Adv. in Comput. Sci. & Eng. **1**(1) (2007), 47-59

COMPUTING A COMPLETE HISTOGRAM OF AN IMAGE IN LOG(n) STEPS AND MINIMUM EXPECTED MEMORY REQUIREMENTS USING HYPERCUBES

TAREK M. SOBH

School of Engineering
University of Bridgeport
Connecticut, U. S. A.

Abstract

This work first reviews an already-developed, existing deterministic parallel algorithm [2] to compute the complete histogram of an image in optimal number of steps $(\log n)$ on a hypercube architecture and utilizing memory space on the order of $O(x^{1/2} \log x)$, where x is the number of gray levels in the image, at each processing element. The paper then introduces our improvement to this algorithm's memory requirements by introducing the concept of randomization into the algorithm.

1. Introduction

The first algorithm [2] to be reviewed in this paper is concerned with the task of computing the complete histogram of n gray level values in $\log n$ steps. The algorithm is described for hypercubes and computes the complete histogram in $\log n$ time independent of the range of gray level values. The computation of the complete histogram of n such values takes

2000 Mathematics Subject Classification: 68, 68U10.

Keywords and phrases: image processing, computer vision, image histogram, hypercube architecture, randomized algorithms.

Received March 16, 2007

place in a series of $\log n$ steps; after which, the histogram for value i can be found in the lowest-addressed processor whose address ends in i. The algorithm makes use of the association of suffixes of data values of increasing width with suffixes of processor addresses. We shall begin by defining the histogram of an image and the uses of the histogram in different image processing applications, then we shall define the SIMD hypercube multiprocessor and describe its interconnections. The algorithm in [2] will be reviewed after that. Finally, we present an improvement to this algorithm's memory requirements via the usage of randomization.

2. The Gray Level Histogram

One of the simplest and most useful tools in digital image processing is the gray level histogram. The gray level histogram is a function showing, for each gray level, the number of pixels in the image that have that gray level. The abscissa is the gray level and the ordinate is the frequency of occurrence (number of pixels). While the histogram of any image contains considerable information, certain types of images are completely specified by their histograms. When an image is condensed into a histogram, all spatial information is discarded. There are many uses for the gray level histogram. One important use is in digitizing parameters, which is due to the fact that the histogram indicates whether or not an image is properly scaled within the available gray level. Another important use is in boundary threshold selection, as contour lines provide an effective way to establish the boundary of a simple object within an image. The contours may be, for example, the 'dip' between two peaks in the histogram in the case of a light area within a dark area or vice versa. The area and the integrated optical density of a simple object can be computed from its image histogram too.

3. The SIMD Hypercube Microprocessor

A hypercube of dimension k has 2^k nodes and $k2^k$ edges. A hypercube of dimension 2 is analogous to a square, the hypercubes of dimension $d \geq 3$ can be recursively defined as obtained from two hypercubes of dimension $(d-1)$ each, by connecting corresponding nodes of the two hypercubes. That is, two cells share a direct connection if and only if their corresponding hypercube vertices are connected by a hypercube edge. Furthermore, we can see that two cells will share a direct connection if and only if their addresses differ in exactly one bit position (i.e., one in each dimension). In the SIMD multiprocessor model (single-instruction, multiple data), which is our model, all processing elements execute a sequence of instructions, sent from one controller.

4. The Algorithm

This section explains and reviews the algorithm introduced in [2], so that the reader of this article can follow later the development of our own contribution, which is namely, randomizing the algorithm. The histogram which is to be computed will be represented as a set of ordered pairs, each pair will contain an index (represented in binary; for example, $i_{m-1}, i_{m-2}, \ldots, i_3, i_2, i_1, i_0$ will represent an index of length m bites, which implies that the maximum number of gray levels allowed is 2^m), and a count, which is the histogram value of the corresponding index (that is, the number of pixels which have the index value as its gray level).

4.1. Initial configuration

Initially each processing element (PE) in the hypercube includes one and only one pair, which is in fact one pixel value, which means that the count component of the pair have the value "1" throughout the hypercube. This may be considered as if each PE of the hypercube contains a histogram consisting of single pair, which is obtained by pairing the gray level value (index) in the cell with the count "1".

The goal may be considered then to 'combine' all those histograms in different cells to form the complete histogram of the image, which is to be distributed in some reasonable way throughout the hypercube in order to be retrieved easily.

4.2. The basic idea behind the algorithm in [2]

The basic idea behind the algorithm is very simple, it is in fact the idea of 'combining' all the values distributed throughout a hypercube into a single processing element in $\log n$ steps, where n is the number of cells within the hypercube multiprocessor. Combining the set of values in the hypercube is a very simple and standard procedure. It can be described using a simple algorithm consisting of a loop that is to be performed k times, where k is equal to the quantity $\log n$. During iteration number j the value that is stored in the processor

$$a_{k-1}, a_{k-2}, ..., a_{j+1}, 1, a_{j-1}, ..., a_2, a_1, a_0$$

is to be sent to the processor

$$a_{k-1}, a_{k-2}, ..., a_{j+1}, 0, a_{j-1}, ..., a_2, a_1, a_0$$

which is done in exactly one time step, as this is a hypercube edge. Then, combined with the value stored in the latter processor, it can be seen that after k steps the 'combination' of all the elements that were originally distributed throughout the hypercube will be found in a single processor namely the processor

$$0_{k-1}, 0_{k-2}, 0_{k-3}, ..., 0_2, 0_1, 0_0.$$

4.3. The problems arising when using the basic algorithm

In [2], it can be readily seen that the previous sort of algorithm can be readily applied to a large class of problems, namely the class of problems where the amount of storage that is required to store the combined value after the current iteration does not increase or increase significantly, but does so with a slow rate as more values are combined. If this was not true, the amount of storage required would be in fact exponential. One such problem may be the addition problem. For the problem of combining histograms, this basic algorithm will not be suitable, because the output of the operation of combining two histograms may be twice as large as either of the original histograms. The consequence of this will be the exponential growth in the storage required for the histogram in a given processing element.

4.4. The description of the algorithm

The main idea behind the algorithm that the authors developed in [2] is to try to alleviate the problem with the basic algorithm of the exponential growth in memory requirement at the processing elements by allowing the histogram information to remain distributed to a certain degree while still aggregating it in a useful way in a series of $\log n$ steps. This is to be performed using the following algorithmic steps, presented in [2]:

The algorithm is still a loop with k iterations with $k = \log n$. At each iteration j of the loop during the first m steps, the following is to be performed (n = number of PE's; i.e., number of pixels, $m = \log$ (number of gray levels))

All the pairs with index (in binary):

$$i_{m-1}, i_{m-2}, ..., i_{j+1}, 0, i_{j-1}, ..., i_1, i_0$$

which are in a processing element in the hypercube whose address is:

$$a_{k-1}, a_{k-2}, ..., a_{j+1}, 1, a_{j-1}, ..., a_1, a_0$$

are sent to the PE in the hypercube whose address is:

$$a_{k-1}, a_{k-2}, ..., a_{j+1}, 0, a_{j-1}, ..., a_1, a_0$$

At the same time, in a complementary fashion, all pairs with the index:

$$i_{m-1}, i_{m-2}, ..., i_{j+1}, 1, i_{j-1}, ..., i_1, i_0$$

which are in a processing element in the hypercube whose address is:

$$a_{k-1}, a_{k-2}, ..., a_{j+1}, 0, a_{j-1}, ..., a_1, a_0$$

are sent to the PE in the hypercube whose address is:

$$a_{k-1}, a_{k-2}, ..., a_{j+1}, 1, a_{j-1}, ..., a_1, a_0$$

At the very first iteration of the loop, every pair in the hypercube whose index ends with a zero, which is in a PE whose address is:

$$a_{k-1}, a_{k-2}, ..., a_1, 1$$

will be sent to the PE $a_{k-1}, a_{k-2}, ..., a_1, 0$ also any pair with an index ending in a one, which is in a PE whose address is:

$$a_{k-1}, a_{k-2}, ..., a_1, 0$$

will be sent to the PE $a_{k-1}, a_{k-2}, ..., a_1, 1$

During the second iteration, every pair in the hypercube whose index ends with a $0, i_0$, which is in a PE whose address is:

$$a_{k-1}, a_{k-2}, ..., a_2, 1, a_0$$

will be sent to the PE $a_{k-1}, a_{k-2}, ..., a_2, 0, a_0$, also any pair with an index ending in a $1, i_0$, which is in a PE whose address is:

$$a_{k-1}, a_{k-2}, ..., a_2, 0, a_0$$

will be sent to the PE $a_{k-1}, a_{k-2}, ..., a_2, 1, a_0$

It can be readily noticed that some use is being made of the association of suffixes of the indices and suffixes of the PE addresses. After the values are sent, whenever two pairs with the same index (i.e., two pixels with the same gray level) are collected in the same PE, they are to be combined to form one pair with the same index and with the count value equal to the sum of count values of each individual pair, thus forming the histogram. At first glance, it might seem that the problem of the exponential growth in memory requirements at each PE still exists, due to two facts:

1. The number of bits required for the count increases by one each time a combine is performed.
2. The possibility that many pairs with indices which are the same in the last several bits may be initially located in processing elements such that they all happen afterwards to gravitate to a single cell, which implies that the possibility for exponential growth in the number of pairs stored in a particular processing element still exists.

However, when one takes a closer look at the operation of this algorithm, the situation will turn to be much better than it first appeared to be. Regarding the first concern - the increase of the number of bits needed for the count by one after each iteration - it can be seen that there is no need to store after iteration j the last $j+1$ bits of any index, since these bits will at that time be given by the last $j+1$ bits of the processing element address which contains that index. It is true that the number of bits needed to store the count will increase by one after each iteration but at the same time the number of bits required to store the index will decrease by one. Thus, the total number of bits required to store a pair will remain a constant throughout the whole algorithm. The value of this constant is simply equal to the number of bits required to store the gray level plus one (i.e., $(m+1)$). The argument about the exponential growth in the number of pairs to be stored at each processing element will also be found to be not exactly the case, and this is discussed by the authors in details in [2].

4.5. The formal algorithm

The following is the final algorithm in [2], described in pseudo-code like language.

```
Define $n$ = number of nodes in hypercube (number of pixels).
        $k = \log_2 n$.
        $x$ = number of gray levels.
        $m = \log_2 x$.
 For $j = 0$ to $(m-1)$ do
 $(***)$ Send all the pairs with index
            $i_{m-1}, i_{m-2}, ..., i_{j+1}, 0, i_{j-1}, ..., i_1, i_0$
     which are in a processing element in the hypercube whose address is
            $a_{k-1}, a_{k-2}, ..., a_{j+1}, 1, a_{j-1}, ..., a_1, a_0$
     to the PE in the hypercube whose address is
            $a_{k-1}, a_{k-2}, ..., a_{j+1}, 0, a_{j-1}, ..., a_1, a_0$
 In parallel, send all pairs with the index
            $i_{m-1}, i_{m-2}, ..., i_{j+1}, 1, i_{j-1}, ..., i_1, i_0$
     which are in a processing element in the hypercube whose address is
            $a_{k-1}, a_{k-2}, ..., a_{j+1}, 0, a_{j-1}, ..., a_1, a_0$
     to the PE in the hypercube whose address is
            $a_{k-1}, a_{k-2}, ..., a_{j+1}, 1, a_{j-1}, ..., a_1, a_0$
```

```
If two pairs have the same index are collected in the same PE, then they are to be combined to form one pair with the same index and with the count value equal to the sum of count values of each individual pair;
end For;
If $k = m$ then STOP else
(###) For $j = m$ to $(k - 1)$ Do
    Send the count value stored in the PE whose address is
        $a_{k-1}, a_{k-2}, ..., a_{j+1}, 1, a_{j-1}, ..., a_1, a_0$
    to the PE whose address is
        $a_{k-1}, a_{k-2}, ..., a_{j+1}, 0, a_{j-1}, ..., a_1, a_0$
    sum both count values and leave the combined histogram in the latter PE.
  end For;
end If.
```

The algorithm assumes that the number of bits to represent the gray levels is never more than the number of bits required to represent the number of pixels in an image, which is a logical and true assumption for nearly all realistic situations in computer vision applications.

4.6. Complexity analysis of the deterministic algorithm

It can be seen that with regards to the time complexity of the algorithm, that the algorithm runs exactly in $\log n$ loop iterations, the time for each loop iteration may be considered as one time step (including routing and the combining operations performed within a processing element), thus it is a $\log n$ time complexity process. With regards to space complexity, the maximum space required at a single PE will be an $O(2^{c/2})$ space per cell, where c is the number of bits required to store one of the histogram domain values.

To be more specific, the algorithm requires

$$(\log_2 x + 1)2^{\frac{\log_2 x}{2}}$$

space per PE, where x is the discretized number of gray levels.

5. Comments about the Deterministic Algorithm

The presented algorithm in [2] describes an intelligent approach to solving the problem of finding the histogram of an image using a popular parallel architecture in a number of steps equal to the diameter of the hypercube $(\log n)$, which is the best complexity that can be hoped for. The algorithm makes use of the association of data values (gray level values) and PE addresses to keep a growing collection of values distributed so that their growth is manageable. The space complexity as described above is $O(2^{c/2})$. It may be possible to reduce the space by the use of some encoding techniques during the storage of histograms that result while iterating. However, the algorithm would then have to be modified accordingly and its behavior may become more complicated. The algorithm works nicely for the cases when $m \leq k$, as the formal description of the algorithm imply.

6. The Randomized Version of the Algorithm

A randomized algorithm is best described as an algorithm where some of the decisions are made based upon the outcome of coin flips. The idea is to prove that the algorithm will behave in a certain manner with high probability. In our problem, we are concerned with proving that randomizing the algorithm will make the memory requirements at each processing element less than those for the original algorithm with a high probability. Typically, we mean a probability $\geq 1 - n^{-a}$ for any $a > 1$. Generally speaking, we can divide randomized algorithms into two classes. The first class is one in which the output is correct with high probability and is guaranteed to use a certain amount of resources. In the second one the algorithm is guaranteed to produce a correct output, using a certain amount of resource with a high probability. The first class is called a *Monte Carlo* algorithm the second is called a *Las Vegas* algorithm, our new algorithm is of the *Las Vegas* type.

7. The Randomization Scheme

The randomization scheme we describe is basically a method to change the step marked $(***)$ in the old algorithm to be a randomized routing step. This is instead of sending all the pairs to the specific processor

$$a_{k-1},\ a_{k-2},\ \ldots,\ a_{j+1},\ 0,\ a_{j-1},\ \ldots,\ a_1,\ a_0$$

we send them randomly in a first phase to the $k-j$ processors (c means complement)

$$a_{k-1},\ a_{k-2},\ \ldots,\ a_{j+1},\ 1,\ a_{j-1},\ \ldots,\ a_1,\ a_0 \tag{1}$$

$$a_{k-1},\ a_{k-2},\ \ldots,\ a_{j+1}c,\ 1,\ a_{j-1},\ \ldots,\ a_1,\ a_0 \tag{2}$$

$|$

$|$

$$(k-j-1)a_{k-1},\ a_{k-2}c,\ \ldots,\ a_{j+1},\ 1,\ a_{j-1},\ \ldots,\ a_1,\ a_0$$

$$(k-j)a_{k-1}c,\ a_{k-2},\ \ldots,\ a_{j+1},\ 1,\ a_{j-1},\ \ldots,\ a_1,\ a_0$$

and then, a second phase, sends all 'misplaced' pairs; i.e., pairs with a zero in their jth position in PE's $2 \Rightarrow (k-j)$ to the proper PE's

$$a_{k-1},\ a_{k-2},\ \ldots,\ a_{j+1}c,\ 0,\ a_{j-1},\ \ldots,\ a_1,\ a_0 \tag{2'}$$

$|$

$|$

$$((k-j-1)')a_{k-1},\ a_{k-2}c,\ \ldots,\ a_{j+1},\ 0,\ a_{j-1},\ \ldots,\ a_1,\ a_0$$

$$((k-j)')a_{k-1}c,\ a_{k-2},\ \ldots,\ a_{j+1},\ 0,\ a_{j-1},\ \ldots,\ a_1,\ a_0.$$

Note that this scheme complies with the idea behind the original algorithm regarding that the suffixes of the indices of the pairs are the same as the suffixes of the processing elements addresses, with the advantage that the pairs are dispersed within the hypercube more than before.

8. Probabilistic Analysis of Memory Requirements

An analysis of the memory locations required at each PE is to follow. The analysis is done for a specific processing element $[y]$ at the first step of the randomization procedure, the same analysis could be applied to all PE's within the hypercube without loss of generality, the same method could then be used for the following iterations till $j = m - 1$. The fact that the distributions used to calculate the number of pairs that are expected to be at a certain processor (in the probability tree) are binomial distributions and that the sum of $k - j$ such distributions is still a polynomial random variable could be applied to use Chernoff's lemma. Assume X is the random variable of the number of pairs (memory locations needed), and m is the expected value we calculated, then

$$E(X) = X$$

Implies (since X is polynomial) that the Probability $(X \geq (1 + epsilon)m)$

$$\text{is } \leq e^{\left(\frac{m}{2} epsilon^2\right)} << n^{-a}, \text{ where } a > 1,$$

which satisfies that the memory requirements are less than m with a probability $\geq 1 - n^{-a}$, for any $a > 1$, as described before in the randomization criteria.

9. Complexity Analysis of the Randomized Algorithm

It could be clearly seen that the memory requirements have decreased significantly when applying this randomization scheme. The number of bits for every pair is still equal to $\log_2 x + 1$ at every time during the iterations. Due to the 'balance' we discussed in the probabilistic analysis, the number of pairs will always be close to a constant. Even if the number of pairs increased to be proportional to $(\log_2 x)$, this will definitely be better than $\sqrt{x}$. One may argue that sending the pairs to more than one processor for randomization purposes will increase the time complexity to be more than $O(\log n)$ steps, but this is not true because the number of pairs will always be close to a constant as we proved. Thus, the total memory requirements will be $O(C \log x + (\log_2 n - \log_2 x))$ bits.

The second term is for the count increase by one for the loop $(\#\#\#)$ after the index suffix is scanned for all the PE's. The time complexity remains $O(\log n)$. As an example, $256 * 256 * 8$ images that we used in the original algorithm the memory requirements were 144 bits for each PE. For the randomized algorithm, the memory requirements will be equal to

$$C(\log_2 256 + 1) + (\log_2 65536 - \log_2 256)$$

i.e., $C * 9 + 8$, which, if C was even equal to $\log_2 x$, will be equal to 80 bits only for each PE with very high probability and the effect will be more when the number of gray levels in an image is more.

10. Summary and Comments

The presented randomized algorithm describes a method of sound improvement with regards to memory requirements when compared to the deterministic algorithm. The new memory requirements could be used at each PE and one can be sure that they will suffice with very high probability. Randomized algorithms provide lower bounds for resources like time and memory than those provided by deterministic algorithms in many cases, especially in routing and routing-related problems. The time complexity remains optimal and of the order of the diameter of the hypercube.

References

[1] D. H. Ballard and C. M. Brown, Computer Vision, Prentice-Hall, 1982.

[2] T. Bestul and S. L. Davis, On computing complete histograms of images in Log(n) steps using hypercubes, IEEE Trans. Pattern Analysis Machine Intell. 11(2) (1989), 212-213

[3] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw Hill, 1984.

[4] M. O. Rabin, Probabilistic algorithms, Algorithms and Complexity, J. F. Traub, ed., Academic Press, 1976, pp. 21-40.

[5] S. Rajasekaran and T. Tsantilas, Optimal routing algorithms for mesh-connected processor arrays, Algorithmica 8(1) (1992), 21-38.

[6] L. G. Valiant, A scheme for fast parallel communication, SIAM J. Comput. 11(2) (1982), 350-361.

■