PARALLEL HOUGH TRANSFORM ON DCOM ARCHITECTURE

Savas KOSE, Cenker SISMAN
Nadia ERDOGAN
Istanbul Technical University
Electrical-Electronics Engineering Faculty
Computer Eng. Department
Ayazaga, 80626, Istanbul, TURKEY
e-mail:erdogan@cs.itu.edu.tr

ABSTRACT

A model of parallel computation is proposed for a network of processors which is enhanced with the infrastructure provided by Distributed Component Object Model (DCOM) architecture. A parallelization technique that utilizes the proposed model is presented for Hough transform, with its design and implementation details. The parallel algorithm is analytically and experimentally evaluated.

Keywords : parallel computing, image processing, Hough transform.

1. INTRODUCTION

As computer networks and sequential computers advance, distributed computing systems, such as networks of heterogenous workstations or personal computers become an attractive alternative to expensive, massively parallel machines. Algorithms related with image processing and pattern recognition are usually complex and time consuming. High-speed computation is needed especially in pattern recognition applications. Speed can be attained at low cost if the application can be divided into a set of tasks and if these can be optimally assigned to processors on a network. In this paper, we propose a model of parallel computation for a network of processors which are enhanced with the infrastructure provided by Distributed Component Object Model architecture and we present the implementation details of a parallelization technique for Hough transform to detect circles in a sample image. Our approach focuses on data partitioning.

The main idea of Hough transform is to map edge elements in the image space into a parameter space in such a way that peaks in the parameter space indicate possible instances of the sought pattern in the image [1]. The general algorithm can be applied to circles parameterized by the equation

$$(x-a)^2+(y-b)^2=r^2$$

Reductions in the amount of computation can be achieved if the gradient direction is integrated into the algorithm (Fig 1.). Differentiating the circle equation, recognizing that dy/dx=tan Ö, and solving for a and b, we get the following equations that give the coordinates of the center o a circle, where the value of the angle Ö can be calculated from the neighbour pixels for every edge element (x,y).

$$a = x + r \cos Ö$$
$$b = y + r \sin Ö$$

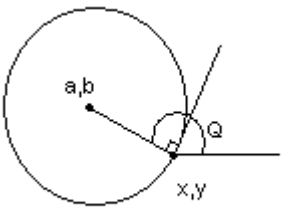


Fig. 1. Reduction in computation with gradient information

Sequential algorithm for Hough transform to detect circles:

Form an accumulator array of possible reference points A(a,b) for a certain value of r and initialize it to zero.

For each edge point (x,y) and for each value of r, repeat the following:

Compute Ö

Calculate the possible center (a,b)

Increment the accumulator array: A(a,b)=A(a,b)+1

Local maxima in the accumulator array correspond to possible location of circles in the image.

2. BASIC STRUCTURE OF COMPONENT OBJECT MODEL

Due to advances in high-speed networking, building software as distributed object applications has become preferable, but still a considerable amount of time is spent to meet computational demands on the network [2]. Component Object Model [3] provides an architecture that supports basic features as remote invocation, versioning, load balancing and fault tolerance. Distributed COM (DCOM) [4] is the distributed extension of COM. It specifies the additional infrastructure needed to further extend the benefits of networked environments. DCOM is a structure that allows applications to communicate with one another across a network. In particular, it allows for sharing of objects that reside on two separate machines. This means that one can create an object in one application, then call the methods of that object from an application that resides on a different computer. One of the most important properties of this technology is that it allows for the distribution of the load of a task across several machines.

Object Model

COM is set of rules defining object hierarchy. Methods are processes in procedure form, in object models. There are two sides in COM terminology: server and client sides. Server side contains objects which are created from the client side by means of interfaces. Processes in client side invoke objects' methods. An **interface** is a group of functionally-related abstract methods that must be located on both of server and client sides. It is a means of communication between sides. **Object class** is an implementation of one or more interfaces. It is located at server side, it is created by means of interface methods and its implemented methods are called from client side. Both interfaces and objects are identified by 128-bit identifiers called GUID (globally unique id). Interface identifier is a GUID known as interface ID (IID) and object class is identified by class ID (CLSID). An **object instance** is an instantiation of some object class. An **object server** is a running instance of a dynamic link library or an executable, capable of creating and hosting object instances of one or more classes. A **client** is a process that invokes an object's methods. IIDs and CLSIDs for specified objects are stored in a system registry file. In this file, they are matched with the location of the server program.

Binary Interface Standard

COM specifies a **binary standard** that provides the basis for reusing software components in their binary form at run time. This ensures dynamic interoperability of binary objects possibly built using different programming languages.

Programming Model

A typical client-server interaction in COM proceeds as follows: the client starts the **activation phase** by calling CoCreateInstance() with the CLSID of the requested object and the IID of the requested interface. A service **control manager** (SCM- located on both the client and the server sides) locates the server and transfers the request to the server SCM. The server creates an object instance, queries for the interface, and returns an interface pointer to the client. Meanwhile, SCM's on both sides create Proxy and Stub modules which establish RPC connection between client and server.

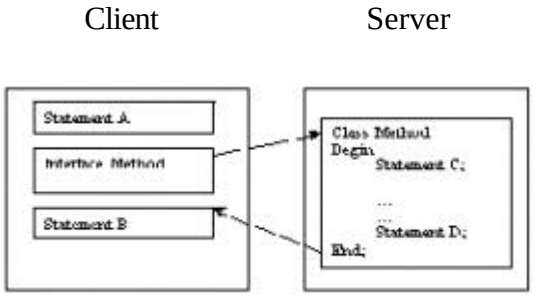
In the **method-invocation** phase, the client invokes methods of the interface through the pointer as if the object resides in its own address space. All interfaces must inherit from Iunknown which is an abstract base class with three methods: QueryInterface():for navigation between instances of the same object instance;

AddRef(): for incrementing reference counts; Release(): for decrementing reference counts. When the client finishes using an interface pointer, it calls Release() on the pointer.

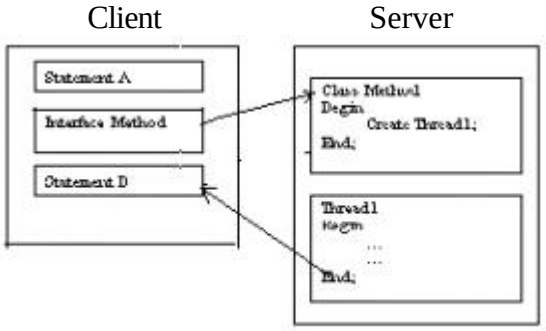
3. MODEL OF PARALLEL COMPUTATION

DCOM is the distributed extension of COM. It extends the remoting architecture across different machines. It is specified as a set of extensions layered on top of Distributed Computing Environment's Remote Procedure Call (RPC) Specifications [5]. RPC paradigm does not provide asynchronous communication. Therefore, a DCOM specific method call is synchronous (Fig. 2a). The caller must wait for the end of dispatching and returning calls. Due to the lack of asynchronous calls, one cannot dispatch and activate multiple processes simultaneously on a multi-computer environment because before dispatching a new process, the previous process must be completed. This prevents maximum parallelism. While DCOM presents threading models, they are not supported by several programming language compilers.

In this paper, we present a new model of parallel computation that uses operating system threads on top of DCOM (Fig.2b). The threads are created at the server side, in a server method called from client side. The task of this method is to create thread(s) on the server side and return synchronously to the caller. As a result, the threads begin execution in server processing space, while the caller on the client side continues its way without waiting for their termination. Thus, the caller may distribute its heavy tasks to several processors on the network, without waiting for them to be completed. The server thread completion may be informed to the client side by the inspection of the client or by an indirect server acknowledgment call made to the client. The client-inspection model is suitable for the parallel computation, because in case of a failure of one of the servers, the client can detect which server is down. The client checks the state of a server thread at certain time intervals or at a time convenient for it. The act of checking is accomplished by a client call to a server object method to inquire the value of a variable that denotes the state of the thread.



2a. Synchronous DCOM call



2b. Asynchronous mode of operation

Fig. 2. Synchronous and asynchronous modes of operation

Parallelization Technique

A network of processors can be used to solve distributed algorithms based on task parallelism, where the programmer defines different types of processes which communicate and synchronise with each other through message passing mechanisms over an interconnection network. In our model of parallel computation, we consider a parallel algorithm to consist of a collection of processes some or all of which can be executed in parallel on a number of available processors. On a DCOM based platform, a distributed algorithm has n server processes (objects) and one client process (object) to control these servers. Initially, all processors are free. The parallel algorithm (the client object) start execution on an arbitrarily chosen processor.

Shortly after it creates a number of computational tasks, jobs, to be performed by partitioning the problem into sub-problems that are as independent of each other as possible. Independence is very important, as an increase in dependency results in higher network traffic due to transmission of data between processes. Task to processor mapping is dynamic, depending on the system state. If a free server object is available, it is assigned the task, otherwise, the task is queued in a job queue to wait for a processor to be free. When a server object completes execution of a job, it becomes free. If a task in the queue is waiting to be executed, then it can be assigned to the processor just freed. Otherwise, the processor is queued and waits for a new task to be created. To exploit effective parallelism on the distributed system, the client executes the job assignment algorithm in parallel with the server objects. When server objects complete execution of a task, the client gets the partial results and recombines them to find the resulting output.

Parallel Hough Transform (PHT)

PHT exploits data parallelism: the problem is decomposed into smaller subtasks and these are mapped onto available processors on the network. The overall implementation is under the responsibility of a controller process, the client object, which performs run time partitioning of PHT data and allows the algorithm to execute, by dynamically assigning tasks (jobs) to available processors. When the client object initialises itself, it determines the number of computing nodes available on the network and creates as many server objects, before partitioning the problem into subtasks. For a given N*M sized image and radius r, the client divides the image into d*d sized sub-images and puts them into a job queue. Every server object executes a copy of the sequential Hough transform algorithm, each time with different input data, as long as waiting sub tasks exist.

Algorithm executed by server objects:

```
Receive d*d sized sub-image and r from client object.
Calculate the accumulator array A for that value of r.
Find local maxima in A.
Pass the results to client object.
```

Algorithm executed by client object:

```
Determine the number of processors available on the network (N).
Create server objects into an array server[] (#N servers).
Create sub problems and put them into a job queue.
While (queue not empty and not all servers have completed) do
  For i = 1 to #N do (for each server)
    If (Server[i] is down) then
      Put its job into queue again
    Else If (Server[i] is idle) then
      If (queue not empty) then
        Assign a job to Server[i],
      End If
    Else If (Server[i] has completed) then
      Get results from Server[i],
      Put new jobs into queue if any are created,
      If (queue not empty) then
        Assign a job to Server[i],
      End If
    End If
  End For
End While
Merge the partial results and find the final solution.
```

4. PARALLELIZATION ISSUES

Partitioning of the image: Hough transform is suitable for parallel implementation because the adjacency of the edge elements in the image are not taken into account during the calculations, and the edge elements can thus be processed in any order. One may consider dividing the image into non-overlapping rectangles which cover the whole image. With this approach, circles whose radii lie on the boundary of a rectangle may go undetected. Therefore, the image should be divided into overlapping rectangles with a minimum overlap size of radius r (Fig.3). This increases the processing time because overlapping sections of the image are processed twice, and even worse the corners are processed four times by server objects.

Partition size: Another important issue is the selection of the size d of the partitions (arrays) the image is divided into (Fig.3). If it is taken very small, there will be too many overlapping sections, resulting in processing duplication. On the other hand, selecting a very large size decreases the degree of parallelism.

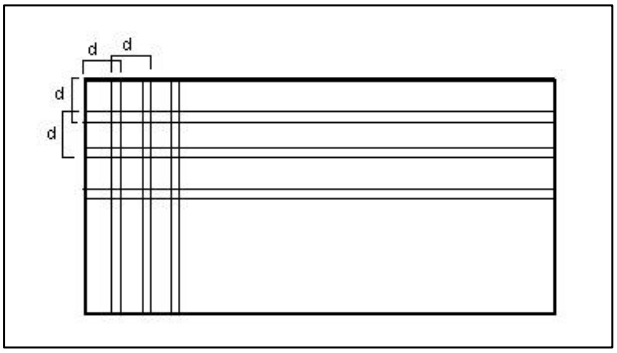


Fig. 3. Partitioning of the image into overlapping sub-images of size d .

Busy waiting loop: The client object does busy waiting, inquiring the state of the server objects. This has no direct effect on the processing time of the algorithm but induces an indirect effect because an overload on the network traffic is created by the communication activity. Therefore, we use a timer activated loop instead of a busy waiting loop in the client object. Every time the client is activated by the timer, it starts to execute the main while loop of its algorithm. Each time, it calculates an average server processing time t, which may vary according to the load of the network, and sets the next time interval to t. Thus, before starting the next polling loop, it suspends its execution for a time interval of t, and resumes execution at a point in time when the probability of the servers having completed their jobs will be high. This mode of operation eliminates unnecessary network traffic.

5. COMPLEXITY ANALYSIS OF PHT

The complexity of sequential and parallel Hough transform algorithm for a N*M sized image is calculated as the following , with

r : max radius to be found.
d : size of sub-image.
m : size of overlapping section.
N,M : image size.
n : number of parallel server objects.
C : complexity

Sequential processing: An N*M sized image is processed r times to find circles with the radii 1 through r. Thus, its complexity C is equal to Equ.(3)

$$C = r\,NM.$$

Parallel processing: If we select the size of the sub-image d as a multiple p of the length of the radius r, then, $d = pr$

Optimum case: Now, we assume we have as many server objects as the number of sub-images. In this case, we see that the optimum complexity is independent of the size of the image.

$$C = r\,NM = rd^2 = r\,(pr)^2$$

Real case: In this case, we have a limited number n of servers and partitions with overlapping sections of length m,

$$C = (1/n)\, r\, NM\, (d^2/(d-m)^2)$$
$$= (1/n)\, r\, NM\, (p^2r^2/(pr-m)^2) \quad [Eq.1]$$

Length of the overlapping section m is greater or equal to radius r. Taking m=r leads to the following equation which shows that complexity is inversely proportional to the number of server objects:

$$C = (1/n)\, r\, NM\, (p^2/(p-1)^2) \quad [Eq.2]$$

We know that the number of servers n is less that the number of partitions, that is

$$n < NM/(pr-m)^2,$$

This leads us to the equation which gives the optimum value of p for which the algorithm is expected to run at optimum speed.

$$p = 1/r\,(NM/n)^{1/2} + 1 \quad [Eq.3]$$

Experiments show that, on a MIMD architecture, network traffic and processing speeds of heterogenous computers executing the servers also play an important role on the total execution time.

6. EXPERIMENTAL EVALUATION AND RESULTS

We evaluated the parallel Hough algorithm on a MIMD platform of Windows NT Pentium processors each with 32MB memory, interconnected through a 100 Mbit Ethernet connection. For our evaluation, we used data for a 600*600 sized image with r = 25, to detect circles with radii between 1-25 pixels. We measured the processing time of the sequential algorithm as 107 seconds. Figure 4. gives the total processing and communication (network transfer) times for different numbers of server objects and for different values of d.

Partition size: We compared the effect of different partition sizes, using arrays with dimensions of 4, 6, 8, and 12 times the length of radius r, and using varying number of processors, between 2 and 12. Fig. 4 shows the results:

- partition size does affect speed-up.
- Optimal partition size is sensitive to processor number.

Number of processors: We evaluated the effect of varying number of processors. We varied the number of processors from 2 to 16. We observed that speed-up increases as the number of processors increases. The best processing time achieved is 13 seconds, with 12 server objects and the length of the partition d equal to 6r. Optimum value for d found from Eq.3 is 7*r. Therefore, we conclude that PHF almost attains optimum speed with 12 server objects executing on 12 different computers, and is about 9 times faster than the sequential implementation. We conclude that PHT exhibits a consistent and almost linear speed-up in parallel with the number of processors involved.

Server Objects	P	Total Time (sec)	Process Time (sec)	Network Transfer Time (sec)
2	4	75	64	11
	6	59	56	3
	8	49	48	1
	12	54	53	1
4	4	35	29	6
	6	27	26	1
	8	28	27	1
	12	31	30	1
8	4	19	11	8
	6	15	13	2
	8	16	15	1
	12	37	36	1
12	4	14	8	6
	6	**13**	**12**	**1**
	8	15	14	1
	12	44	43	1

Fig. 4. Processing times for different values of n and partition sizes (d=pr)

7. REFERENCES

[1] R.J.Schalkoff, "Digital Image Processing and Computer Vision", 1989, John Wiley & Sons, Inc.

[2] Y.Wang, and P.E.Chung, "Customization of Distributed Systems Using COM", IEEE Concurrency, Vol.6, No. 3, July-Sept. 1998, pp. 8-12.

[3] "The Component Object Model Specification" Microsoft Corp., Redmond, Wash., 1995 http:.//www.microsoft.com/com/.

[4] N.Brown and C.Kindel,"Distributed Component Object Model Protocol – DCOM/1.0", Microsoft Corp., 1998; http:.//www.microsoft.com/com/.

[5] "DCE 1.1: Remote Procedure Call Specification", The Open Group, Cambridge, Mass.,1997;http://www.rdg.opengroup.org/public/pubs/catalog/c706.htm.