Shared Memory Multiprocessor Implementation and Evaluation of Hough Transform Algorithm *

Alok N. Choudhary and Ravi Ponnusamy
Department of Electrical and Computer Engineering
Syracuse University, Syracuse, NY 13244.

Abstract
In this paper we present several techniques to implement hough transform computations on a shared memory multiprocessor and present their performance.

1 Introduction
Hough Transform is one of the most common methods to detect lines in binary or edge images [1]. A straight line in a two dimensional space can be represented by two parameters $\rho$ and $\theta$, where $\theta$ is the angle that the normal to the lines makes with the origin, and $\rho$ is the length of the normal. A serial algorithm to compute hough transform has complexity $O(N^2 \times \theta_{\max})$, where $\theta_{\max}$ is the number of quantizations in the $\theta$ space. In this paper we present several parallel implementations of hough transform on a shared memory multiprocessor; namely, the Encore Multimax. Section 2 contains various implementations strategies, and Section 3 presents the implementation results.

2 Parallel Implementations

2.1 Uniform Partitioning
In this scheme, the image is equally divided into as many blocks as there are number of processors. Each processor computes the hough vote count for its portion of the image for all values of $\theta$.

2.2 Static Edge Partitioning
In this partitioning scheme equal number of edges are partitioned among processors. Hence, for non-uniform edge image, the computational load can be evenly distributed among processors. However, in order to perform this partitioning, some preprocessing is required in which sum of edge count in the entire image is first computed. Then the image is partitioned along rows such that each partition contains almost equal number of edges.

2.3 Update Schemes
Fine-grain Update(SEP.FG): In this policy, each time a new $\rho$ value is computed by a processor, the global hough array ($H$) is locked, and the corresponding ($\theta, \rho$) cell in $H$ is updated.

Fine-Grain Update with Lock Arrays: In order to avoid congestion for a lock on the hough array for fine-grain update algorithm, an array of locks is provided. A reasonable compromise is to provide one lock for a set of rows of the hough array. Suppose the number of locks is $N_l$. Then there will be $\theta_{\max}$ rows of $H$ per lock.

Coarse-Grain Update (SEP.CG): To decrease the time spent in waiting for locks and access to the common bus for updating the global hough array, additional memory is used to compute vote count locally by increasing the update granule size. For example, a processor can compute the vote count for some number of edges (specified as input parameter) and all $\theta$ values, and then update the global array. Therefore, local granule size and update granule size can be varied.

Local Array Update (SEP.LA): In this strategy, each processor has its own local hough array. Therefore, each processor needs $2\sqrt{2} \times \theta_{\max} \times N$ memory. However, locking is completely avoided in this case. For its edge partition, each processor updates its local array. When all processors finish the vote computation phase, their partial hough arrays are combined in order to obtain the final hough array.

Parameter Partitioning (PP): Parameter partitioning assigns mutually exclusive sets of parameters.

---

*This work was supported in part by the Engineering Foundation Research Initiation Grant no. RI-B-90-23.
to different processors. In other words, the parameter space is partitioned into $\theta_{\text{max}}/P$ portions. Each processor computes vote count for all edges in the image, but only for its subset of parameters.

3 Performance Evaluation

Figure 1 shows the speedups for various schemes to compute the hough vote count for static edge partitioning (SEP). This figure contains results for an image containing 5% edges that are uniformly distributed. Fine grain update algorithm with single lock (SEP-FG) does not provide any speedup at all, and in fact, performance progressively decreases compared to sequential processing as the number of processors is increased. The reason for this behavior is that the time for each vote computation is much smaller than the time spent waiting for a lock on the hough array. Therefore, most processors spend time waiting to update the array rather than performing useful computation. The best performance is obtained for coarse-grain update (SEP-CG16). In Figure 1, graph (SEP-CG16) for which the coarse-grain size is 16 edges (hence update size of $16 \times \theta_{\text{max}}$) is given.

Parameter partitioning (PP) performs the second best. When SEP-FG is ignored, fine-grain update with lock arrays (SEP-LCK4, i.e., 1 lock per 4 rows of $H$) performs the worst.

Varying Computational Load: The relative behavior of the performance of the partitioning schemes changes very little as the fraction of edges is increased as illustrated in Table 1. The table shows the computation times for static edge partitioning scheme for 5%, 10% and 20% edges in the input image.

Effect of Locking Granularity: Figure 2 shows a performance comparison of fine-grain update schemes when locking granularity is varied. When locking granularity is low (SEP-LCK1 and SEP-LCK4, i.e., 1 lock per row and 1 lock per 4 rows of the hough array respectively), the speedup improves as the number of processors is increased. As the locking granularity is increased further, the speedup increases as the number of processors is increased up to a certain number and then it declines as illustrated in Figure 2. It is clear that larger the locking granularity, smaller is the number of processors for which there would be any speedup gains for fine-grain update. Also, as the number of processors increases, the locking granularity needs to be reduced in order to obtain performance improvements. But it should be noted that coarse-grain update schemes require a much larger memory (which also increases linearly with the number of processors) than that is required for fine-grain update schemes.

References