML emphasizes explicit coarse-grain parallelism, where programming at the top level amounts

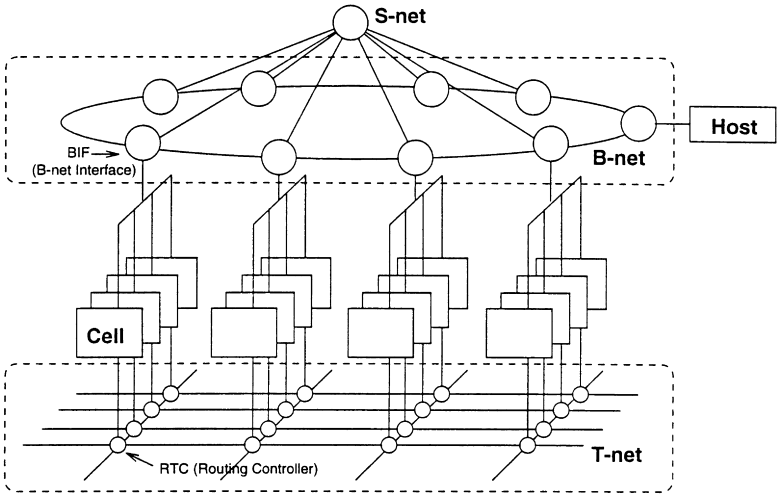

Fig. 1. Architecture of the AP1000 multiprocessor.

to viewing processes as actors which pass messages to each other.

Consistent with this coarse-grained model of parallelism, there is no global memory, and processes communicate with each other using messages passing through typed channels. A channel is associated with a particular process and enables other processes to send messages to it. A process is only able to receive messages from its own channel, but can send messages to any channel attached to any other process.

3. The Hough transform

The Hough transform is a powerful image processing technique, allowing the extraction of the parameters of lines, circles and other parameterizable curves. The principal advantages of the method are its ability to recognize partial or occluded objects, and its ability to cope well with noise and discontinuities in edges.

The major disadvantages of the Hough transform are its requirement for an enormous amount of computation and a large amount of storage capacity. A further restriction of the method is that it must be known in advance what pattern is being searched for. The latter restriction means the method has primarily been used to search for straight lines, circles and ellipses, although more general shapes can be detected by Ballard's generalized Hough transform.

We assume that the image has already been subject to an edge detection operator to produce a binary image consisting of only the edge (or feature) points of interest. The Hough transform produces a solution to the equation:

$$\rho = x\cos(\theta) + y\sin(\theta),\ 0 \leq \theta < \pi \quad (1)$$

where ρ is the length from the origin of the normal to the line and θ is the angle that the normal makes with the horizontal, so that the line extracted is described.

The Hough transform is a mapping from the image plane to a parameter space. Every feature point in the image votes for all the possible lines passing through that point. These votes are accumulated in an array in the parameter space. This array is known as the accumulator or Hough array. To determine the parameters of a possible line, the maximum vote in the accumulator array is determined.

Finding the parameters ρ and θ of a line therefore involves transforming each feature point (x,y) in the image to find a value for ρ using Eq. (1) in steps of π/θ_{res}, where θ_{res} is the desired resolution for the θ index of the accumulator array. For an image of size $N \times N$, the ρ index ranges from $-\sqrt{2}N$ to $\sqrt{2}N$. Transforming a single point (x,y) generates a series of points resembling a sinusoidal locus in the parameter space. Transforming two points reveals two sinusoidal loci. Each point of each sinusoid represents a particular line. The point at which the two curves intersect indicates that both of the two image points have the line represented by the intersection passing through them. Thus, by determining the coordinates of the intersection point, the parameters ρ and θ of the (imaginary) line joining the two image points are revealed.

3.1. Parallelizing methodology for the Hough transform

The basic Hough transform involves two distinct steps:

- Accumulation of the votes in the accumulator array. This involves transforming each feature point in the image and recording each set of possible parameters by placing a vote in the correct element of the accumulator.
- The detection of peaks in the accumulator array to determine the parameters of the instance of the object required.

This section describes parallel implementations of the Hough transform and speedup results on the AP1000. There are a large number of possible ways to parallelize the Hough transform on different multiprocessor systems. In this section, we describe four algorithms to parallelize the Hough transform, implement them in the AP1000, and evaluate their performance by measuring execution times on the machine.

The experiments were run on images of size 64×64, with a varying concentration of edge points. For example, a single line with parameters $\theta = 80^\circ$ and $\rho = 50$ and of one pixel width, contains 64 points, or a concentration of 1.6% of the total pixels. In this work, we are primarily interested in the different parallelizing methods and the efficiency of the data communication.

Experimental results were collected and stored in a file by the AP1000 CellOS operating system. This file was filtered through an appropriate conversion utility, which collated the data into an ordered and sorted manner showing the output of each processor numbered from 0 to 127. Timing information for the experiments were obtained from this output.

Mechanisms for timing in paraML are not very advanced. All timing measurements were made by getting the system time in a single user mode. The time returned was the elapsed time since the start of the program.

3.1.1. Algorithm 1: distributing the image

The first algorithm to parallelize the Hough transform involved distribution of parts of the image to different cells. The task scheduling was a master/slave type of computation and communication model adopted as a way of coordinating the tasks requiring work. Using such a model, a single master process coordinates the work, distributing and collecting data from a number of slave processes which are given the task of doing the bulk of the computation in parallel with each other. The accumulator arrays in the different slaves were passed on to the master for combining.

Sine and cosine look-up tables were used during the Hough computation to speed up the calculation. Experimental

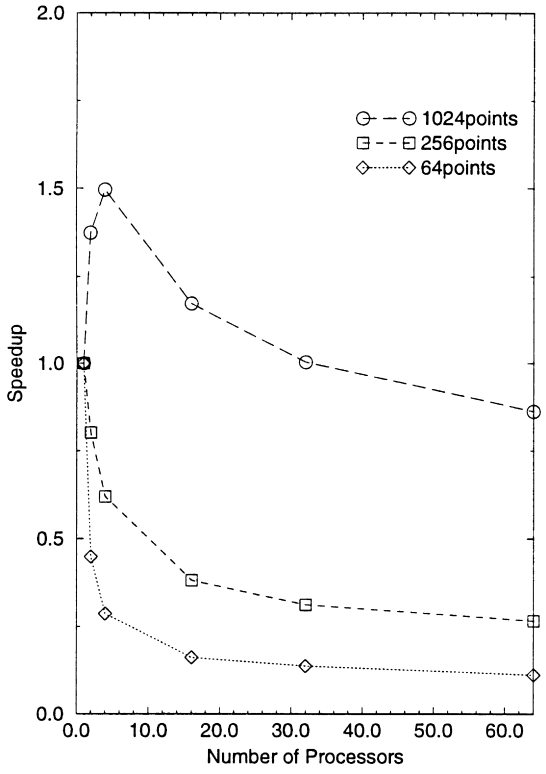

Fig. 2. Speedup for Algorithm 1.

tests showed that using look-up tables reduced the time for repeated calls to sine and cosine by approximately half.

The pseudo-code for this algorithm was:

1. Create the image as a two-dimensional array.
2. Store the edge points as a list of equally-sized sublists of image points.
3. Top-level function creates slaves (passing master's channel name as an argument).
4. Master sends each slave an equally-sized sublist of image points.
5. Slaves gets their own list of edge points and compute ρ for the full range of θ values and update their local accumulator.
6. Slaves combine their accumulators in a binary tree fashion and the final complete accumulator is sent to the master.
7. Master gets the final accumulator, finds the peak, and returns ρ and θ values.

Table 1 shows timings with varying numbers of edge points in the image and the number of slaves. Fig. 2 shows a speedup graph for this algorithm. All figures are averages from two runs of each combination.

Speedup results for this algorithm show that speedup is non-existent or very poor. In most cases, performance is worse than the single processor version. The greatest speedup occurs only when there is enough work to be done by the slave processors (1024 edge points), and when there is a relatively small number of processors (four processors gives a speedup of almost 1.5). The benefit of this method is a fast Hough calculation because of the relatively small number of edges in each slave, 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 idle. After the second combining stage three quarters of the cells are idle.

It was clear that in this simple approach, the amount of work done by each slave in the transform calculation was not balanced — the majority of rows in the image had no edge points at all, while a small number had up to six edge points. The algorithm was then adjusted to send an equal number of edge points to each slave.

3.1.2. Algorithm 2: distributing the accumulator array

The major problem with the first algorithm was the amount of time spent in the combining of the accumulator arrays from the slaves. By distributing different parts of the accumulator array among the slaves, rather than distributing the edge points, it was anticipated that better performance could be achieved.

The second partitioning scheme divides the accumulator array amongst the slaves. The entire set of edge points is sent to each slave while the accumulator is split among the cells in the θ dimension only, so that each slave 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 sub-portion of the accumulator can be sent back to the master process so that the peak can be found based on the entire accumulator array, 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.

Table 1
Algorithm 1 timing

Number of processes	64 points	256 points	1024 points
1	2.106 s	4.991 s	16.357 s
2	4.685 s	6.221 s	11.902 s
4	7.345 s	8.061 s	10.936 s
16	13.005 s	13.066 s	13.946 s
32	15.489 s	15.940 s	16.280 s
64	18.770 s	18.770 s	18.969 s

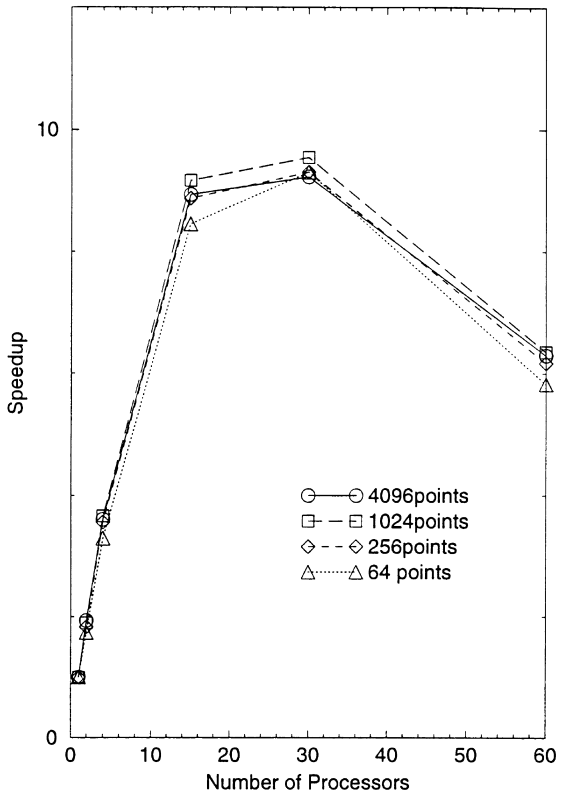

Fig. 3. Speedup for Algorithm 2.

The latter method has been used in the algorithm which follows. 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 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.

Much of the code used for the implementation of this algorithm is similar to that used for Algorithm 1. The main differences are:

- Since there is no binary tree combining required, slave to slave communication is not required. This results in a simpler model for the channels.
- The whole list of image edge points is sent to every cell, one after the other.
- Each cell contains its own local accumulator with a size dependent on the number of slaves used. For example, with n slaves, the local accumulator is $1/n$th the size of the complete accumulator.
- Each cell finds a peak in its own local accumulator.

Some fine adjustments to the algorithm were explored. Interestingly, it was found that performance improved if the image points were sent to the slaves as integers rather than reals. This required more work in the slaves (converting the integer values to real values) but less communication traffic as integers are smaller than reals.

Table 2 shows the execution times, while Fig. 3 shows the speedup. Time is measured starting from just after the creation of the image, until after the peak has been found. Therefore the time to create the slave processes is included.

Overall, the results were an improvement over the single processor case, but did not scale with more processors. Speedup increased to a maximum of approximately nine for 30 slaves, and then decreased as the number of slaves increases to 60. This is due to the amount of time that it takes to send the image serially to each slave one after the other. In effect, the serializing effect of sending the image leaves the last slave idle while waiting for its work to arrive. When its work arrives, earlier slaves may have already finished their work and are reporting results back to the master.

3.1.3. Algorithm 3: broadcasting the image

After we commenced our work, paraML introduced a multicasting primitive (using the B-net of the AP1000) to enable broadcasting of the same message to all the processes that are part of a group of processes. The object of this new primitive is to speed up the sending and receiving of identical messages which are required by a process group.

Clearly, in a method which involves sending the same set of image edge points to every slave process, broadcasting the edge points seems the obvious way to improve performance. Thus, with the introduction of this new primitive, the partitioning strategy employed in Algorithm 2 can be used more efficiently.

Table 3 shows the timing results, while Fig. 4 shows a speedup graph. Speedup for Algorithm 3 is far superior to that achieved in Algorithm 2. The bottleneck in the second algorithm (slaves waiting to be sent the image points) is avoided by broadcasting the image initially. All slaves can

Table 2
Algorithm 2 timing

Number of processes	64 points	256 points	1024 points	4096 points
1	1.945 s	4.702 s	15.808 s	59.285 s
2	1.118 s	2.569 s	8.319 s	30.596 s
4	0.590 s	1.319 s	4.312 s	16.439 s
15	0.230 s	0.529 s	1.724 s	6.599 s
30	0.208 s	0.504 s	1.654 s	6.422 s
60	0.333 s	0.761 s	2.488 s	9.410 s

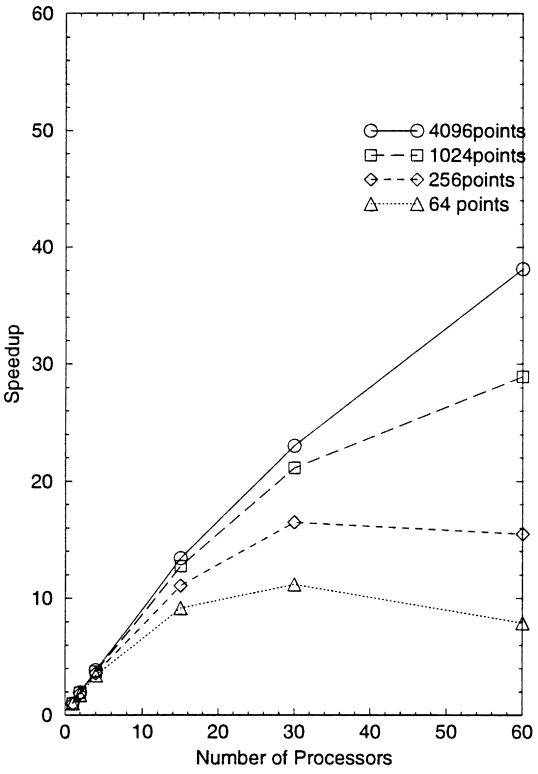

Fig. 4. Speedup for Algorithm 3.

then determine their peak and send the results back to the master.

3.1.4. Algorithm 4: iterative refinement

Although Algorithm 3 showed good speedup, it may be argued that the actual times for the uniprocessor program were unacceptably slow. Following the work of Illingworth and Kittler [18] and Wallace [19], a different kind of algorithm was developed to improve performance over those which have been described above. Such an algorithm relies on an iterative technique of refinement based on dynamic parameter range reduction. This approach reduces the amount of redundant computation which can often be present in the standard Hough transform. However, this may be done at the expense of some accuracy if careful attention is not paid to the algorithm.

The algorithms of the previous sections used a master/slave model of computation and communication. The new algorithm to be presented in this section uses a different model known as the worker farm (or processor farm) model. In this method there exists worker processes (equivalent to the slaves) and a farmer process (similar to the master). The farmer coordinates the work tasks and receives results back from the workers, but is different to a master since it dynamically assigns work to the next available worker from a pool of waiting tasks. When a worker has finished its current task it alerts the farmer that it is available and ready to receive the next task. There is no direct communication between workers.

The algorithm devised for this experiment is similar to that by Wallace et al. [19] and is based on an iterative technique whereby most of the processing is done on areas of the accumulator which are likely to possess the highest peaks. The aim of the technique is to lower the amount of work done by starting off the computation at a very coarse accumulator quantization and then gradually focusing in on a few selected parameter ranges at each iteration [11,18]. Very small accumulators are used but since the parameter range is reduced at each iteration, the quantization is quite fine by the final iteration. The accumulator size chosen for this experiment is 4×4 and the maximum number of iterations is also four.

The farmer begins the processing by creating the image and storing the edge points as a list of (x,y) coordinate pairs. In common with Algorithms 2 and 3, the entire set of image edge points is sent to every worker process. The basis of the algorithm is that the farmer sends an item of work to a worker. This work item is actually a tuple containing a ρ value, a θ value and a current iteration level number. The ρ and θ values define a subarea of the current accumulator and are the lower bounds of a range of parameter values. The iteration level number allows the upper bounds to be calculated by the worker when it receives the tuple of work. The worker is also able to calculate and store its own sine and cosine look-up tables (four values in each), before proceeding with the accumulation of Hough transform votes. After the voting, the accumulator is searched for candidate peaks above a certain threshold. The threshold is chosen depending on the total number of edge points. Varying this threshold will have a great impact on the actual amount of work done by the worker.

Table 3
Algorithm 3 timing

Number of processes	64 points	256 points	1024 points	4096 points
1	1.945 s	4.702 s	15.808 s	59.285 s
2	1.115 s	2.554 s	8.137 s	30.149 s
4	0.567 s	1.309 s	4.220 s	15.431 s
15	0.212 s	0.425 s	1.243 s	4.432 s
30	0.174 s	0.285 s	0.748 s	2.575 s
60	0.246 s	0.303 s	0.546 s	1.554 s

Table 4
Algorithm 4 timing (seconds) — 252 points

Threshold	Number of workers						
	1	2	4	8	16	32	64
5%	63.65	34.13	17.26	8.97	4.78	8.31	8.14
10%	31.23	16.82	8.53	4.55	2.68	2.97	6.14
15%	11.08	6.05	3.19	1.98	1.51	1.39	1.42

If there are no peaks in the current accumulator, the worker alerts the farmer that it is now free and ready to receive more work. If there are peaks present and the process has not yet reached the final iteration level, then one of the peaks is kept by the worker for further refinement. The others are sent back to the farmer to be distributed further. If the final iteration level has been reached then any final peaks (if they exist), are sent to the farmer.

In addition to varying the number of worker processes, the variable quantity, which had a great impact on execution time, was the threshold for peak determination. Initially a straight line of 64 points was used as the pattern to be detected and the parameters were successfully found. The threshold was varied from 10% of the total number of edge points to 90%, in 10% increments. It was found that the parameters could not be determined for thresholds above 60%. It was also found that there was no threshold that used all of the workers at all times. In fact, with a 20% threshold, at most 28 workers were used, while for a 60% threshold at most only five workers were used.

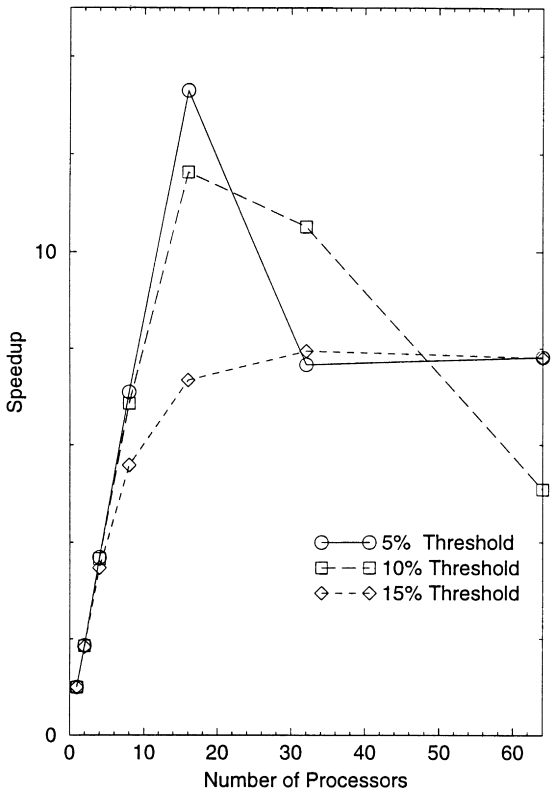

Fig. 5. Speedup for Algorithm 4.

Due to the unrealistic results shown for 64 points, an image with four lines was produced (252 edge points), in order to increase the amount of work. Table 4 shows the timing results. Fig. 5 shows the speedup graph. It was determined that since the threshold is a percentage of the number of edge points and there are now four lines to be detected, the threshold needs to be reduced by a factor of four to produce similar numbers of peaks as found for 64 points. Thus 15% is equivalent to 60% previously. Due to the increased workload, more of the workers were used. In fact, all workers were used when up to 32 workers were created. However, with 64 workers created, the number of used workers was 47 at most.

For 64 points, single processor timings are relatively fast. However, multiprocessor timings are not substantially better and when the threshold is above 50% they begin to become worse than the single processor. For 252 points, the single processor timings increase dramatically, yet the multiprocessor timings are favorable with good speedups for up to 16 workers. It is interesting to note that some of the timings actually start to increase again as the number of workers increases. Although the extra creation time of the workers makes some small contribution to this, it is not enough to produce the variations shown. A possible explanation is that as the amount of work increases, the number of messages increases dramatically. Since the work tasks are quite small, the bottleneck becomes the farmer, which may be kept so busy dealing with all the messages that the workers may become idle waiting for something to do.

The single processor version of the algorithm however, must be tuned to match the input data. If too low a threshold is selected, the algorithm does unnecessary work. If too high a threshold is selected, the peak is not found. In contrast, the paraML iterative version can have a conservative threshold selected and will then utilize its pool of workers if unnecessary computation is required.

4. Conclusions

The Hough transform has been implemented on the Fujitsu AP1000 distributed memory multiprocessor system. Four different parallel algorithms of the Hough transform were tested for speedup in the machine. Each algorithm illustrated various ways of partitioning the image and

accumulator array among the different processors. The speedup resulting from the algorithms has been presented. The multicasting feature of the Fujitsu AP100 has been investigated and found to be very useful in the parallel implementations of the Hough Transform.

Acknowledgements

The authors would like to thank the CAP project at the Australian National University for providing access to the AP1000 machine. We would also like to thank Peter Bailey for developing the paraML system. Peter has has been an invaluable source of information, ideas, and help. Both Peter and David Sitsky also provided timely and friendly system-administration help during the project.

References

[1] B.R. Sklar, A.K. Somani, Ray tracing: parallelization via image decomposition and performance impact, 1996 International Conference on Parallel Processing, Aug 12–16, 1996, pp. II–108–115.
[2] G. Knittel, A parallel algorithm for scientific visualization, 1996 International Conference on Parallel Processing, Aug 12–16, 1996, pp. I–116–123.
[3] V. Leavers, Shape Detection in Computer Vision using the Hough Transform, Springer, London, 1992.
[4] T.M. Silberg, The Hough transform on the geometric arithmetic parallel processor, IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management, 1985, pp. 387–393.
[5] C. Guerra, S. Hambrusch, Parallel algorithms for line detection on a mesh, Journal of Parallel and Distributed Computing 6 (1989) 1–19.
[6] C.S. Kannan, H.Y.H. Chuang, Fast Hough transform on a mesh connected processor array, Information Processing Letters 33 (1990) 243–248.
[7] D. Krishnaswamy, V. Govindan, C. Nagendra, The Hough transform — a fine grain algorithm for mesh connected processors, IEE Colloquium on Hough Transforms, London, May 1993, pp. 9/1–4.
[8] J. Princen, J. Illingworth, J. Kittler, A hierarchical approach to line extraction, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, June 4–8, 1989, pp. 92–97.
[9] J.M. Jolion, A. Rosenfeld, An $O(\log n)$ pyramid Hough transform, Pattern Recognition Letters 9 (1989) 343–349.
[10] C.A. Sher, A. Rosenfeld, A pyramid Hough transform on the connection machine, Technical Report CAR-421, Centre for Automation Research, University of Maryland, February 1989.
[11] M. Atiquzzaman, Multiresolution Hough transform — an efficient method of detecting pattern in images, IEEE Transactions on Pattern Analysis and Machine Intelligence 14 (11) (1992) 1090–1095.
[12] M. Atiquzzaman, Pipelined implementation of the multiresolution Hough transform in a pyramid multiprocessor, Pattern Recognition Letters 15 (9) (1994) 841–851.
[13] Y. Pan, Y.H. Chuang, Parallel Hough transform algorithms on SIMD hypercube arrays, International Conference on Parallel Processing, August 13–17, 1990, pp. III 83–86.
[14] S. Ranka, S. Sahni, Computing Hough transforms on hypercube multicomputers, The Journal of Supercomputing 4 (2) (1990) 169–190.
[15] A.N. Choudhary, R. Ponnusamy, Implementation and evaluation of Hough algorithms on a shared-memory multiprocessor, Journal of Parallel and Distributed Computing 12 (2) (1991) 178–188.
[16] H. Ishihata, T. Horie, S. Inano, T. Shimizu, S. Kato, An architecture of highly parallel computer AP1000, IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, May 1991, pp. 13–16.
[17] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, Cambridge, 1990.
[18] J. Illingworth, J. Kittler, Adaptive Hough transform, IEEE Transactions on Pattern Analysis and Machine Intelligence 9 (5) (1987) 690–698.
[19] A.M. Wallace, G.J. Michaelson, P. McAndrew, K.G. Waugh, W.J. Austin, Dynamic control and prototyping of parallel algorithms for intermediate- and high-level vision, Computer 25 (2) (1992) 43–53.

Austin Underhill. Biography not available at time of publication.

Mohammed Atiquzzaman received his M.Sc. and Ph.D. degrees in electrical engineering and electronics from the University of Manchester Institute of Science and Technololgy, England in 1984 and 1987, respectively. He had been an academic staff member at La Trobe University and Monash University, Melbourne and King Fahd University of Petroleum and Minerals, Saudi Arabia. Currently he is a faculty member in the Department of Electrical and Computer Engineering at the University of Dayton, Ohio. He is a senior editor of the IEEE Communications Magazine and has been a guest editor on special issues on 'ATM Switching' and 'ATM Networks' of the International Journal of Computer Systems Science and Engineering, a special issue on 'Parrallel Computing on Clusters of Workstations' in the Parrallel Computing Journal, and a special issue on 'Enterprise Networking' in the Computer Communications Journal.

He has also served as a technical program committee member of several international conferences, including the IEEE INFOCOM and the IEEE Annual Conference on Local Computer Networks.

His current research interests are in Broadband ISDN and ATM networks, multiprocessor systems, interconnection networks, parrallel processing and image processing. He has published widely in the above areas. He can be contacted at atiq@engr.udayton.edu.

John Ophel. Biography not available at time of publication.