Parallel Traversal of Signature Trees for Fast CBIR

Aaron Davidson
Dept. of Computing Science,
University of Alberta, Canada
davidson@cs.ualberta.ca

John Anvik
Dept. of Computing Science,
University of Alberta, Canada
janvik@cs.ualberta.ca

Mario A. Nascimento
Dept. of Computing Science,
University of Alberta, Canada
mn@cs.ualberta.ca

ABSTRACT

This paper presents a technique for searching a memory-resident signature tree (S-tree) in a parallel fashion. Our implementation uses a very compact binary signature for the images' global colour histogram thus allowing the use of main-memory to store and search a reasonably sized image collection. Our results show that our parallel search of an S-tree is able to achieve good speedups; for instance, a 50,000 signature S-tree, with a branching factor of 8, searched using 4 processors is 3 times faster than using a single processor.

1. INTRODUCTION

One of the primary distinctions between previous document collections and the World Wide Web is the proportion of multimedia documents, images in particular. This has lead to a resurgence of research for efficient and effective retrieval of images from large image collections.

The primary technique used for querying image databases is Query By Example (QBE) in which a sketch or sample image of what is sought is provided to a Content-Based Image Retrieval (CBIR) system. (e.g., IBM's QBIC and Virage's VIR). The CBIR system then processes the query and returns a set of images deemed similar, based on extracted features from the query and database images. In order to assess similarity, images are compared based on extracted visual features, the most common abstraction being the Global Colour Histogram (GCH). An alternative to GCH presented in [2] is more space efficient for indexing images. Whereas a GCH is a feature vector containing the percentage of colours in the image, this alternative technique uses binary signatures to represent ranges of colour percentages in the image. Not only are binary signatures more space efficient than n-dimensional vectors, but they also have higher retrieval effectiveness [2]. We therefore set to use such signatures as our image metadata.

However, despite how effective an image abstraction is, a linear scan comparing all image metadata to the query's metadata is not practical. Nascimento et al. [7] improved on the Signature tree (S-tree) [4, 10] in order to use it more efficiently for indexing and searching colour images. Section 2 describes the improved signature tree and its processing.

As tree structures are amenable to parallelization [12] we have furthered the work done in [7] by devising an algorithm that searches in a parallel fashion an in-memory S-tree for image signatures. This technique improves the query response time of an image retrieval system using colour signatures by quickly pruning branches during the S-tree search. Section 3 describes our technique for searching the S-tree in parallel and presents experimental results for searching the memory-resident signature tree. Finally, Section 4 concludes the paper and offers directions for future work.

2. IMAGE SIGNATURES AND S-TREES

A GCH is, for an n-colour model, an n-dimensional feature vector $(h_1, h_2, , ..., h_n)$ where h_i represents the (typically) normalized proportion of each colour c_i. Images are therefore compared by a similarity metric that is usually based on the Euclidean distance between the two feature vectors representing the images. While GCH provides a good abstraction of images, storing an n-dimensional vector for each image may require significant storage space. There are plenty of other image abstraction techniques in the literature; due to the lack of space we refer the reader to [3, Ch. 2] for a thorough survey. We next discuss how the storage overhead of GCHs has been addressed by using binary signatures as a means to represent an image.

Traditionally, binary signatures are used to index text documents with the signatures representing individual words or collections of words in the document [5]. Chitkara et al. propose the use of similar signatures for indexing colour images [2]. Each of these image signatures is a composition of binary sub-signatures representing the image's colours. As is typically done in the CBIR domain, the colour space is quantized in a reasonably small number of colours (e.g., 64) in order to eliminate the effects of small colour variations in the image and to avoid large files for high resolution representations. An image signature is composed by n sub-signatures, one for each quantized colour. Each sub-signature is t bits long, each bit representing a range (in percentage) of colours in the image. These ranges may be of variable or equal size and the bit set in the colour image signifies that the dominance of colour c lies in a specific range. Thorough experimental results [2] have shown the so-called Variable Bin Allocation (VBA) scheme to be superior when compared to GCH, hence we use it in our work. Figure 1 shows some images and their respective (binary)

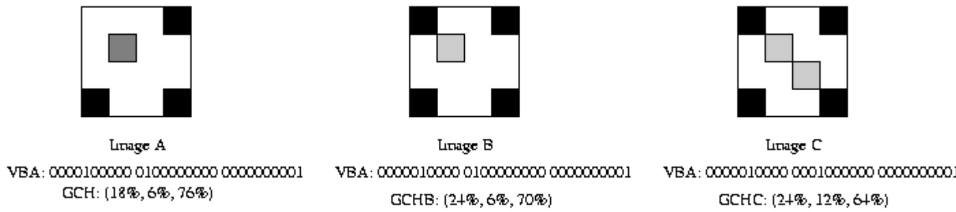


Figure 1: Sample images and their abstractions.

VBA signatures and GCH vectors.

The intuition offered in [2] to support the selection of the above ranges in the VBA signature is that images often contain a few colours and that most of these colours occur in small proportions. That is, having the signature focus on smaller portions of colours allows the system to be more discriminate when comparing images. It is important to note this scheme does not ignore the effects of colours with large proportions. Next we briefly discuss VBA's performance in terms of storage overhead.

As a single bit represents each range of colour proportion, image signatures provide compact and effective representation of the colour content of an image. For example, if a real number is represented by r bytes, then the space required to store a GCH for an image with n colours would be $(r \times n)$ bytes. Assuming $r = 2$, $n = 64$, and $t = 10$, then storing an image signature takes 80 bytes compared to 128 bytes for the corresponding GCH. However, as at most a single bit is set in each constituent sub-signature, we need only to record the location of the bit set instead of the whole colour signature. For a colour signature with 10 bins, the signature could be reduced from 10 to 4 bits for a space requirement of only 32 bytes; a savings of 75% over GCHs, hence using 4 Mbytes of main memory one would be able to store and search through a mid-size image database of roughly 100,000 images. Therefore we argue that it is indeed feasible to store and search reasonably sized image databases using only main-memory. Finally, the similarity between image signatures is also based on a Euclidean distance and is defined as:

$$d(Q, I) = \sum_{j=1}^{n} \left[pos(B_Q^j) - pos(B_I^j)\right]^2$$

where Q is the query image signature, I is another image signature, and $pos(B_R^m$ gives the position of the bit set in the bin for colour m of image R.

In order to index binary signatures, S-trees were first proposed in [4] and further improved in [10]. They are akin to R-trees [6] in the sense that they are dynamic, height-balanced and are capable of processing nearest-neighbour search efficiently (compared to a linear scan of the dataset). Each entry in an S-tree leaf node contains an object (e.g., image) signature and a respective pointer to it; hence, each leaf node contains a set of signatures. Those signatures are then OR'ed in order to form a super-signature which will represent all the individual signatures in the leaves and will be used in the internal nodes of the S-tree. Thus the further up in the tree, the more ones (1s) are set per node entry. In the case of a text signature this may become a problem as the upper nodes will have a low discriminatory power. Fortunately, in the case of the VBA signatures this is not as

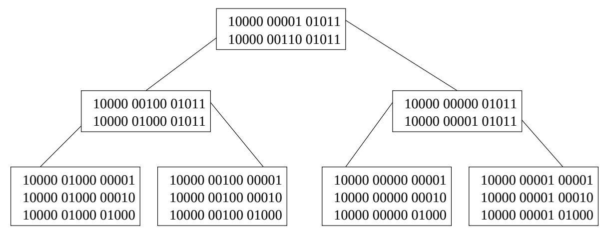


Figure 2: Example of an S-tree.

critical as the number of bits set per signature is fairly low. One should note that the VBA signatures used within the S-tree are not the compressed (location-encoded) signature mentioned above, but rather the explicit (using $n \times t$ bits) signature. Figure 2 shows an example of an S-tree.

In the recent work of Nascimento et al. [7] image signatures are grouped together in the leaves of the S-tree according to the similarity distance defined above. The authors enhance the traditional S-tree in two ways. The first is a new similarity function for nearest-neighbour searching of the tree called *minDist*. This function is similar to the distance function presented previously for colour signatures. However, *minDist* finds the distance of the closest bit in the (OR'ed) node signature to the bit set in the query, since the signatures in the nodes contain more than a single set bit. The *minDist* function is defined as:

$$minDist(Q, S) = \sum_{j=1}^{n} \left[pos(B_Q^j) - optPos(B_S^j)\right]^2$$

where the new function gives the location of the bit closest to position. The proper use of *minDist* allows the tree to be searched in an optimized depth-first order. The second enhancement is the use of a quadratic split algorithm (modified from [10]) during the splitting of nodes. For the leaf nodes the distance function is used, but for interior nodes the *minDist* function is applied. While this algorithm is more complex than the original linear signature assignment algorithm in [10], it produces lower weight nodes and therefore allows more selective tree traversals.

The main purpose of using an S-tree is that relevant matches to a query can be found faster than by doing a simple linear scan. The typical operation done with CBIR is to return the k most similar images to the query. For that purpose we use an algorithm very similar to the classical nearest-neighbour algorithm in [9]. Since *minDist* is overly-optimistic it is admissible[1] [8], pruning is safe (i.e. a pruned

[1] Indeed, this algorithm can be thought to be a version of the A^* algorithm [8].

node will provably not contain any of the k-best matches) and the nearest-neighbour algorithm will yield correct results.

A priority queue is used to store a list of unexpanded nodes in the tree, initially containing the root node. Nodes in this queue are ranked by their *minDist* to the query signature. A second queue, empty at first, will contain the k most similar leaf signatures found so far, ranked by their distance from the query signature. While there are nodes left in the node queue, the search algorithm removes the most promising node from the queue and expands it. Internal nodes have their children inserted into the node queue, and leaf nodes have their child-signatures inserted into the result queue. Once the result queue contains a minimum of k results, the k^{th} element's distance becomes the current pruning threshold. As argued above, if a node has a *minDist* greater than or equal to the threshold, it can be safely pruned.

3. IMPROVING QUERY PROCESSING WITH PARALLEL COMPUTATION

In parallel computation, we aim at using multiple processors to overlap the computation of portions of a program that have low data dependencies. The less dependence between portions of a program, the easier it is to divide into multiple, parallel tasks. Ideally, a parallel algorithm scales linearly with the number of processors. However, since parallel algorithms generally require extra communication between the processors that does not occur in a sequential program, the additional overhead prevents most parallel algorithms from scaling linearly. Thus, the goal in designing a parallel algorithm is to find the portions of the sequential algorithm that are not immediately dependent on each other. By overlapping these independent tasks and attempting to minimize the communication between processors, while still balancing the load between processors, good speedups can be obtained.

The first approach at improving the tree searching time used the *minDist* and d functions as the unit of work to parallelize the computation. The system creates a pool of threads, each of which runs on a different processor, and whenever a node is expanded, the children are placed on a work queue. The threads each take work off the queue, and calculate the *minDist* or d function for that item, and insert the result into the appropriate priority queues (either the node queue, or result queue). However, our experiments showed that the overhead of communication and synchronization between threads to be roughly as much as the cost of computing the distances sequentially, so this approach was quickly abandoned.

In the improved version, entire nodes are processed in parallel instead of doing individual distance calculations. With nodes being expanded in parallel, the order that the tree is traversed may be quite different from that of the sequential version. Each thread pulls the most promising node from the node queue, ranks all of the children, places them into the appropriate queues, and updates the global threshold value. Since several nodes are accessed in parallel, the threshold is often narrowed sooner, resulting in a slight improvement in performance.

To our knowledge, only two works are closely related to ours. Bosque et al. [1] performed a study in which they compared the results of running a parallel CBIR system for twenty-nine million images on both a cluster of 25 workstations and on an SGI Origin 2000 with eight processors. Our contribution differs from theirs in two aspects. First, we are using an S-tree and a threshold value to avoid searching the entire tree, while they linearly search all the signatures on all of the nodes. Second, we restricted ourselves to shared memory since, as argued earlier, the VBA signatures are compact enough to allow such an approach. While Bosque et al. did run the parallel CBIR system on an SGI Origin 2000, a shared memory machine, they focused more on their distributed memory strategy. Tousidou et al. [11] presented techniques for leaf distributions using a multiple-disk single-processor system, while we focus on shared-memory, multiple-processor systems.

3.1 Experiments and Results

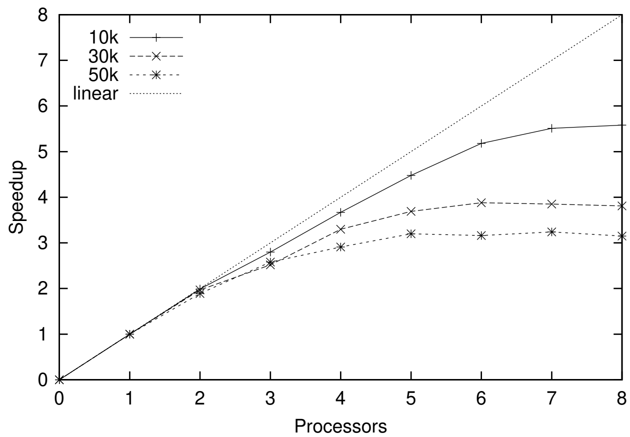
Our implementation was done in Java, as the S-tree data structure decomposes nicely into the object-oriented paradigm, and Java's native threading abilities facilitated considerably the implementation of the parallel algorithm. We carefully avoided unnecessary synchronization in Java which can be quite costly. Experiments were conducted on an SGI Challenge with eight 150Mhz MIPS R4400 processors. The S-trees were built with a branching factor of eight. Leaf nodes contained sixteen signatures - experiments showed that while larger leaves (with more than sixteen signatures) improve parallel performance, they also degrade sequential performance, biasing results unfairly towards our algorithm. For testing, we used three of the five signature collections used in [7]; each having 10,000, 30,000, and 50,000 signatures. For the query size, we used $k = 10$.

Table 1 shows how much time (average and percentage-wise) was saved, by using the parallel S-tree traversal, for a query on each of the datasets using different numbers of processors. As one can see, savings range from 47% to 82% when compared to the sequential version of the search algorithm. In order to allow an absolute comparison, the sequential S-tree required 3.99, 6.69 and 9.53 secs to process the dataset with 10,000, 30,000 and 50,000 signatures respectively. One should note though that the actual processing times are not as relevant as the relative savings given that processing times are not dependent on the proposed algorithms, but instead are dependent on the computer's CPU and main-memory speed (which may vary highly).

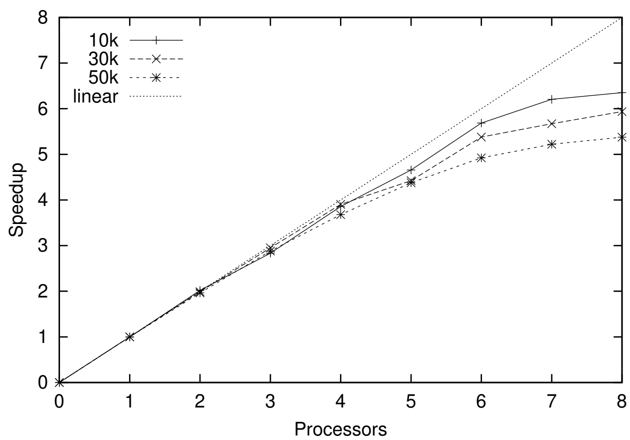
Dataset Size	2 proc.	4 proc.	8 proc.
10k	50%	73%	82%
30k	49%	70%	74%
50k	47%	66%	68%

Table 1: Savings in parallel query time ($k = 10$), relative to the sequential processing.

Speedup is a measure of how well a parallel algorithm scales and is calculated by dividing the sequential time by the parallel time. Figure 3 shows how our algorithm scales with the number of processors. The line labeled 'linear' shows the theoretical limit of speedups. The speedups manage to scale fairly well until about six processors, where the contention for accessing the priority queue begins to dominate the computation. Note, also that with larger S-trees, the speedups diminish at a faster rate. This is primarily because the granularity of processing internal nodes is much



(a) Using 16 signature leaves



(b) Using 24 signature leaves

Figure 3: Speedup of algorithm using up to 8 processors

smaller than at leaf nodes. Since a large S-tree has more internal leaves, processing them adds more relative overhead than for a smaller tree. However, by increasing the branching factor and the number of signatures per node, speedups can be improved. For instance, with 50,000 signatures, a branching factor of eight, and sixteen signatures at each leaf node, the speedup for eight processors is 3.2 (Figure 3(a)). If we use a branching factor of eight, and twenty-four signatures per leaf node, the speedup increases to 5.4, as shown by Figure 3(b).

4. CONCLUSIONS AND FUTURE WORK

We have shown that our technique for searching an VBA-based S-tree in parallel is significantly better than a sequential search of the same tree. In addition to traversing different branches of the S-tree in parallel, a cutoff threshold is found sooner and less nodes are expanded.

Our experimental results have been dealing with the situation where the entire S-tree is stored in memory. With current memory capacities, one could likely hold sets of image signatures of over a million entries before becoming short on memory. However, with extremely large data sets, there may be no other option than to store the S-tree on disk. A nice feature of our parallel algorithm is that it can easily overlap disk accesses. As leaves are accessed in parallel, all disk access times overlap. We are currently investigating this issue. One weakness in our parallel approach seems to be the priority queue. While it is essential in coordinating the search, it is also a bottleneck. We plan to investigate ways of alleviating this in our future work.

Future extensions of this work include investigating how this algorithm performs in a distributed memory environment, such as a cluster of workstations, where the communication costs are higher than in a shared memory system.

Acknowledgments

A. Davidson was supported by a grant from iCORE and a scholarship from the Province of Alberta, Canada. M. A. Nascimento was partially supported by a Research Grant from NSERC, Canada. We would also like to thank Profs. P. Lu and J. Schaeffer for discussions and help regarding parallel implementation issues. Java is a trademark of SUN Microsystems Inc.

5. REFERENCES

[1] J. L. Bosque et al. Study of a parallel cbir implementation using mpi. In *Proc. of the 5th IEEE Intl. Workshop of Computer Architectures for Machine Perception*, pages 195–204, 2000.

[2] V. Chitkara. Color-based image retrieval using compact binary signatures. Master's thesis, Dept. of Computing Science, University of Alberta, 2001.

[3] A. del Bimbo. *Visual Information Retrieval*. Morgan Kaufmann, 1999.

[4] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *Proceedings of the 9th ACM SIGIR Intl. Conf*, pages 77–87, 1986.

[5] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*, chapter 4. Prentice Hall, 1992.

[6] V. Gaede and O. Guenther. Multidimensional access methods. *ACM Comp. Surveys*, 30(2):170–231, 1998.

[7] M. Nascimento et al. Color based image retrieval using signature trees. Technical Report 02-01, Department of Computer Science, University of Alberta, 2000.

[8] J. Pearl. *Heuristics*. Addison-Wesley, 1984.

[9] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of the ACM SIGMOD'95 Intl. Conf.*, pages 71–79, 1995.

[10] E. Tousidou, A. Nanopoulos, and Y. Manolopoulos. Improved methods for signature-tree construction. *The Computer J.*, 43(4):301–314, 2000.

[11] E. Tousidou, M. Vassilakopoulos, and Y. Manolopoulos. Performance evaluation of parallel S-trees. *J. of Database Management*, July-Sept. 2000.

[12] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, 1999.