SCCS: A SCALABLE CLUSTERED CAMERA SYSTEM FOR MULTIPLE OBJECT TRACKING COMMUNICATING VIA MESSAGE PASSING INTERFACE

Senem Velipasalar[*], *Jason Schlessman*[*], *Cheng-Yao Chen*[*], *Wayne Wolf*[*], *Jaswinder Pal Singh*[†]

[*]Princeton University, Dept. of Electrical Engineering, Princeton, NJ 08544
[†]Princeton University, Dept. of Computer Science, Princeton, NJ 08544
{svelipas, jschless, chengc, wolf}@princeton.edu, jps@cs.princeton.edu

ABSTRACT

We introduce the **S**calable **C**lustered **C**amera **S**ystem, a peer-to-peer multi-camera system for multi-object tracking, where different CPUs are used to process inputs from distinct cameras. Instead of transferring control of tracking jobs from one camera to another, each camera in our system performs its own tracking and keeps its own tracks for each target object, thus providing fault tolerance. A fast and robust tracking method is proposed to perform tracking on each camera view, while maintaining consistent labeling. In addition, we introduce a new communication protocol, where the decisions about when and with whom to communicate are made such that frequency and size of transmitted messages are minimized. This protocol incorporates variable synchronization capabilities, so as to allow flexibility with accuracy tradeoffs. We discuss our implementation, consisting of a parallel computing cluster, with communication between the cameras performed by MPI. We present experimental results which demonstrate the success of the proposed peer-to-peer multi-camera tracking system, with accuracy of 95% for a high frequency of synchronization, as well as a worst-case of 15 frames of latency in recovering correct labels at low synchronization frequencies.

1. INTRODUCTION

Reliable and efficient tracking of objects by multiple cameras is an important and challenging problem which finds wide-ranging application areas and has attracted much attention from the research community. However, using multiple cameras poses the additional challenge of establishing correspondences between moving objects in different views.

Many existing systems assume that multiple cameras are processed on a single CPU or by a centralized server. These are not scalable approaches. For a single CPU system, the amount of processing necessary to track multiple objects on multiple camera views can be excessive for real-time performance. Furthermore, scalability is debilitated as each additional camera imposes greater performance requirements. Server-based multi-camera systems are not practical in many realistic environments and have high installation costs. In general, multi-camera systems will operate on peer-to-peer computing systems where different CPUs are used to process different cameras. There may be significant communication delays between the CPUs, which necessitates the design of tracking algorithms requiring relatively little inter-process communication. Also, efficient protocols are needed which alleviate the problems caused by communication delays.

Chang et al. [1] propose a multi-camera system which is implemented on an SGI workstation. They use Bayesian networks to combine modalities for matching objects across multiple camera views.

Atsushi et al. [2] use multiple cameras attached to different PCs connected to a network. They use calibrated cameras and track the objects in world-coordinates. They send message packets between stations. Ellis [3] also uses a network of calibrated cameras. But, they do not discuss the type of communication, synchronization, or communication delays.

Nguyen et al. [4] propose a system using multiple cameras with processors embedded in them. A main controller on a PC is used to retain the current state of the scene. Once an algorithm on a camera terminates, the results are returned to the main controller. The controller and cameras use a TCP/IP based system to communicate. The cameras do not communicate directly, rather, they go through the main controller.

We present a scalable peer-to-peer multi-camera system and introduce: a) an improved, more efficient tracking algorithm that uses sparse communication, b) a novel protocol that coordinates multiple tracking components across the distributed system.

2. TRACKING IN SINGLE CAMERA VIEW

In the proposed system, we incorporate parallel tracking processors which communicate with each other as needed. That is, instead of a tracking task being transferred from one camera to the other, each camera performs its own tracking and keeps its own tracks for each target object. This provides improved tracking as well as distributed processing. Keeping tracks of objects in each view as long as possible also makes the system fault tolerant.

We utilize a robust and fast algorithm which efficiently tracks moving objects. First, foreground (FG) objects are

This work has been funded by NSF ITR grant 325119, NSF grant CCR-0329810, and ARO grant #W911NF-05-1-0480.

segmented from the background in each camera view by using the background subtraction (BGS) algorithm presented by Stauffer and Grimson [5]. Then, connected component analysis is performed, which results in FG blobs. When a new FG blob is detected within the camera view, a new tracker is created, and a mask for the tracker is built where the FG pixels from this blob and background pixels are set to be 1 and 0 respectively. The box surrounding the FG pixels is called the bounding box. Then, the color histogram of the blob learned from the input image is saved as the model histogram of the tracker. It is first determined if this object is visible by any other camera, as explained in Section 3.3. If the object is visible only by the current camera, the tracker is assigned a new label.

For each existing tracker, the algorithm evaluates detected FG blobs. If the boundary box of a blob intersects with that of the model mask of the tracker, the Bhattacharya coefficient between the model histogram of the tracker and the color histogram of the FG blob is calculated. The Bhattacharya coefficient $\rho(\mathbf{y})$ is defined as:

$$\rho(\mathbf{y}) \equiv \rho[p(\mathbf{y}), q] = \int \sqrt{p_{\mathbf{z}}(\mathbf{y}) q_{\mathbf{z}}} d\mathbf{z}, \qquad (1)$$

where $q_{\mathbf{z}}$ is the density function of the feature $\mathbf{z}$ representing the color of the target model while $p_{\mathbf{z}}(\mathbf{y})$ is the feature distribution of the FG blob centered at $\mathbf{y}$. The tracker is assigned to the FG blob which results in the highest Bhattacharya coefficient, and the mask of the tracker is updated. The Bhattacharya coefficient with which the tracker is matched to its object is called the *similarity coefficient*. If the similarity coefficient is greater than a predefined distribution update threshold, the model histogram of the tracker is updated to be the color histogram of the FG blob to which it is matched.

When objects merge, multiple trackers are matched to one FG blob, and the labels of all matched trackers are displayed on this FG blob, as shown in Figures 1 and 2, and the masks of the trackers are updated in the previously discussed fashion. The trackers that are matched to the same FG blob are put into a *merge state*, and in this state their model histogram is not updated. When objects split from each other, trackers are matched to their targets based on boundary box intersection and Bhattacharya coefficient criteria mentioned above.

There may be rare but unfavorable cases where a FG object, appearing after the split of merged objects, may not be matched to its tracker. We deal with these cases as follows: Let's denote two trackers by T_1 and T_2, and their target objects by O_1 and O_2 respectively. When these objects merge, $O_{1\cup 2}$ is formed, and T_1 and T_2 are both matched to $O_{1\cup 2}$. After O_1 and O_2 split, $B_{T_i O_j}$ are calculated, where $\{i, j\} \in \{1, 2\}$, and $B_{T_i O_j}$ denotes the Bhattacharya coefficient calculated between the histograms of T_i and O_j. Based on $B_{T_i O_j}$, both T_1 and T_2 can still be matched to O_2, for instance, and stay in the *merge state*. Let's denote the similarity coefficient of T_i by S_{T_i}. Thus, in this case, $S_{T_1} = B_{12}$ and $S_{T_2} = B_{22}$. This can happen because the model distributions of the trackers are not updated during the *merge state*, and there may be changes in the color of O_1 during and after the merge. Another reason may be O_1 and O_2 having similar colors from the outset. When this occurs, O_1 is compared against the trackers which are in the *merge state*, and intersect with the bounding box of O_1. That is, it is compared against T_1 and T_2, and $B_{T_1 O_1}$ and $B_{T_2 O_1}$ are calculated. Then, O_1 is assigned to the tracker T_{i^*}, where:

$$i^* = \underset{i \in \{1,2\}}{\operatorname{argmin}} (S_{T_i} - B_{T_i O_1}). \qquad (2)$$

If a FG blob cannot be matched to any of the trackers, and if there are trackers in the *merge state*, the unmatched object is compared against those trackers by using the logic in (2), which is also applicable if there are more than two trackers in the *merge state* as shown in Fig. 2.

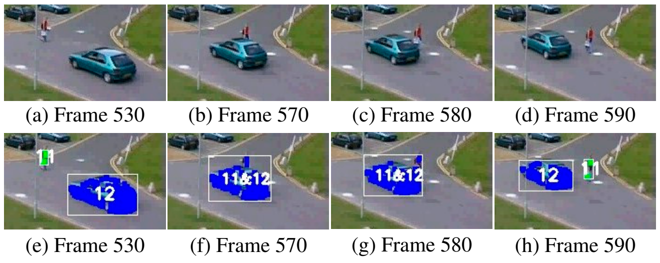


Fig. 1. Example of successfully resolving a merge. First and second rows show the original images, and the tracked objects with their labels respectively.

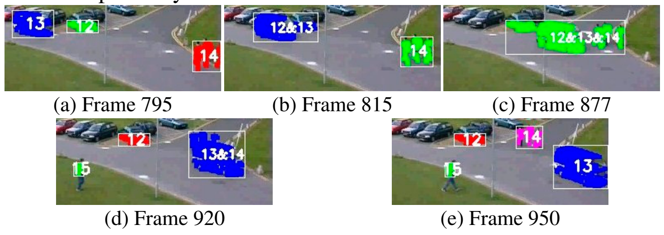


Fig. 2. Example of resolving the merge of multiple objects.

3. INTER-CAMERA COMMUNICATION

Our system provides a means for communication between the camera nodes by using the MPI library [7]. We take advantage of the proven usefulness of this library, and treat it as a transparent interface between the camera nodes. This reduces both the overhead and development time of a fully custom communication protocol. The proposed method is based on four primary design criteria: the communication infrastructure, the events during which data should be requested from other cameras, the data to be transferred between the cameras, and the points during execution at which these transfers should be made. In the following, we refer to camera C_i and C_j for a requesting and replying camera node, respectively. The block diagram in Fig. 3 illustrates the concepts discussed in this section. Note that at some point during execution each camera node can act as the requesting or replying node.

3.1. The Communication Protocol

Our protocol is designed to be efficient both by reducing the number of times a message must be sent as well as the message size. Furthermore, we utilize a *non-blocking* method of

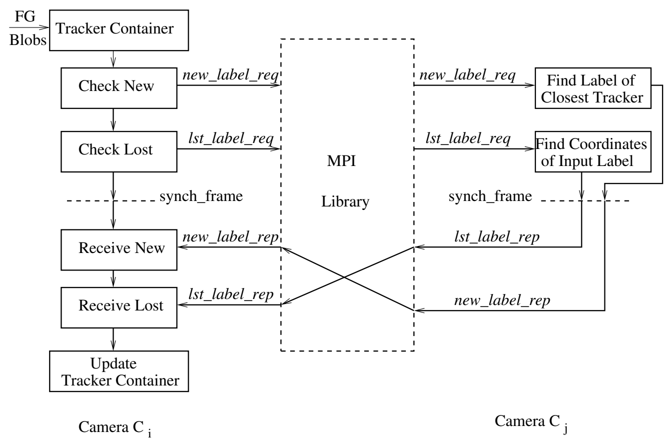


Fig. 3. Communication between two cameras.

communication. This effectively allows for a camera node to make its requests, noting the requests it made, and then continuing its processing, with the expectation that the requestee will issue a reply message at some point later in execution. Messages transferred contain tags specifying their type of command, with either *_req* or *_rep* appended, for requesting data or replying to a request, respectively. This is in contrast to *blocking* communication which suffers from two main drawbacks: first, the potential for parallel processing is reduced, as a camera node will be stuck waiting for its reply, while the processing program will likely require stochastic checks for messages. In the non-blocking case, checks for messages can take place in a deterministic fashion. Another drawback of the *blocking* communication is the increased potential for deadlocks and process starvation. This can be seen by considering the situation where both cameras are making requests at or near simultaneous instances, as neither can process the other node's request while each waits for a reply.

In addition, our protocol utilizes point-to-point communication, as opposed to some previous approaches that require a central message processing server. Our approach offers a latency advantage, as one less stopping point for a given message is necessary. Also, there is a scalability advantage, since for a central server implementation, the server quickly becomes overloaded with the aggregate sum of requests from an increased number of nodes.

Finally, our protocol sends minimal amounts of data. Messages consist of 256 byte packets, with character command tags, integers and floats for track labels and coordinates, respectively, and integers for the requesting camera id. Clearly, this is significantly less than the amount of data inherent in transferring streams of video or even image data and features. The protocol provides further efficiency by the nodes determining if a request should be made based upon relative field of view (FOV) line information discussed in Section 3.3.

3.2. Synchronization

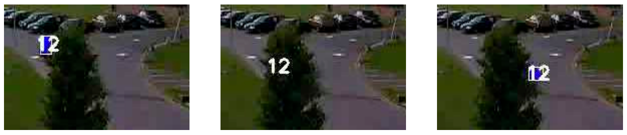
To the best of our knowledge, existing systems do not discuss how to allow for communication and processing delays. Even if the cameras are synchronized or time stamp information is available, communication and processing delays pose a serious problem for distributed camera systems. For example, if camera C_i sends a message to camera C_j asking for information, it incurs a communication delay. When camera C_j receives this message, it could be on a frame behind camera C_i depending on the amount of processing its processor has to do, or it can be ahead of C_i due to the communication delay. As a result, the data received may not correspond to data appropriate to the requesting camera's time frame. To alleviate this, our protocol provides *synchronization points*, where all nodes are required to wait until each node has reached the same point. During this time, message requests and replies are issued as needed, and reply data is used to update existing trackers. These points are determined based on a synchronization rate, *synch_rate*, where the camera nodes are synchronized once every *synch_rate* frames.

3.3. Request Events

A camera requests information from the other cameras when: a) a new object appears in its FOV, or b) a tracked object is lost. Our protocol refers to these events as *new_label* and *lost_label*, respectively.

In the *new_label* case, when a new object enters the FOV of a camera, it is possible that this object was already being tracked by other cameras. If this is the case, the camera will request the label of this object from those cameras, thus maintaining consistent labeling. If a FG object viewed by camera C_i cannot be matched to any existing tracker, a new tracker is created for it, and the visibility of this object by other cameras is checked using the *FOV lines*. The projections of the $3D$ FOV lines of camera C_j onto the image plane of camera C_i will result in $2D$ lines denoted by $L_i^{j,s}$, and called *FOV lines*, where $s \in \{r, l, t, b\}$ corresponds to one of the sides of the image plane. The FOV lines are recovered as described in our previous work [6]. The midpoint of the bottom line of the bounding box of the object is used as its location. If this point lies on the visible side of all $L_i^{j,s}$, it is deduced that it is visible by C_j. All the cameras that can see this object are found in the same way, and a list of cameras to communicate is formed. After ensuring C_j can see the object, the CPU processing C_i, sends a message to the one processing C_j.

C_i could also need information from another node when a tracker in C_i loses its target object, and this is called the *lost_label* case. A tracker can lose its object due to errors in BGS or if its object is occluded. For every tracker that cannot find its match in the current frame, it is first checked if the object of that tracker has left the FOV of the camera. This is done by checking the direction, duration and the distance of the tracker to image boundaries. If the target object is not out of the view, the cameras which can see the most recent location of the tracker are found as in the *new_label* communication scenario. If it is deduced that the object is visible by C_j, a message, containing the label of the lost object, is sent to the CPU processing C_j. The most recent location of this label, in the coordinate system of C_j is received as the reply. Then, the corresponding point on the view of C_i is calculated as described in [6], and the location of the tracker is updated. As seen in Fig. 4, a car begins to go behind a tree in frame 372, and as a result will not be seen by the current camera. In frame 386, the car's location can still be updated, thanks to receiving its coordinates from the other camera. At frame 388, the car is seen again, and its tracker catches up to its object.



(a) Frame 372 (b) Frame 386 (c) Frame 388

Fig. 4. Example of successfully recovering occluded object.

4. EXPERIMENTAL RESULTS

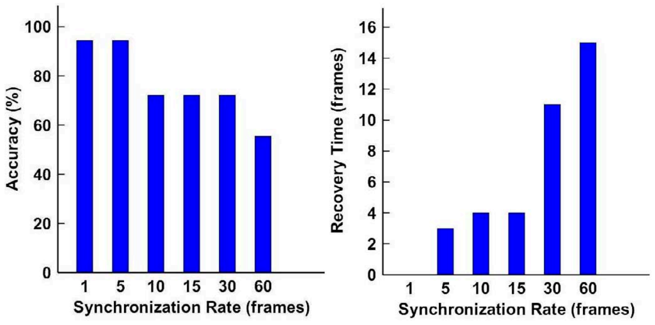
The proposed approach was implemented using a computing cluster containing uniprocessor compute nodes. Our sample input consisted of video sequences from the PETS2001 database. A C/C++ implementation was developed, using LAM/MPI [8] for communication. The rate of synchronization was varied, with assessments of vision performance acquired relative to the performance of the single processor case. The accuracy percentage with respect to vision performance was determined by the following formula:

$$accuracy = \frac{\#correct\_updates}{total\_requests} * 100 \tag{3}$$

where *#correct_updates* represents the number of times a new or lost label request is correctly fulfilled. The determined accuracy values are shown in Fig. 5a. As can be seen in the figure, the system achieves 95% accuracy in the single frame synchronization case, with levels monotonically decreasing with increasing synchronization rate. Further, even with allowing the processors to operate up to 2 seconds without communication, a level of 55% accuracy is still attained.

In addition to accuracy, the recovery time for unfulfilled new and lost label requests was determined. This represents the number of frames a node must wait before acquiring the requested information from the other node. Consider the case of a *new_label_req* by camera C_i to camera C_j. For *synch_rate* S, in the worst-case, S frames of recovery time would be necessary prior to C_i receiving its desired *new_label_rep* from C_j. Average values for this recovery time are shown in Fig. 5b. As can be seen in the figure and as expected, recovery time increases with increasing synchronization rate. It should be noted that even when allowing for up to 2 seconds of recovery time, the average wait is one-fourth of this.

Regarding speed, our parallel implementation allowed for a system that can handle more than two video sources, while providing a speedup of 3.05 times relative to an initial implementation running on a single uniprocessor node. Clearly, this is more ideal. Furthermore, the presented protocol is scalable, and thus ameliorates the speed limitations imposed by single processor implementations.



(a) Accuracy (b) Recovery Time

Fig. 5. Performance Results.

5. CONCLUSIONS

As the previous section indicates, our approach to multiple-camera tracking shows promise. It is our belief that by combining effective algorithms, robust communication protocol based on proven infrastructure (MPI), and provision of synchronization, our system obviates previous multi-camera system design issues. Further, our experiments provide analysis of synchronization frequency and communication issues. The addition of distinct processors for video sources, with the ability of both individual tracking as well as tracking with assistance from other cameras, provides a robust and fault-tolerant system. The newly designed protocol allows for non-blocking communication, such that processors are not required to wait for replies to requests. This is accomplished without deterioration of vision algorithm performance. Furthermore, our approach and system allow for greater security by reducing the amount of transmitted data, which in turn requires less energy for communication.

6. REFERENCES

[1] T. -H. Chang, and S. Gong, "Tracking Multiple People with a Multi-camera System ," *Proc. of IEEE Workshop on Multi-Object Tracking*, pp. 19-26, 2001.

[2] N. Atsushi, K. Hirokazu, H. Shinsaku, and I. Seiji, "Tracking Multiple People using Distributed Vision Systems" *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 2974-2981, 2002.

[3] T. Ellis,"Multi-camera Video Surveillance,"*Proc. of Int'l Carnahan Conf. on Security Technology*, pp. 228-233, 2002.

[4] K. Nguyen, G. Yeung, S. Ghiasi, and M. Sarrafzadeh, "A General Framework for Tracking Objects in a Multi-Camera Environment,"*Proc. of Int'l Workshop on Digital and Computational Video*, pp.200-204, 2002.

[5] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking, "*IEEE Int'l Conf. on CVPR*, vol. 2, June 1999.

[6] S. Velipasalar and W. Wolf, "Multiple Object Tracking and Occlusion Handling by Information Exchange between Uncalibrated Cameras", *Proc. of IEEE ICIP*, pp. 418-421, Sept. 2005.

[7] The MPI Standard, *http://www-unix.mcs.anl.gov/mpi/*.

[8] LAM/MPI Parallel Computing, *http://www.lam-mpi.org/*.