REAL-TIME MOTION TRACKING: A CASE STUDY IN PARALLELIZING VISION ALGORITHMS

By

Scott Bulmer

B. Sc. (Computer Science / Mathematics) University of Victoria, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

FEBRUARY 1995

© SCOTT BULMER, 1995

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date February 22, 1995

DE-6 (2/88)

ABSTRACT

For real-time motion tracking the computational requirements suggest a parallel processing solution be followed. This thesis describes the implementation of a parallel motion tracking system. Focus is placed on the resulting system as well as some of the factors involved when performing a conversion from an existing sequential system. Different parallelization methods are compared along with many of the other related issues involved in implementing vision systems on multiple processors. The model-based motion tracking system uses detected line segments from the image as features, and tracks these features with a least-squares solution to match the model with the correct feature configuration. A pipelined multiprocessor system was constructed which is able to achieve a high level of throughput using many features of the TMS320C40 parallel processors. The system consists of six C40 processors which communicate through ports controlled by a software router. A "handshaking" scheme is employed to coordinate data transfers between processors. The motion tracking system was able to track moving objects at the rate of about 4 frames per second.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I owe a great deal of gratitude to my thesis supervisor, Dr. David Lowe, for the completion of this work. He provided considerable advice, guidance, and support, as well as helping to maintain focus on the topic. I would also like to thank Dr. James Little for taking the time to be the second reader of this thesis, and for initiating my interest in computer vision.

Thanks also to Rod Barman, who provided continued hardware support and advice, and to Stewart Kingdon, who supplied the assembly language convolution code and other software support.

When I started my graduate program at UBC, I had really no idea about the kind of work and effort needed to complete a Master's thesis. I owe a special thanks to my family and friends who have stood by me while I discovered what it was actually like to be a graduate student. They have sacrificed a lot and have provided much needed encouragement.

Thank you to everyone who has contributed to make my experience at UBC a very rewarding one. I have certainly learned a lot and have benefitted greatly from the entire experience.

Chapter 1
Introduction

Computer vision is concerned with the extraction of information about a scene from its images. One important piece of information is the location of objects of importance. With static images, the location task is arduous enough since no motion cues are present to aid in finding interesting things in the image. However, with moving objects things become a lot more complicated. A significant amount of computation time is still required to locate the object in each image, but suddenly there is also a limit on the amount of computation time that can be allotted to any single image, since the object is moving and could be lost while conducting a search. Thus only a certain amount of time can be spent finding the object in any image, or the processing must be restricted to a certain subset of the images received.

Motion tracking presents a solution to this problem. Instead of locating the object in each image frame, the object is *tracked* through ensuing images. This procedure makes the problem more tractable. In this way motion tracking differs from tasks such as object recognition; the continuity between successive images can be used to give an approximate location of the object in the scene.

If the location of the object is known in a previous frame, tracking the object reduces to simply searching for the object in a small area around the previous location. This saves a lot of computation time since the entire image does not need to be searched. Of course, as the object is allowed to move with increased velocity, the size of the search area must also increase since the spatial disparity between corresponding image features becomes larger. Although position estimates based on velocity predictions can be used, a limit is usually placed on how far the object can move between frames. This is commonly a certain number of pixels per frame.

For real-time motion tracking, however, the computation time is usually still too high to process every incoming image. Therefore, only a certain number of images can be processed. Care must be taken, of course, that the temporal gap between images does not become so great as to risk losing important information about the scene and the objects it contains. Sampling the images in larger intervals can also lead to difficulty in predicting object motion since the range of potential motion increases. The number of images used is directly related to the throughput of a real-time system, hence an attempt should be made to process as many incoming images as possible.

The prevailing computational conditions (speed requirements, complexity, and data volume) require that a different strategy be taken for motion tracking to become a feasible real-time operation. Specially designed hardware can be used to aid this endeavor, but a more economical and flexible solution is gained through the employment of parallel processing techniques. Parallel systems provide the resources necessary for real-time operations of this nature.

This thesis will focus on the implementation of a parallel motion tracking system and some of the factors involved when performing a conversion from an existing sequential system. This work is based on a system created by Lowe [43, 44]. Different parallelization methods will be discussed, as well as many of the related issues involved in implementing vision systems on multiple processors. The processors used in this system are the Texas Instruments TMS320C40 processors. The configuration of the motion tracking system is shown in Figure 1.1. This figure displays the actual physical connections in the system and does not infer anything about the information flow present. (Figure 4.4 depicts the flow of data and information in the system.) Six C40 processors are connected in a network which is linked to a host Sun workstation. The network was designed so that there is at most one intermediate node between any two C40s. This will simplify the router's work in transferring information between processors. Images are received from a CCD camera through a frame grabber. The frame grabber is physically connected to one of the C40 nodes, so image acquisition must take place on this particular node. Another node is connected to a frame buffer, so the system's display output is performed from this node to a monitor via the frame buffer.

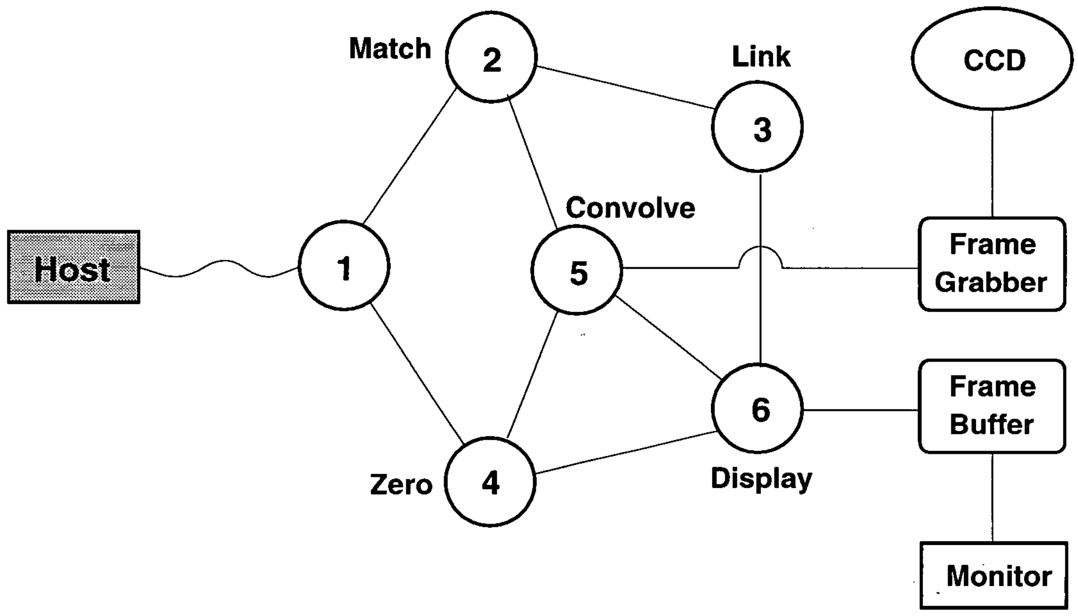

Figure 1.1: C40 Network configuration.

Figure 1.1 also shows the tasks which are necessary to perform in this motion tracking system, and their mapping onto the C40 network. Three nodes handle the edge detection process using a thresholded, magnitude-of-gradient operator for use in Marr-Hildreth edge detection [46]. A fourth node creates correspondences between these edges and an object model in a matching process. Since the object motion can be arbitrary, the resulting model pose disparity is unpredictable. Hence a position prediction is not made at this stage based on trajectory estimates. A final processing node is responsible for displaying the tracking results. These tasks will be discussed in detail in Section 4.2.

Chapter 2 will describe motion tracking in greater detail and will present a number of possible applications and previous approaches that have been taken. Chapter 3 will discuss different parallelization methods. Chapter 4 will present the implementation of the system, describing the C40 processors and the issues confronted during the conversion. The execution of the system will be examined in Chapter 5, tabulating some of the results obtained. Finally, Chapter 6 will summarize the findings and will outline some of the future improvements which could be made to this system.

Chapter 2
Motion Tracking

This chapter will further the discussion on motion tracking and will discuss possible applications for this type of work. Some of the previous approaches which have been taken to this problem will also be presented and reviewed.

2.1 Description of Motion Tracking

"3D model-based vision" is concerned with finding the occurrence of a known 3D object within an image, and obtaining a measure of the object's location in the image. The existence and location of the object can then be used for tasks such as robotic manipulation, process monitoring, vehicular control, and other applications discussed in Section 2.2.

"Model-based tracking" is model-based vision applied to a sequence of video images. It appears initially to be a much more computationally intensive problem than model-based vision, due to the high data-rate in an image sequence. However, the continuity between successive images can lead to it being a less cumbersome task, because the position of the object can be anticipated with some precision. Since there is the physical constraint that most moving objects change visible structure only very slowly over time (or not at all for rigid objects), tracking a small number of features into the next image gives us information about where the other features should appear.

Some definitions must be made at this point in order to further the discussion on motion tracking. A *feature* represents a pattern in an image and is characterized by a *feature vector* whose value denotes the state of the pattern in the signal at any instant. For a visual tracking system with a 2D image input, the features can be patterns such as points, line-segments, or vertexes and attached edges. A *feature configuration* is a particular arrangement of feature vectors.

As only certain aspects of the object (such as the geometry of its edges) are utilized, these aspects form a model of the object and it is the occurrence of the model that is sought. Hence, a *model* can be described as a set of parameters representing the changeable state of an object, together with geometric descriptions of fixed properties of the object. Each local model part which can correspond to a single image feature is called a *model feature*. A model *pose* is a certain configuration of its parameters which is represented by patterns in the image.

A geometric model is attractive to work with, because the strong geometric invariances under perspective projection can provide reliability and computational simplicity. The utility of a geometric model will be greatest for rigid objects, and for those that are simply joined (e.g., a box with a hinged lid) or simply parameterized (e.g., a telescoping tube). The geometry of flexible, malleable, and generically described objects is harder to use. Non-geometric models, utilizing such attributes as colour and texture, may serve to reveal the existence of the object, but not a quantitative measure of its 3D location.

The geometric models used for tracking must be simple to extract (i.e., computationally cheap) if processing is to proceed at near video-rate. They must also be reliably present in the images. Computationally expensive and unreliable model features, such as closed regions representing surfaces, cannot be afforded. This indicates the use of simple local features such as points (or "corners") and edges.

Edges can generally be extracted more cheaply and reliably from images than point features. Straight edge segements also form good features since linearity is invariant under perspective projection, and many strong geometric invariances can be used. However, unless specialized image-processing hardware is used, straight-edge extraction over the entire image will be much slower than video-rate. Other concerns apparent when using edges as image features include fragmentation due to drop-out or clutter, and incomplete and variable termination at junctions.

A brute-force method of solving the motion problem, though not efficient, is to solve the 3D object recognition problem separately for each successive image frame. This would allow the position of the object to be known at certain time intervals, and other properties such as the rate of motion of the object could be computed using the time interval between frames. However, the recognition aspect of the method makes it computationally intensive and would not result in a real-time solution. By the time the position of the object is determined for one frame, it may have moved a significant distance. This method does not take advantage of "temporal coherence"; knowing where the object is in a previous frame gives great insight into where it can be in the next frame, provided that the interval between frames is not large.

Another more efficient method for solving the motion problem is to compute only the correspondence between successive frames, that is, determine the change in the object's position and orientation from one frame to the next. Assuming that the object (represented by a model) corresponds to a set of simple features in the image, a change in the configuration of the features denotes a change in the position of the object (and hence, the model). What must be done here is to establish some sort of relationship between the changes in image feature configurations and changes in poses of modeled structures. This relationship is bi-directional: one can either determine the pose of a model from the configuration of the image features, or one can extract the image features corresponding to a particular model pose.

Hence there are two possibilities for exploiting the known position of an object in a frame: it can allow the use of image-based techniques in tracking the object into the next frame; or it can constrain a model-based search for the object in the next frame. The former is faster, but can fail; the latter allows recovery from such failure and is not overly complicated since perspective projection is relatively easy to perform. This in turn leads to two general solution paradigms [73]:

(i) *motion-searching*, which hypothesizes and tests 3D model poses, and

(ii) *motion-calculating*, which uses back-projection to directly estimate the model pose from changes in image feature configuration.

2.2 Applications of Motion Tracking

The techniques described in the previous section can surprisingly be applied to many applications in a diverse range of fields. These varied usage possibilities confirm that motion tracking research is a very worthwhile endeavor.

One application for a cheap, robust, fast object tracker is for the visual control of a robot [62]. Currently, the position of a robot's arm or grasping device is deduced from sensors positioned on the joints, and is susceptible to a build-up of errors since the arm is not perfectly rigid. Visual measurement of the position of the end-effector would be independent of the deflection of the arm. The removal of the joint sensors would mean that the arm could be made of a lighter and more flexible material. This would be highly desirable for the purposes of controlling the robot, and would allow much faster and better controlled movements to be performed.

Motion tracking could be used by the Space Shuttle to retrieve a satellite [23]. If the satellite is tumbling, it is necessary to match the motion of an arm to the motion of the satellite, so that the arm can grasp the satellite and slowly bring it to rest without producing excessive forces and torques. Other possible applications exist in assembling structures in space, since pieces of structure may be floating in space or may be moving due to flexure in a large structure.

The abilities of motion detection, tracking, and estimation are essential to many automation tasks such as traffic flow monitoring and control, traffic speed monitoring, and tollway billing [63]. Moving target detection and tracking is naturally of great interest to the defense industry [75]. In coding of image sequences for transmission, effective bandwidth reduction can be achieved through the estimation of motion in the image sequence. A large reduction of bandwidth is particularly in demand for video conferencing and videophone systems. Estimation, characterization, and understanding of human motion in athletic activities, dancing, pilot training, and patient rehabilitation are of interest in several disciplines. The analysis of human heart motion allows diagnosis of heart problems [69], and other anatomical structures can be tracked in time sequences of magnetic resonance or ultrasound medical images [3]. Velocity estimation for fluid flow is of interest to meteorology and experimental fluid mechanics with applications ranging from weather forecasting to aircraft design. Obstacle detection by a moving vehicle is necessary for collision avoidance and path planning for autonomous navigation. Further applications include hand-eye coordination for robots [1, 4] and road-following [14, 51].

2.3 Previous Work in Motion Tracking

Many different approaches to the motion tracking problem have been taken. Originally, motivation for studying two-dimensional motion came from the desire to analyze a vast quantity of satellite images of clouds [21, 36]. Many of these early attempts, however, were more concerned with simply detecting the motion or separating objects of interest from the background without actually tracking the motion of a certain object. Soon work was focused on actually following the motion of an object, but initially tracking systems concentrated on line drawings or other similar input images.

Early Efforts

Tsuji, Osada, and Yachida [70], produced a dynamic scene analyzer which separated moving objects from the background and followed their motion patterns in dynamic line images such as cartoon films. They used flexible templates to analyze the motion of non-rigid objects.

Asada *et al.* [2] tracked moving objects in a scene which contained only polyhedral objects (i.e., blocks-world scenes). 3D polyhedral models were employed using vertices as features. The input was assumed to be perfect line drawings, and legal correspondences were labeled in the image before processing. Problems of feature extraction and configuration were not addressed.

Roach and Aggarwal [57] used a modified least-squared error method to determine the three-dimensional model and motion of an object. This approach required two or three views of the object, and ignored the low-level processing tasks of extracting feature points.

Other early efforts in motion tracking handled either a sequence of stored images, or a number of "snapshot" images with large temporal gaps in between. In either case, these attempts did not provide real-time processing. Other systems performed more "recognition-like" tasks, relocating the object in each input image.

Gilbert *et al.* [25] produced a real-time video tracking system (RTV) for missile and aircraft identification and tracking. This system analyzed motion from videotape.

O'Rourke and Badler's visual tracking system [54] used a 3D model of the human body to predict and track image sequences of human motion. For each image their system generated predictions of the 3D structure of the human body, and then searched for predicted features in certain regions of interest. They assumed that tracking a few locally-distinguishable features (e.g., hand and foot) would be enough to determine the complete model parameters. The Alven system [69] is similar to O'Rourke and Badler's system. The positions of markers (point features) in the image of a beating heart individually determined the parameters of the model.

KALMAN FILTERING

The use of Kalman filtering theory has been another popular approach to tracking. For the purposes of real-time tracking, it is essential that the estimation of model poses proceed sequentially as new observations become available. This is the premise of Kalman filtering theory. The Kalman filter operates by continuously updating an estimated state vector and an error covariance matrix. The Kalman filtering approach was first applied to the incremental estimation of rigid object motion by Hallam [26], and has also been used in solutions by Broida and Chellappa [7], Gennery [23], Schick and Dickmanns [59], Harris [28], Terzopolous and Szeliski [65], and Matthies *et al.* [50].

Deriche and Faugeras [18] built a line segment based token tracker. Line segments were tracked using a parameter prediction system, attaching a kinematics model to each parameter of the object representation. A prediction is made, based on this model, of the position of the parameter in the next frame, and then a search is conducted in this neighbourhood. A Kalman filter provides estimates of regions where the matching process has to seek for a possible matches between tokens.

RAPiD (Real-time Attitude and Position Determination), developed by Harris [28], is a model-based three-dimensional tracking algorithm for a known object executing arbitrary motion and viewed by a standard video-camera. The 3D object model consists of selected control points on high contrast edges, which can be surface markings, folds, or profile edges. The use of a Kalman filter permits rapid object motion to be tracked, and produces stable tracking results. In uncluttered situations, RAPiD has maintained object tracking at object rotation rates of 10 rad/s and angular accelerations of 100 rad/s^2.

Schick and Dickmanns [59] use a generic parameterized model for the object types. They solve the more general problem of estimating both the motion and the shape parameters. The motion model of a car moving on a clothoid trajectory is applied including translational as well as angular acceleration. The estimation machinery of the simple extended Kalman filter (EKF) is used. So far, however, their approach has only been tested on synthetic line images.

Matteucci *et al.* [49] use a Kalman filter approach in a real-time surveillance system in applications where unknown objects entering the working area of an autonomous vehicle have to be detected and tracked. Subareas in the image are identified where changes have been detected from a background image. Feature correspondences between two successive frames are created and a tracking module uses a Kalman filter to measure and predict quantities relating to the postition of the object.

Motion-Searching Methods

Motion-searching involves utilizing object models to facilitate a search for possible object positions. Because a model-based approach is able to exploit global attributes of the object being tracked, it can provide significant advantages over purely local methods for situations in which the environment is cluttered, there are multiple moving objects, or there may be a large motion of an object from one frame to the next.

Algorithms have been developed which do handle the full generality of the 3D model-based visual tracking problem. Many recent motion tracking systems have concentrated on model-based approaches, and have stressed the importance of throughput or computational speed.

To aid in the processing speed of the systems, several parallel architecture systems have been created.

Thompson and Mundy [68] perform a search for the correct model pose which "spirals" outward from a best-guess model pose hypothesis based on trajectory estimates (see Figure 2.1). The subsequently generated candidate hypotheses are tested using a pose-clustering technique. Good matches between image and model vertex pairs add up "votes" for poses. Dense clusters of such votes indicate global consistency, and the centers of such clusters constitute the model pose which corresponds to the image feature configuration. Thompson and Mundy's approach

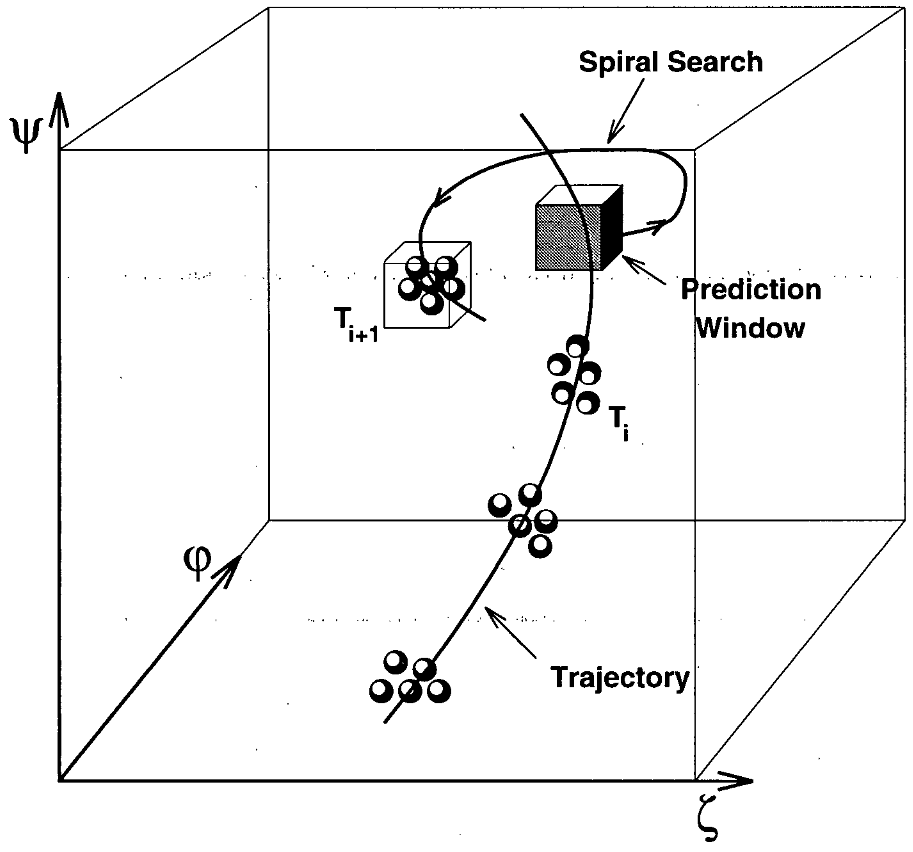

Figure 2.1: Thompson and Mundy's search strategy.

suffers, however, because the voting procedure has similar space and time limitations that Hough transform algorithms have. The system also allows the pose of an object in consecutive images to vary by as much as 30 to 40 degrees. This means that the results obtained from the tracking algorithm do not portray a continuous representation of the object motion.

Tracking systems designed by Verghese, Gale, and Dyer [73] place great importance on using the "spatiotemporal image assumption", which requires that the temporal sampling rate of images be high with respect to the rate of spatial changes in the image. Adhering to this assumption means that the object pose can change only slightly in successive frames, and that the image features will persist over time and will change their geometric properties only very slowly. They have devised two different methods which use this assumption.

Their first method uses the motion-searching paradigm, and requires real-time processing to be performed on an infinite input stream of images, producing a corresponding output stream of model poses. The throughput needed to satisfy the spatiotemporal image assumption is obtained through the use of parallelism. New object pose hypotheses are generated and tested in parallel by searching for the correct model pose in a neighbourhood around the previous model pose. These hypotheses are correlated with a stream of edge-detected feature maps by performing a logical "and" between the edge points and the model points projected from each hypothesis.

A gradient-ascent algorithm is used by Worrall *et al.* [76] in order to estimate the pose of a known object in a car sequence. Since no motion model is used, the previous estimate is used at every time instant to initialize the iteration. Marslin, Sullivan, and Baker [48] have enhanced the approach by incorporating a motion model of constant translational acceleration and angular velocity. Their filter optimality, however, is affected by use of the speed estimates as measurements instead of the image locations of features.

Koller, Daniilidis and Nagel [35] detect and track moving vehicles from road traffic scenes. They used a parameterized vehicle model for an intraframe matching process, and a recursive estimator based on a motion model is used for motion estimation. An illumination model was required in order to avoid incorrect matches between model segments and image edge segments due to shadows. The matching of image edge segments is based on the Mahalanobis distance of line segment attributes as described by Deriche and Faugeras [18]. This system also calculated initial guesses from grouping nearby optical flow vectors.

Morgan *et al.* [51] guide the tracking of edge points through a sequence of images using a simple model of a pair of road edges. An oriented "edge-finder" operator is described that determines the edge point positions which are the data for a weighted least-squares curve fitting process to determine the parameters of the model at each frame. Edge detection by correlation with an edge template is performed after predictions from previous knowledge is stored as a model. A circular arc model representation of a simply curving road is used, and guides selective image processing.

An innovative system has been developed by Huttenlocher *et al.* [30], which addresses many of the problems involved in motion tracking. The technique is model-based, but differs in that the model can dynamically change shape and an *a priori* description of the model is not even required. The idea is to split the image of a solid object moving in space into two components:

- a two-dimensional motion in the image, corresponding to the motion of the visible aspect of the object, and
- a two-dimensional shape change, corresponding to a new aspect of the object becoming visible or an actual shape change of the object.

The object can move about in the image in an unconstrained manner, providing its shape does not change too much from one image to the next.

The model of the object is acquired dynamically from the image sequence. It is represented as a sequence of binary images, one corresponding to each frame of the input image sequence, so that the model is a subset of the image features at any time. The model changes from one frame to the next as the object's shape in the image changes. The binary images are compared using methods based on the Hausdorff distance to determine the new location of the object in the next image. The models used in this method are 2D models providing no 3D object location information. This method is not real-time and a trade-off also exists between the ability to track non-rigid objects and tracking the background.

Motion-Calculating Methods

Motion-calculation methods do not resort to an expensive model search to locate the object. Instead, image-based techniques are used to implement edge tracking, from which the model parameters can be reconstructed. These techniques usually have the drawback, however, of not possessing an efficient method for error recovery, such as is evident in the motion-searching processes.

Bray [6] created a system which was able to track polyhedral objects in 3-D space using optic flow techniques. Line segments are tracked via a flow field computed between consecutive frames. This takes advantage of surface texture and other contextual information. After taking as input a precise position for the object, it projects the model at this position and uses image-based techniques to predict features in the image. A recovery stage was implemented which takes over processing if the tracker gets lost.

Stephens [62] describes a system which uses a technique of localized image access that enables it to achieve frame-rate operation without the need for any special-purpose hardware. The direct access method also avoids the need for image segmentation. Multiple cameras and a network of five transputers per camera are used to predict the location of a number of edges, and then process the image in small regions to locate edges close to the predicted positions. The lateral displacements of these edges from the predicted positions are collected from the cameras, and they form a set of constraints which are solved using a Hough transform. Hough transforms are intrinsically parallel, and they provide an efficient way of combining information from two or more cameras. A central processor combines the Hough spaces from the different cameras, and interprets it to update the position of the object. Stephens' system has been demonstrated running in real-time at about 10 Hz.

Gennery [22, 23] has proposed another approach for tracking 3-D objects of known structure. He uses a "feature tracker" to track a number of individual features, such as the vertices of a polyhedron. Another module then uses the information from a number of cameras to compute the three-dimensional positions of the features. This method has the advantage that, even if only a few features are seen by two cameras, this may be enough information to determine the absolute three-dimensional information for all of the features, even those that are seen only by one camera. The three-dimensional position and orientation parameters of the object are determined by a model matcher, which creates correspondences between the image features and a 2D projection of a predicted model pose. Finally, new data from the feature tracker (running concurrently with the other modules) is used to update the object position and orientation to the time of the most recent data.

Verghese, Gale, and Dyer's second motion tracking approach [73] performs spatiotemporal tracking by determining the model pose directly from the feature configuration disparity. Each individual 2D model feature is tracked directly to their dynamically evolving 2D image features. Edge maps are passed to a feature tracker which keeps track of the position of each model edge. The edges are tracked by performing a 2D search in the area around the previous position. Once the position of all of the edge features is determined, a back-projection process is performed to compute the model pose for that feature configuration. The back-projection procedure is based on the Newton-Raphson iterative technique described by Lowe [41]. Because errors in computation may invalidate some of the feature vectors (because of noise, occlusion, disocclusion, and proximity of identical local features), these conditions are detected and corrected by a validation process. This process will reinitialize the feature tracker when errors occur.

Other motion-calculating approaches have been followed by Murray, Castelow, and Buxton [52]; and Crowley [15]. Crisman [14] has developed a system which specializes in detecting unstructured roads and can detect intersections without map shape and location information. It tracks the road by classifying pixels based on their colour.

Active Contours

One of the more recent approaches to tracking involves utilizing active contour models, or snakes, to aid in the tracking process. These deformable contours are actually energy-minimizing splines which are guided by external constraint forces and influenced by simulated image forces that pull it toward features such as lines and edges. The snakes are 2D models which provide no 3D information.

The idea of tracking objects in time-varying images using snakes was originally proposed by Kass *et al.* [33]. Kass was seeking to design an energy function whose local minima comprised the set of alternative solutions available to higher-level processes. These energy functions were then applied to track a speaker's lips through dynamic forces generated from the images. Once a snake finds a salient visual feature, it "locks on". If the feature then begins to move slowly, the snake will simply track the same local minimum.

Terzopolous and Waters [64] describe a more thorough tracking of articulate facial features using multiple snakes. Since snakes conform readily to complex biological structures, many applications have been in the area of biomedical image interpretation [3, 37].

Cippolla and Blake [12, 13], and Curwen *et al.* [16] use real-time snakes to track the occluding contours of 3D objects as seen from a camera attached to a moving robot arm. These applications have demonstrated that snakes are very well suited to tracking rigid and, especially, nonrigid objects. As it tracks an object of interest, a snake can provide detailed quantitative information about its position, velocity, acceleration, and its evolving shape in the image plane.

Terzopolous and Szeliski [65] construct "Kalman snakes" by constructing continuous Kalman filters that incorporate the dynamic snake into their system and prior models.

Yuille and Hallinan [77] combine snake and model-based methods in their motion tracking approach, exploiting advantages of both. A deformable template is used for feature recognition. A flexible geometrical model and an imaging model are then used to detect a variety of examples of the same basic feature under different light conditions. They show how this can be applied to eye detection and high energy particle detection.

Curwen and Blake [17] achieve video-rate tracking performance using dynamic contours combined with parallel computing in the form of a small transputer network. Dynamic contours are elastic, as snakes are, but defined parametrically using B-splines [16]. They have demonstrated the applications of such tracking in surveillance of people and vehicles, robotic path-planning, and grasp-planning.

A unique approach to the problem has recently been taken by Dickinson *et al.* [19]. This method is able to perform tracking with little or no knowledge of the 3-D object, but differ from active contour methods in that it is able to track not only the position of the object, but also the orientation. The tracking is performed by detecting visual events in the image, such

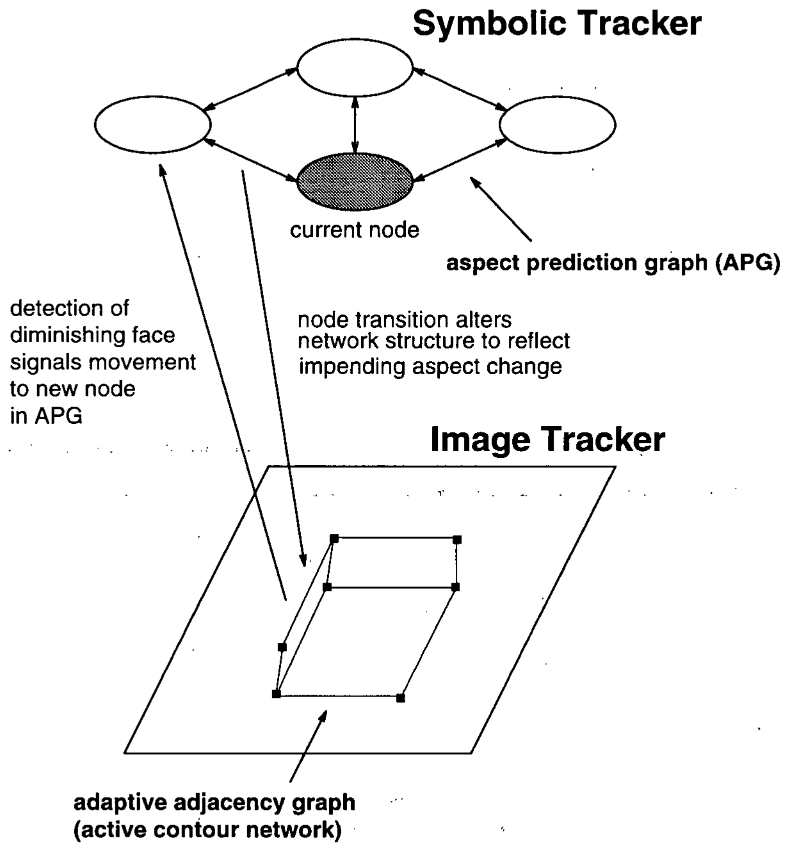
Symbolic Tracker
current node
aspect prediction graph (APG)
detection of diminishing face signals movement to new node in APG
node transition alters network structure to reflect impending aspect change
Image Tracker
adaptive adjacency graph (active contour network)

Figure 2.2: Dickinson's snake tracking system overview.

as the appearance or disappearance of object faces. A network of active contours called an *adaptive adjacency graph* [32] tracks and monitors aspects of the object from one frame to the next. An *aspect graph* is constructed whose nodes correspond to different views of the object, so that the tracking strategy simply involves moving from node to node in the aspect graph (see Figure 2.2). This method does not provide accurate poses of the object. However, it does qualitatively describe the motion of the object without specifying the object's exact geometry.

Also, the method cannot handle occlusion since the aspect graph only contains nodes where all object faces in a given viewpoint are present.

2.4 Parallel Tracking Systems

The majority of new motion tracking systems are parallel systems designed to satisfy real-time constraints. These systems range from implementations executing on a small network of parallel processors to others using large arrays or specialized parallel image processing hardware.

The system devised by Stephens [62] was tested on a single T800 transputer, but for real-time implementation the system shown in Figure 2.3 would be used. Images from the cameras are captured by two framestores, each of which is accessed by a T800 transputer, and all of the edge displacement measurements are performed by these processors so there is no need to communicate image data.

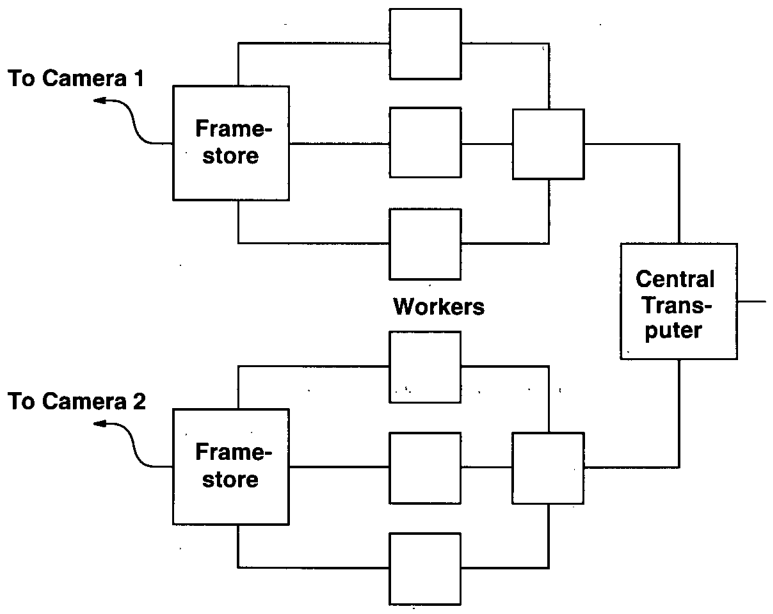

Figure 2.3: Hardware configuration of Stephens.

The worker processors are T800 transputers, and they carry out two tasks: projection of the model edge points into the image, and the formation of the Hough transform. The Hough transform is parallelized by dividing the edge points up among the workers, so the major communication overhead in the system is that of summing the Hough spaces from all workers. The workers are configured to minimize the communication overhead. A central transputer combines the Hough spaces from the different cameras, and interprets it to update the position of the object.

Verghese, Gale, and Dyer [73] constructed systems for motion tracking using two multiprocessor subsystems: an 8-stage Aspex Pipe and a Sequent Symmetry (see Figure 2.4). The two-level organization corresponds to the two kinds of processing requirements in computer vision. The low-level, iconic image processing is performed by Pipe and the high-level, symbolic processing is done using the Sequent. The system was able to track objects moving up to 15 *pixels/sec* with a time in the critical path of 266.7 *msec.*

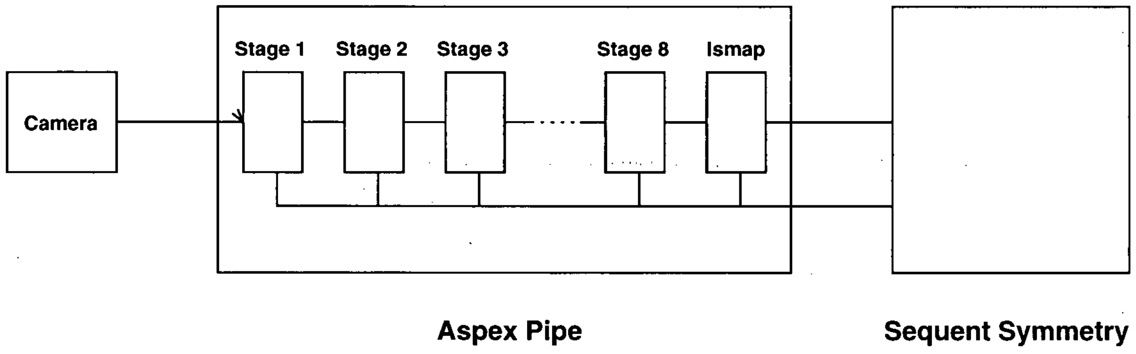

Figure 2.4: Verghese, Gale, and Dyer system configuration.

Pipe is a linearly-connected set of eight pipeline processing stages [34]. The computations in the stages are all performed in one Pipe cycle (1/60 sec) so that communication between stages is synchronous. Output from Pipe to the Sequent Symmetry is achieved through the Ismap processing stage. Ismap's image and histogram buffers are memory-mapped into the Sequent's address space so that these buffers can be directly read or written by either Pipe or the Sequent. The Sequent Symmetry is a tightly-coupled, shared-memory multiprocessor. Communication between Pipe and Sequent is asynchronous.

The main parallel programming techniques in used in these implementations are pipelining on the Pipe and semaphores between the Sequent and the Pipe. Synchronously, all stages receive input from their predecessors, perform their computation, and send output to their successors. Thus, during a single Pipe cycle, all stages send images forward. New images can be input at the rate of transformation completion (one cycle) rather than task completion (several cycles). Semaphores are designed so that the two subsystems can coordinate their computations; the Pipe can wait for the Sequent to complete some computation before proceeding to subsequent operations or vice versa.

The motion-calculating method of Verghese *et al.* is inherently more parallel than their motion-searching method. The motion-searching method cannot be completely parallelized because of data dependencies between the operations in the search-update loops, though multiple model pose hypotheses can be generated and tested in parallel. On the other hand, the motion-calculation operations of feature tracking and back-projection can be separated and performed concurrently. Independence of the image feature tracking process means it can be performed synchronously with the image acquisition and feature extraction operations.

Rygol *et al.* [58] have developed a parallel stereo tracking algorithm which is able to return the position and orientation of a moving object in 3D every 200 *ms*. They track an object by individually tracking a set of 3D line segments that comprise the object. The tracking algorithm thus employs *featural* parallelism, in that the features are tracked concurrently. The algorithm operates on the MARVIN system (Multiprocessor ARchitecture for VIsioN), which is composed of 25 T800 transputers wired as a regular, fully-connected mesh. In addition, MARVIN provides a hybrid processing scheme with the addition of frame-rate pipelined hardware for the computation of low-level image operations. The image data is dispersed among the processing array using 4 video buses and a specially developed transputer card which allows the simultaneous acquisition of up to 4 image streams.

The tracker is structured internally as a *processor farm*. The *master* of the farm is known as the *virtual tracker* and each worker in the farm is a *feature tracker*. The system exploits featural parallelism to perform real-time object tracking by decomposing the object into a set of visible component features which are tracked concurrently.

Chapter 3
Parallel Systems for Computer Vision

Computer vision tasks require an enormous amount of computation, especially when the data is in image form, demanding high-performance computers for practical, real-time applications. Parallelism appears to be the only economical way to achieve the level of performance required for many vision tasks. Researchers in human and machine vision share the belief that massive parallel processing characterizes low-level vision.

Low level (or early) vision is characterized by the local nature of its computations. There are few conceptual difficulties in designing parallel algorithms for these tasks and several existing systems do many of these tasks in parallel. Intermediate level vision involves segmentation - a reduction of the incoming visual information to a form that will be effective for the recognition step of high level vision. High level vision involves the *cognitive* use of knowledge. In general, parallelism is not immediately evident for the recognition step of high level vision. Figure 3.1 displays the various processing levels and the types of data associated with these computations.

3.1 Types of Parallelization

The objective of parallel processing is to perform a task more quickly and efficiently by distributing the work over a number of processors. Parallel processing is most effective when the work being carried out by each processor is equal to and independent of the workload of the other processors. Consequently, load balancing is a challenge involved in parallel systems which is absent in sequential systems. In order to efficiently utilize the added computing power, the processing load, programs, and data must be distributed among the processors in a balanced way in order to keep the resources active as much of the time as possible.

The two paradigms of programming parallel systems are:

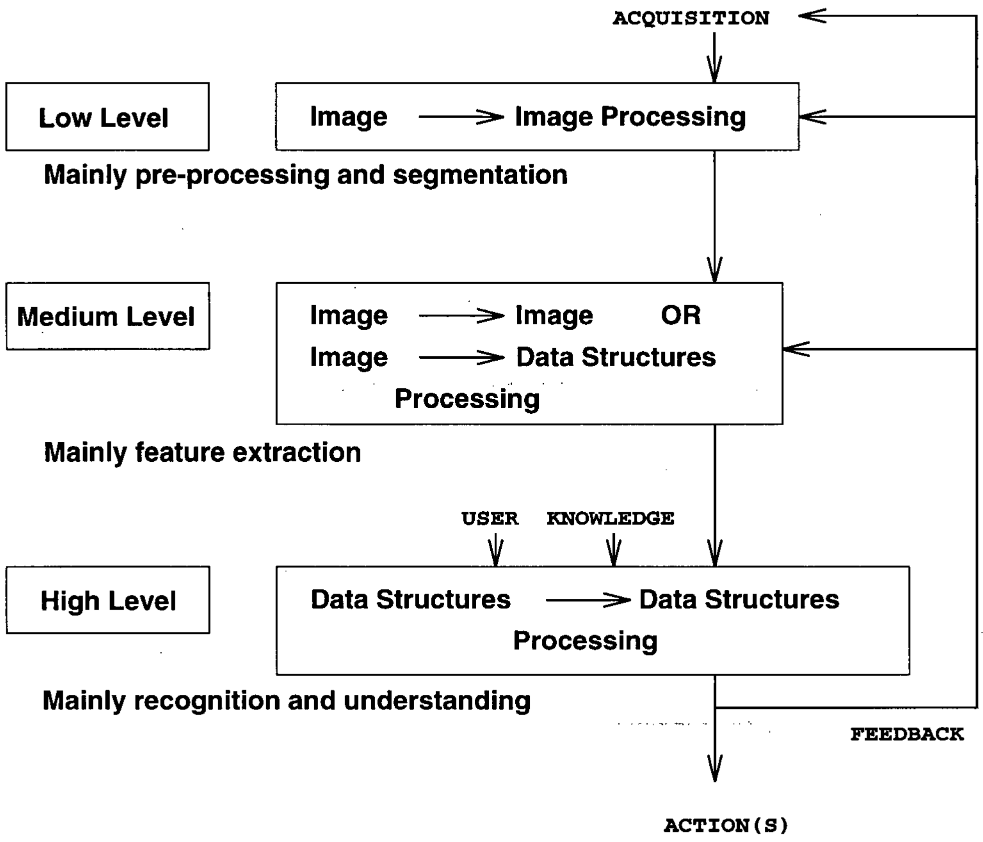

Figure 3.1: Processing levels and operation chaining for a typical vision based application.

(i) data parallelism and

(ii) task parallelism.

3.1.1 Data Parallelism

Data parallelism involves partitioning the data, in this case an image, such that each part or sub-image is processed by a different processor. Each processor accepts its portion of the image and executes the same program or series of tasks on the sub-image. Results are then combined from the separate processors once the computations have completed.

The image partitioning can be performed in different ways, exploiting the inherent geometrical parallelism present in digital images. A large two-dimensional array of processors can be

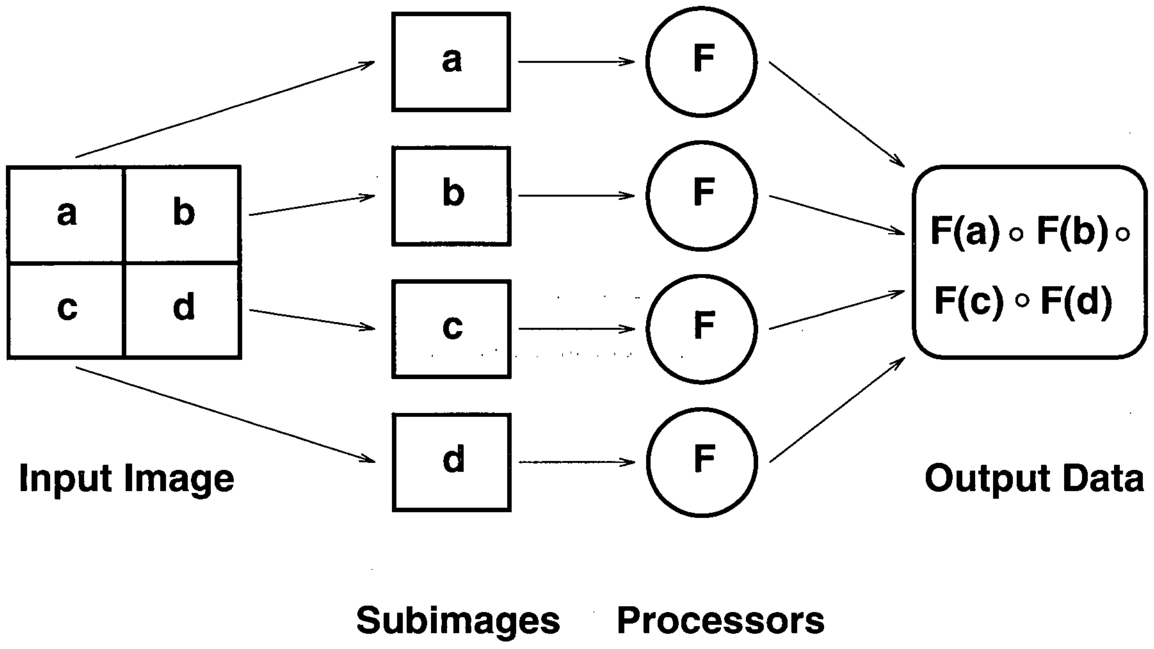

Figure 3.2: Data parallelism.

used to allot one processor per pixel. Of course, by current technological standards, this type of arrangement is not possible for large images, approximately exceeding 128x128 pixels. The image is customarily segmented into square blocks or strips, and each segment is assigned a specific processor. A problem encountered in this solution is the border effects that are created due to the image split and recreation in many processors. These effects can be diminished by slightly overlapping adjacent segments with the side-effects of requiring increased memory and communication. The amount of overlap required can be high for some operations. Hence data parallelism involves operators of the form:

$$Y \;=\; F(X) \;=\; F(x_1 \circ x_2 \circ x3 \circ x4) \;=\; F(x_1) \circ F(x2) \circ F(x_3) \circ F(x_4)$$

where X is an image which is divided into 4 parts, and $\circ$ denotes data composition. Figure 3.2 depicts such an operation.

Local neighbourhood operations can be handled very effectively with data parallelism. These operations take the form:

$$y_{ij} \;=\; \mathcal{F}(x_{i+r,j+s}) \;\; (r,s) \;\in\; A$$

where x_{ij}, y_{ij} are the input and output images respectively. $\mathcal{F}$ is an operator (linear or nonlinear) and A is its processing window. In this case a local processor must have access or communication to the neighbour data or processors.

Another method of partitioning the data involves decomposing an image into b bit planes, where b is the number or bits in the image pixel (usually b=1 or 8). Several image processing operators can be applied to each bit plane independently. The arithmetic that is performed on bit planes is called *distributed arithmetic.*

The communication overhead involved in data parallelism is not too substantial, since only small portions of the image are involved in any communication segment. Also, if the image is divided up into equal portions, the load will automatically be balanced across processors since each processor is performing the same task on the same amount of data.

3.1.2 Task Parallelism

Task parallelism involves the partitioning of various elements of the task or algorithm into sub-tasks such that each processor carries out a different sub-task. Each processor usually performs its computations on the entire data set rather than just a portion, and the data must commonly pass through a number of these tasks during execution.

When the number of processors grow, task parallelism is difficult to apply as there is a maximum number of sub-tasks in the algorithm. With data parallelism, however, the image can efficiently be divided into smaller subimages creating more data sets and allowing all of the processors to be utilized.

The communication becomes more involved with task parallelism. The data is usually transmitted through a number of processors before computations are complete. This data will likely occur in different forms as well, so that the communication is not homogeneous and must also take different forms. Moreover, there are synchronization and load balancing issues to consider since some processors will complete their work before others.

Two different types of task parallelism exist, and which type is utilized depends on the

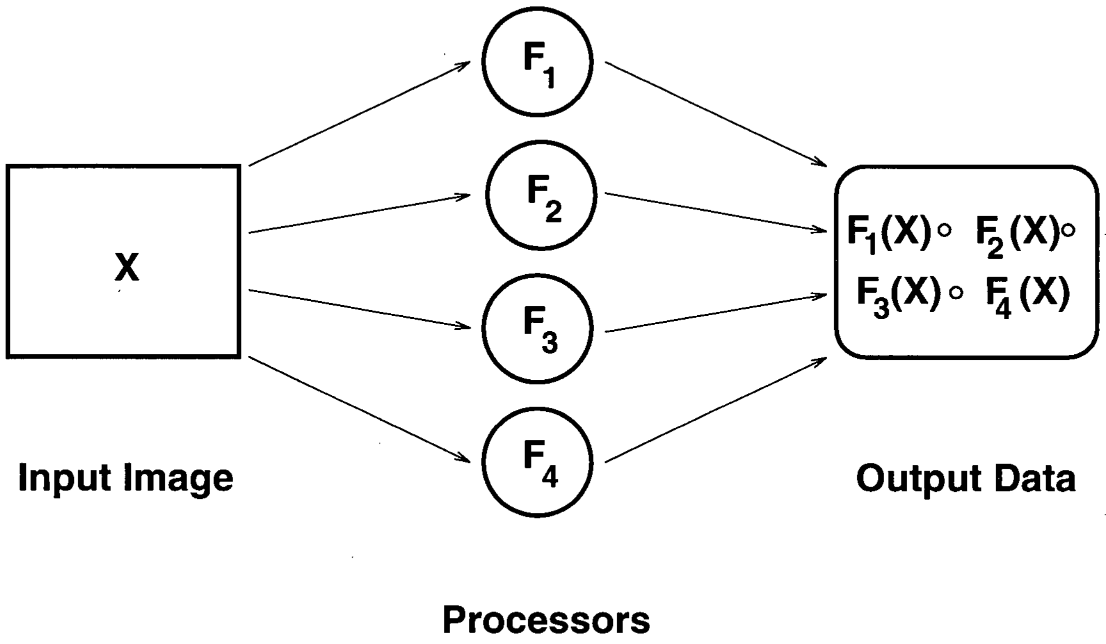

Figure 3.3: The parallel decomposition form of task parallelism.

algorithm and how data must be processed. They are: *pipelining* and *parallel decomposition*. Parallel decomposition involves operators of the form:

$$Y = F(X) = F_1(X)||F_2(X)||...||F_n(X)$$

where || denotes parallel execution. This can be used with algorithms which perform separate, independent functions on the same data, and these different tasks can be performed concurrently on separate processors. Figure 3.3 shows the parallel decomposition of an algorithm onto 4 processors.

Pipelining is more commonly used, and can be expressed by:

$$Y = F(X) = F_n(F_{n-1}(...F_2(F_1(X))...))$$

where X is the input image or image subregion, Y is the output image, F is an operator and F_i, $i = 1, ..., n$ is its cascade decomposition. This type of processing takes data and passes it along a sequence of processing stages. Each stage in the pipeline is assigned a different function, performing the same operation on every data set that flows through. Pipelined processing provides the possibility of processing the image in real-time since input data can be taken directly from the sensing device, no storage of intermediate results is required, and communication between stages is relatively uncomplicated. Images can continuously be passed to the system as

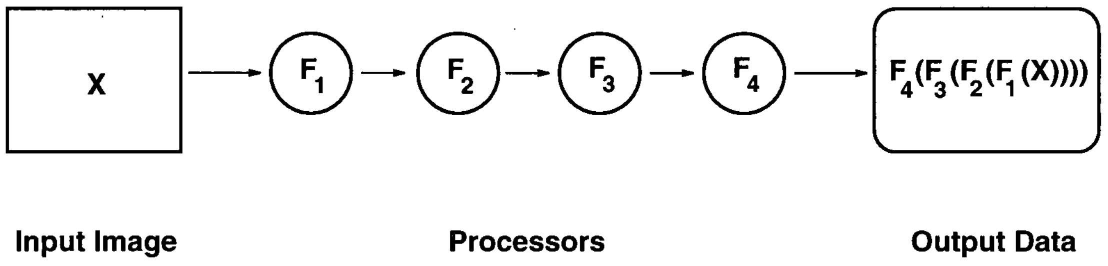

Figure 3.4: The pipeline processing form of task parallelism.

the first processor becomes idle, resulting in a number of images being processed in the pipeline at any given time. However, this form of processing has the drawback of lower flexibility, especially processors specifically developed for the pipelined processing of certain algorithms. Once set up for this particular algorithm, no changes can be made. Figure 3.4 portrays a pipeline with 4 processing stages.

Analysis of a pipelined implementation reveals:

- A pipelined implementation suffers from unbalanced dataflow. For example, the output of a convolution stage would result in a large dataflow to the next module while the amount of data to be exchanged in subsequent modules may be notably decreased. This exacerbates the problem of maintaining optimal use of the processors in the pipeline.
- A pipelined architecture optimizes *throughput* rather than minimizing *latency*. If a pipeline comprises N stages, each requiring an execution time of t (for an optimal pipeline), the temporal latency between receiving input data and producing output data is clearly Nt. This may not be acceptable for some systems which require rapid response to visual

stimuli. Some pipelined systems reduce latency by transferring partial results to the next processor. For example, the Datacube passes on each pixel after it is processed.

- Pipelining the modules requires the dismantling and rebuilding of the interconnected data structures as they are transferred from the local memory of one processor to the local memory of another.
- The length of a pipeline is limited to the maximum number of modules in the pipeline; this limits the number of processors available to efficiently implement the pipeline.

Most often, data parallelism seems to be better suited for implementing low-level vision algorithms on parallel systems. These algorithms use techniques which involve a great number of computations, usually in the form of elementary arithmetic and logic operators, performed on large pixel arrays of image data. It would be a laborious chore to split these computations into a number of different tasks. For high-level vision algorithms, for which data structures are manipulated containing more symbolic data and not large numeric arrays, the task parallelism approach is better suited.

Often a combination of the two paradigms is the best way to implement many vision algorithms. Initially the image can be divided into sub-images on which low-level computations can be performed, and then these results can be used to concurrently perform different tasks on separate processors. Alternatively, the processing can be pipelined, utilizing data parallelism in some stages where the data can be successfully decomposed.

The motion tracking system described in Chapter 4 follows a pipelining approach. The reasons for this decision are provided in Section 4.2.2.

3.2 Parallel Vision Algorithms

Parallel algorithms for a myriad of other operations have been researched, including visual recognition, smoothing, histogram generation, clustering, edge thinning, contour extraction, template matching, region growing, segmentation, stereo analysis, and object recognition [10].

The first stage of edge detection involving convolution by a mask is inherently simpler to implement on parallel hardware than the final stages of linking, grouping, and thinning operations. This is because the amount of processing required for any sub-image is predictable and equal to all of the others. It is an ideal situation to employ data parallelism.

The convolution operation is easily parallelized by dividing the image into equal-sized regions, possibly overlapping, with each processor receiving one partition. Templates are generated for the convolving function and applied by each processor to its corresponding sub-image. Many different implementations on various computer platforms have been published, ranging from hardware pipelines on systolic arrays to mesh-connected array processors.

The Canny edge detector [9] is implemented on MARVIN [8] using data parallelism with the image partitioned into a number of horizontal strips distributed over 16 transputers. A subsequent joining operation must then be employed to connect together edges which straddle a strip boundary; see Figure 3.5. A quadratic interpolation procedure is introduced to locate the precise edge position to sub-pixel accuracy.

A histogram of gray level content of an image provides a global description of an image. Conceptually, the image is divided into equal sub-images and histograms generated for each sub-image, which are then combined in a second pass to form the histogram for the entire image. Little *et al.* [38] use the Connection machine [29], where each processor determines whether its left neighbour is less than itself. Each processor for which this holds sends its cube address to the histogram table, resulting in a cumulative frequency distribution table which is easily converted to a histogram.

The Hough transform is a technique by which straight line segments are extracted by detecting global consistencies in the image data. It involves the mapping of candidate edge points in the image into a two-dimensional (ρ, θ) accumulator or Hough parameter space. The Hough space is defined in such a way that collinear sets of pixels in the image give rise to peaks in the Hough space.

The two main strategies to implement the Hough transform on a distributed system are

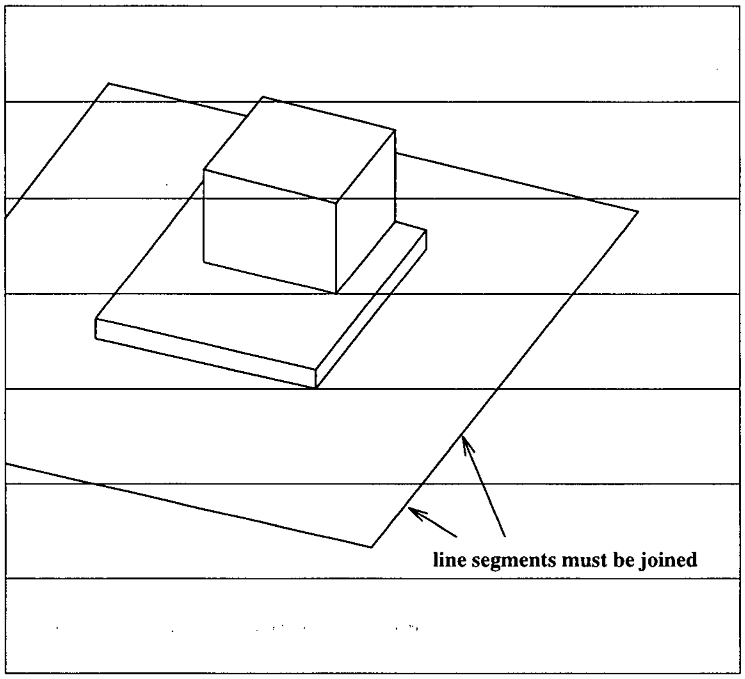

Figure 3.5: Edges which straddle strip boundaries.

either to distribute the image memory or to distribute the accumulator memory array across the processors. Images can be partitioned into horizontal strips or rectangular windows, but these methods could lead to memory contention when updating the accumulator. Alternatively, the Hough space could be split into a number of angle or line orientation intervals with each processor being responsible for a different range of θ. Memory contention at the input while reading the image is possible in this case. Finally, each of the P processors can be allocated a unique angle, evaluating the first P tasks in parallel and then processing the next P tasks.

3.3 Parallel Architectures for Vision

The past decade has witnessed the introduction of a wide variety of new computer architectures for parallel processing that complement and extend the major approaches to parallel computing developed beginning in the 1960s. The recent proliferation of parallel processing technologies has included new parallel hardware architectures (systolic and hypercube), interconnection technologies (multistage switching topologies), and programming paradigms (applicative programming). New architectures for parallel computer vision are constantly being proposed as hardware and theoretical advances are made.

A *parallel architecture* provides an explicit, high-level framework for the development of parallel programming solutions by providing multiple processors, whether simple or complex, that cooperate to solve problems through concurrent execution.

Various attempts have been made to classify parallel architectures. A central problem for specifying a taxonomy for modern parallel architectures is to satisfy the following set of imperatives [20]:

- exclude architectures incorporating only low-level parallel mechanisms that have become commonplace features of modern computers,
- base taxonomy on useful elements of instruction and data streams, and
- include pipelined vector processors and other architectures that intuitively seem to merit inclusion as parallel architectures, but which are difficult to accommodate.

Duncan [20] outlines a possible taxonomy of parallel architectures which is presented in Figure 3.6.

In this figure, the architectures are classified into three broad categories:

(i) Synchronous - concurrent operations are coordinated in lockstep through global clocks or central control units.

(ii) MIMD - concurrent operations are performed by multiple processors in a largely autonomous manner.

(iii) MIMD-based architectural paradigms - these architectures have MIMD characteristics but are also based on a distinctive organizing principle which is fundamental to its overall design.

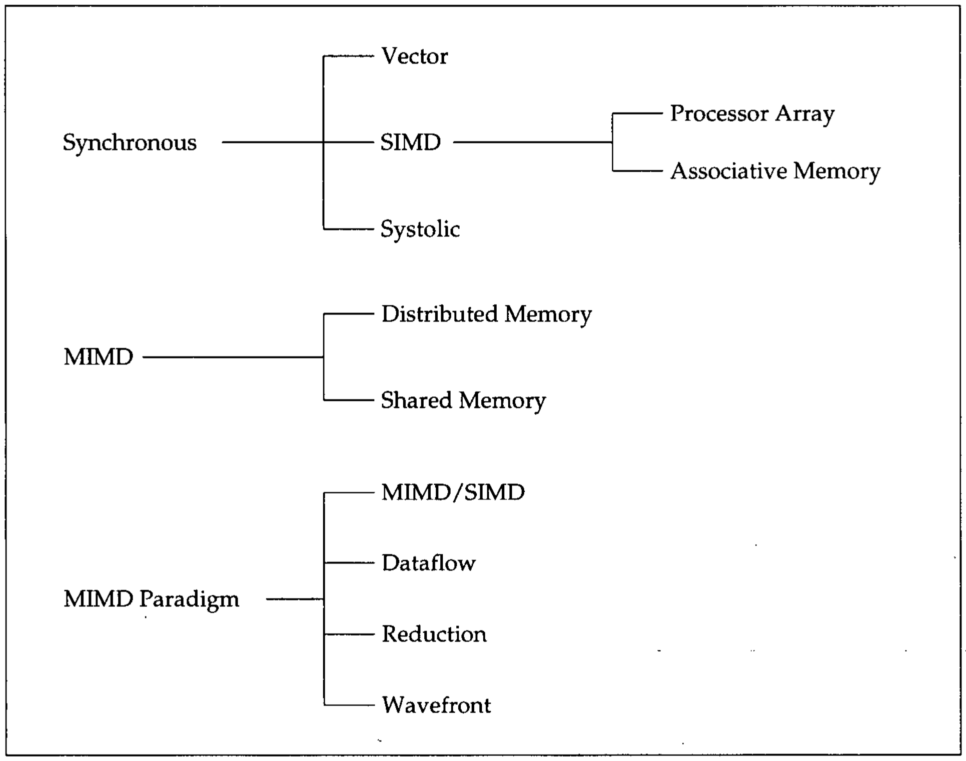

Figure 3.6: Parallel architecture taxonomy.

The items in the third category can be generally described as everything that doesn't fit into either of the first two classifications. These architectures are all grouped together to create a somewhat orderly taxonomy. Still, the two basic types of multiprocessor systems are *Single Instruction Multiple Data* (SIMD), and *Multiple Instruction Multiple Data* (MIMD) systems. Each type will now be examined in closer detail.

SIMD Architectures

A *SIMD* machine is composed of a set of identical synchronized processing elements which simultaneously perform the same task on different data. Vector and array processors are machines of this type; in general the interconnection network connects each processor only to its neighbours. Generalized information may be broadcast to all processors simultaneously but more processor specific information can take a long time to disseminate as it must be propagated from processor to processor in single steps. Data movement through large processor arrays can be a major problem. Generally, simple and efficient algorithms are used for data movement through large processor arrays. This may mean that several processors remain idle, but this is less complicated than trying to keep all of the processors busy by routing tasks and information around the large array.

These machines are well suited to localized, low-level vision tasks and global operations performed over the entire image. Such tasks include convolution, linear image transforms, image correlation, histogram generation, and component labeling. These architectures are not, however, well suited to data dependent operations such as edge linking and tracking.

Processor array structures for numerical SIMD execution have often been employed for large-scale scientific calculations, such as image processing and nuclear energy modeling. Various interconnection network schemes have been used to provide processor-to-processor or processor-to-memory communications, with mesh and crossbar approaches among the most popular.

MIMD Architectures

MIMD machines are those composed of multiple processors which are fully programmable and are able to execute different series of instructions. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner. Although sortware processes executing on MIMD architectures are synchronized by passing messages through an interconnection network or by accessing data in shared memory units, MIMD architectures are asynchronous computers, characterized by decentralized hardware control. MIMD machines may behave in a way similar to SIMD machines simply by arranging that each processor performs the same operation at the same time.

Data dependent vision tasks are better performed by MIMD architectures as processors may carry out different operations on various parts of the image. For most intermediate and high-level vision tasks though, the computations are not very regular and parallelism is not immediately evident. The main difficulty with these architectures is that of workload distribution among the processors. An example of this is the Hough transform; the operations to be performed are data dependent and are only executed in certain areas of the memory which can lead to work starvation of the processors. *Processor farming* is an effective method for load balancing when the amount of processing required for each data packet is unpredictable at the time of configuration. With this technique, tasks are allocated to processors as they become free.

A further classification of MIMD machines is into those which have memory distributed among processors and those which have a shared common memory. In shared memory computers, the processors all access the same memory through a central switching mechanism. Global image processing parallel algorithms particularly require memory of this type since this allows the access of each processor to all of the image pixels. Shared memory computers do not have some of the problems encountered by message-passing architectures, such as message sending latency as data is queued and forwarded by intermediate nodes. However, other problems, such as data access synchronization and cache coherency must be solved.

In the case of distributed memory architectures, each processor has its own private memory, and all communication and synchronization is completed via message passing between processors. Processing nodes are connected with a processor-to-processor interconnection network. The messages are passed using multiple communication channels, each one establishing a point-to-point processor communication path. These architectures have principally been constructed in an effort to provide a multiprocessor architecture that will "scale" (accommodate a significant increase in processors) and will satisfy the performance requirements of large scientific applications characterized by local data references.

Interconnection Networks

In all of the parallel computers, whether it is SIMD, MIMD, shared memory, or distributed memory, the processors need to be able to communicate with each other. Various interconnections between processors have been suggested, each with its unique properties. The interconnections used in commercial machines include mesh, pyramid, shuffle-design, hypercube, butterfly, and cube-connected cycles. Some of these will be briefly described here, though a good overview is given in [56]. The basic structure of some of these networks is depicted in Figure 3.7.

Ring topology architectures are simple networks most appropriate for a small number of processors executing algorithms not dominated by data communications. The communication diameter can be reduced by adding chordal connections.

A two-dimensional mesh, or lattice, topology has n^2 nodes, each connected to its four immediate neighbours. Wraparound connections at the edges or additional diagonal links are sometimes provided to reduce the communication diameter. The topological correspondence between meshes and matrix-oriented algorithms encourages mesh-based architecture research.

Tree topology architectures have been constructed to support divide-and-conquer algorithms for searching and sorting, image processing algorithms, and dataflow and reduction programming paradigms.

A Boolean n-cube or "hypercube" topology uses $N = 2^n$ processors arranged in an n-dimensional cube, where each node has $\log_2 N$ bidirectional links to adjacent nodes. Hypercube architecture has been strongly influenced by the desire to develop a scalable architecture that supports the performance requirements of 3D scientific applications.

Although distributed memory architectures possess an underlying physical topology, reconfigurable topology architectures provide programmable switches that allow users to select a logical topology matching application communication patterns. A single architecture can act as many special-purpose architectures that efficiently support the communications patterns of particular algorithms.

SPECIAL-PURPOSE SYSTEMS

Systolic architectures were proposed to solve the problems of special-purpose systems that must often balance intensive computations with demanding I/O bandwidths. Systolic arrays

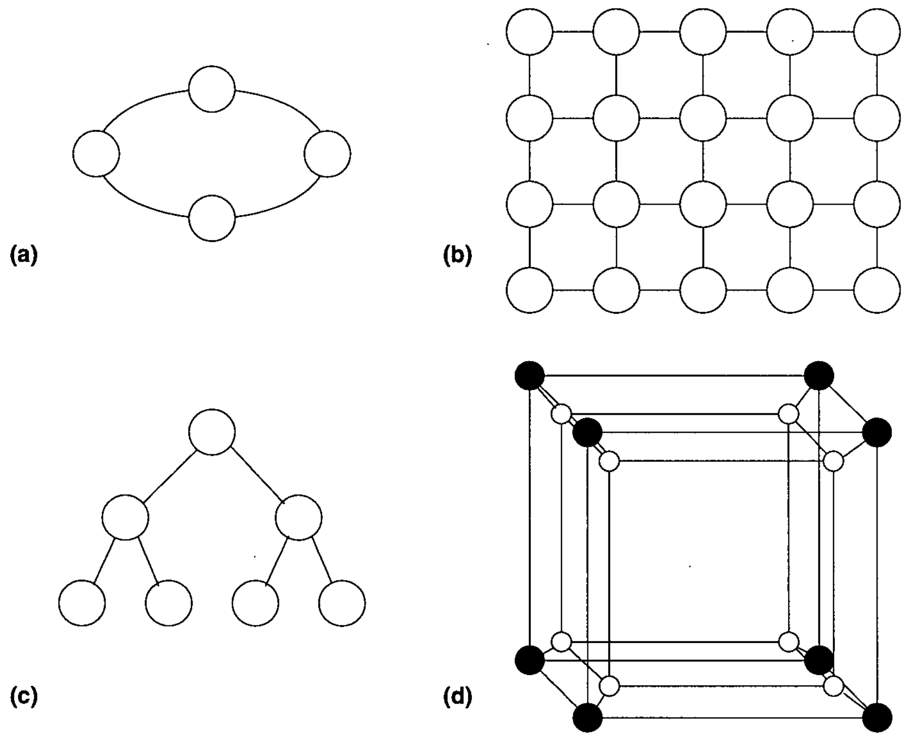

Figure 3.7: Interconnection networks: (a) ring, (b) mesh, (c) tree, (d) hypercube.

are pipelined multiprocessors in which data is pulsed in rhythmic fashion from memory through a network of processors before returning to memory. This pipelined data flow is synchronized, and consists of operands obtained from memory and partial results to be used by each processor. During each time interval these processors execute a short, invariant sequence of instructions.

Systolic arrays address the performance requirements of special-purpose systems by achieving significant parallel computation and by avoiding I/O and memory bandwidth bottlenecks. A high degree of parallelism is obtained by pipelining data through multiple processors, typically in two-dimensional fashion. Once a piece of data enters the systolic array, it is passed to any processor that needs it, without an intervening store to memory. Only processors at the topological boundaries of the array perform I/O to and from memory.

Processors with specific characteristics for the implementation of distributed memory MIMD systems have been developed. The INMOS transputer was the first microprocessor designed for this purpose [31]. The transputer is a small complete Von Neumann computer, and a number of transputers are usually assembled into a network or array. The transputer provides specific hardware for context switching between processes on the same processor, four serial links for connecting processors on a point-to-point basis, on-chip memory, and efficient direct memory access mechanisms for data transfer, in and out of the links.

MARVIN [8] is a transputer based general purpose vision engine capable of employing different types of parallelism within a single overall application. It consists of 25 T800 transputers organized as a regular, fully-connected mesh with 3 rows and 8 columns. One row of processors consists of special locally developed transputer cards (named TMAX) provided with data buses used to route large amounts of image data around the system and provide fast data transit. The MARVIN system has its own software infrastructure which allows the programmer to ignore the physical topology of the system and to work with a logical topology which may be chosen and changed dynamically.

The very different nature of various computer visions algorithms means that no one architecture is ideally suited to all stages. What appears to be required to implement a complete vision system is a front end processor performing global operations and employing data parallelism with a more sophisticated later stage to implement task parallelism for the data dependent higher level vision operations. An early attempt at just such an architecture is the Disputer [55] which employs a combination of a 256 processor SIMD and a 42 processor MIMD machine in order to optimize the strongest properties of both.

Little *et al.* [40] constructed a "Vision Engine" to support various levels of vision computation. The system consists of a Datacube MaxVideo-20 image processor for low-level vision tasks, and a MIMD multicomputer (16 transputers) for intermediate and high-level processing. The system is pipelined and the connections between the Datacube and the transputer subsystem pass through a crossbar so that the data can be sent to independent transputers for each module.

Other examples of work in this field include HBA [74], Kiwivision-2 [71], NETRA [11], the

Image Understanding Architecture [60], and the Warwick Pyramid Machine [53].

The Hierarchical Bus Architecture (HBA) [74] consists of 24 processors built around a digital video I/O bus. The Apply routine [27] is used which enables the vision programmer to write image-to-image transformations without regard to the details of parallelism, looping, or boundary conditions. NETRA [11] is a scalable architecture whose topology is a recursively defined tree-type structure. The leaf nodes consist of a cluster of processors which can operate in SIMD, MIMD, or systolic-like modes. The internal nodes are scheduling processors which handle task scheduling, load balancing, and global memory management. The Image Understanding Architecture (IUA) [60] was developed to embed three abstract levels of vision processing into an architecture. At the high level IUA is a MIMD parallel processor, the low level operates in pure SIMD mode, while the intermediate level operates in synchronous-MIMD or MIMD mode. Communication and data transfer between different levels is achieved using a shared memory.

3.4 Parallelization Issues

When parallelizing an existing system or constructing a new one, a lot of implementation decisions will depend upon the capabilities of the hardware that is available. For simplicity, we will assume that a collection of distributed memory MIMD processors are used.

The type of parallelism that should be used depends upon the form of the data and the types of tasks that have to be performed. Data parallelism should be used for low-level vision tasks involving local operations performed on large arrays, while task parallelism should be employed for higher-level tasks involving the manipulation of data structures. The data will usually be in the form of images initially, and thus will be well suited for data parallelism. A parallel implementation may also be based on a hybrid approach, using methods of both multiple-processor paradigms.

If data parallelism is selected, the next obvious task is to decide how to divide the data among the processors. Different methods that have been used are splitting the image into blocks, strips, individual pixels, or bit-planes. Other non-image partition methods may be used for such operations as the Hough transform, which partitions the Hough space. In any case, the type and extent of data partitioning depends upon the form of the data, the pending operations, and how many processors are available for use.

When combining the results after computation, adjustments must be made for any data partitioning effects which may have altered the data. This includes such things as convolution border effects, and line segment truncation during edge detection.

The first task in a task parallel conversion of a sequential algorithm should be the complete implementation of the algorithm, if possible, on a single processor. This should be relatively straightforward since only hardware dependent code should need to be altered. There are no communication concerns, no synchronization methods to establish, and no restructuring of the data required. Of course, with only one processor there are other concerns which must be addressed, such as memory, and hence it may not be possible to implement all features of the system, but the general functionality of the system should be attained.

Constructing this system initially has several advantages. Firstly, it will be very easy to tell quickly if it is possible for the hardware to support the system. If the algorithm cannot be executed on one processor there is not a great likelihood that it will run on a network of processors. Secondly, it results in a working system from which results can be displayed. Once parallelization of the algorithm begins, it will facilitate the subsequent division of the system onto separate processors with the ability to run the complete system while observing results to make sure that everything is functioning properly. Once the initial system on a single processor is constructed, work can then begin on splitting off tasks in a modular fashion onto separate processors, one at a time, checking the validity of incremental results.

The benefits of a parallel system may be lost if an effective data communication plan is not conceived. The topology of the interconnection network will depend on the types and amount of data which will be transferred between processors. Complex routing algorithms will be required in some cases to transfer data between non-adjacent processors limiting the amount of lag and intermediate storage.

Other parallelization issues will be discussed in the next chapter, when the motion tracking case study is presented.

Chapter 4
Motion Tracking in Parallel

This chapter will focus on the motion tracking system and how it was parallelized. The C40 processor will be described and will be evaluated based on many of its features for parallel systems. The implementation of the motion tracking system in parallel will then be discussed, including many of the decisions that were made and problems that were faced. Finally, a discussion will be presented which will describe how the system could be implemented using a data parallelism approach.

4.1 The C40 Processors

The need for parallel processing in today's computing world is quickly growing. Single processors alone cannot attain the performance required for real-time systems. Processors not designed for parallel processing are inadequate for the task as their device I/O capabilities cannot maintain reliable interprocess communication without degrading computing efficiency. Processors have been designed to meet some of the requirements of today's parallel processing applications.

One such processor is the Texas Instrument TMS320C40, which was the processor used to implement the motion tracking system described in this chapter. The TMS320C40 (henceforth referred to as the C40) is a floating-point processor well suited to the types of computations which must be performed in such a system. Some of the C40 device features include [66]:

- Six communication ports for high speed interprocessor communication.
- Six-channel DMA coprocessor for concurrent I/O and CPU operation, thereby maximizing sustained CPU performance by alleviating the CPU of burdensome I/O.
- High-performance CPU capable of 275 MOPS and 320 Mbytes/sec under ideal conditions.

- Two identical external data and address buses supporting shared memory systems and high data rate, single-cycle transfers.
- On-chip analysis module supporting efficient, state of the art parallel processing debug.
- On-chip program cache and dual-access/single cycle RAM for increased memory access performance.
- Separate internal program, data, and DMA coprocessor buses for support of massive concurrent I/O of program and data throughput, thereby maximizing sustained CPU performance.

A few of these features will now be examined in slightly more detail.

Communication Ports

The C40 processors are designed for multiprocessor use, and this is accomplished via the C40's six communication ports. These ports are capable of transferring 20 Mbytes of data each second between two processors in either direction. All data transfers are buffered, both input and output, so that no requests or data packets are lost. These ports greatly increase processor throughput since the communication need no longer take place through the external memory interfaces, keeping memory accesses and communication operations free from conflict.

DMA Coprocessor

The C40 possesses an on-chip DMA (direct memory access) coprocessor which is a very important feature for parallel performance and is one of the most important architectural differences between the C40 and its predecessors. The DMA coprocessor allows the CPU to focus all of its power on computations without also having to transfer data within the processor's memory map. The DMA handles the I/O tasks, and is able to read and write to any location in the memory map through its six channels. This is especially beneficial when interfacing with slow external memories and peripherals. Dedicated DMA address and data buses minimize conflicts between the CPU and DMA coprocessor.

Memory Organization

The C40 has a 16-Gbyte continuous address space for reaching program and data memory, as well as registers affecting timers, communication ports, and DMA channels. The system designer can choose to locate programs and data at any desired location, resulting in a great deal of flexibility in designing system memory use. Tables, coefficients, program code, and data can be stored in either RAM or ROM. Each RAM and ROM block is also capable of supporting two memory accesses in a single cycle to notably increase performance, especially when data resides in on-chip RAM.

The on-chip RAM takes the form of a 512-byte instruction cache. This feature is provided to store often-repeated sections of code or chunks of data, thus greatly reducing the number of needed off-chip memory accesses. Code can thus be stored off-chip in slower, lower-cost memories. Utilizing the cache also frees the external buses for use by the DMA, external memory fetches, or other devices in the system.

Internal Buses

A large portion of the TMS320C40's high performance is due to internal busing and parallelism. Separate buses allow for parallel program fetches, data accesses (read and write), and DMA operations. These buses connect all of the physical spaces (on-chip memory, off-chip memory, and on-chip peripherals) supported by the C40. The CPU can, for example, access two data values in one RAM block and perform an external program fetch in parallel with the DMA coprocessor loading another RAM block, all within a single cycle. As for external buses, two identical external interfaces are present: the global memory interface and the local memory interface. Both buses can be used to address external program and data memory or I/O space. These external buses have a high port data-transfer rate of 100 Mbytes/sec.

Pipelined Architecture

As is the case with most other modern processors, the internal operations of the C40 are pipelined to minimize the instruction cycle time. The major units of the C40 pipeline structure are as follows:

- instruction fetching,
- instruction decoding and address generation,
- operand reads,
- instruction execution, and
- DMA transfer.

These functional units operate in parallel to aid in achieving the high throughput that is evident.

Altogether, the C40 CPU is capable of eleven operations per cycle throughput including computations, data accesses, address register modifications, and counter updates. This results in a high-degree of parallelism and sustained CPU performance.

External Interfaces

The interface design of the C40 can be used to implement a wide variety of system configurations. The two external buses and DMA capability provide a flexible parallel 32-bit interface to various devices; the communication ports provide an interface to other C40s; and the interrupt interface, communication ports, and general-purpose digital I/O provide communications with different peripherals. Each interface is independent of the others, and different operations may be performed simultaneously on each interface.

The global and local buses implement the primary memory-mapped interfaces to the device. These interfaces allow external devices such as DMA controllers and other microprocessors to share resources with one or more C40s through a common bus. Devices that can be interfaced to the C40 include memory, DMA devices, and numerous parallel and serial peripherals and

I/O devices. Figure 4.1 illustrates a typical configuration of a C40 system with different types of external devices and the interfaces to which they are connected.

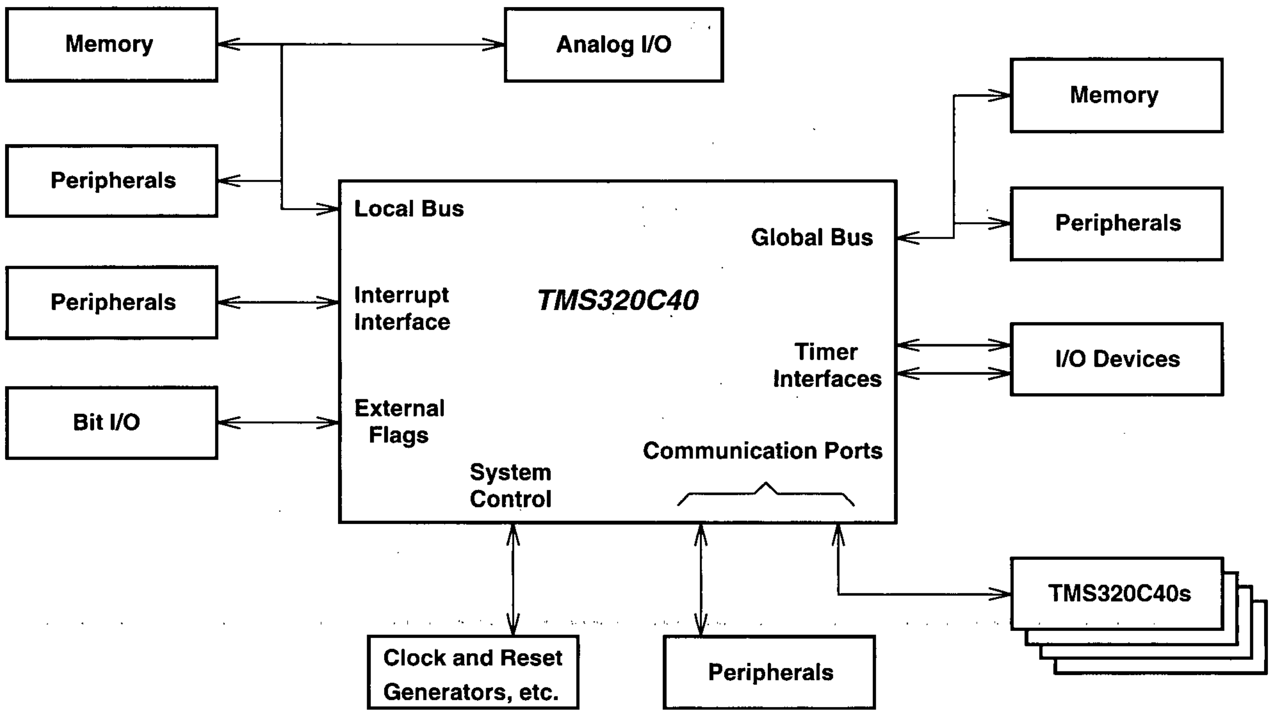

Figure 4.1: Possible system configurations.

DEBUGGER

The C40 C source debugger is a software interface that aids in developing, testing, and refining C40 C programs and assembly language programs. The data displays provided by the debugger allow the values of variables, arrays, data structures, and pointers to be changed, while this information is continuously updated on the screen. The C40 debugger allows the usual contol over program execution with features such as breakpoints, conditional execution, and single-stepping. Individual breakpoints can be set on any or all processing nodes in the system. Also, a memory map can be defined which identifies portions of the target memory that the debugger can access.

Supplementing the standard debugger features is an analysis module which allows the user to monitor the operations of the target system. The analysis module captures C40 bus cycle information in real time and reacts to this information through actions such as hardware breakpoints and event counting. Such features supply the ability to stop the processor and track the path that the program took before reaching the breakpoint or event. Types of information that can be monitored include bus accesses, CPU clock cycles, branches, interrupts, and instruction fetches.

In addition to the basic debugging environment, a profiling environment is available. The profiling environment provides a method for collecting execution statistics about specific areas in the code. This can provide immediate feedback on the system's performance. The types of statistics that can be gathered include:

- the number of times each area was entered during the profiling session, and
- total execution time of an area, including or excluding the execution time of any subroutines called from within the area.

These statistics can aid in identifying bottlenecks which are hindering the performance of the application.

See Section 4.3 for more discussion about the debugger.

4.2 System Description

This section will describe the underlying motion tracking algorithm and the major modifications to the system which were required to construct the parallel implementation. Obviously, in any parallel conversion, one of the major changes will be the addition of interprocessor communication, and the form that this communication took in the motion tracking system will be described. Also the division of labour onto the C40 nodes and the resulting tasks will be discussed.

4.2.1 Single Node Implementation

The first task in the pipelined parallel conversion of a sequential algorithm should be the complete implementation of the algorithm, if possible, on a single processor. This step was previously discussed in Section 3.4.

The motion tracking system was implemented on a single node with a couple of modifications. Since the image capture and display capabilities reside on separate processors, it was necessary to allow the system to acquire and process images originating from an input file. Also a loop iteration scheme was not followed initially; only one image would be processed before termination. This enabled the meticulous examination of results after one iteration from which conclusions could be drawn regarding their validity. Once the results obtained from this version were deemed satisfactory, a few iterations were executed in order to totally initialize the system, fill up the pipeline, and check all communication passages including the message transfer from the final node back to the image grabbing node.

At the time of implementation, the system was running on a single node in about 2 seconds per loop. Once the system was running acceptably on one node, the chore of dividing up the work between processors was begun.

4.2.2 Pipeline Structure

As was previously mentioned, it was decided to follow a pipeline approach in parallelizing the motion tracking system. The reasons for reaching this decision are as follows:

- A pipelined implementation is analogous to the logical flow of data and information present in the system.

- The motion tracking system can be readily and sensibly split up into 5 or 6 separate subtasks which can be executed in sequence. This corresponds to the number of available C40 processors.

- A pipelined version of the system can be easily implemented with few changes needed to the original algorithm. Compared to the data parallelism paradigm, this method requires much less effort.
- No edge effect problems are present when combining results as in the data parallelism scheme.
- A simple communication strategy is possible; synchronous communication occurs when adjacent nodes have both completed processing on their current data structures.
- A pipelined approach is much more reasonable for the types of operations that will be performed. Data parallel approaches cannot intuitively decompose high-level operations without creating severely unbalanced processing loads and major communication overhead.
- Computation time following this approach is relatively similar for all tasks; any one processor will not require a disproportionate amount of processor attention. This results in efficient use of resources and reasonably balanced processing loads.
- This method makes it easy to gauge progress; programming for each sub-task can be completed and the results displayed without a lot of extra work. The data parallelism approach would require combining results at the end of every stage to determine the validity of the data.

The decision to follow a pipeline approach is not without problems, however. Some of the disadvantages associated with pipelining are as follows:

- Unbalanced dataflow is present. The early stages transfer complete images while nodes later in the pipeline are communicating much smaller data structures.
- Temporal latency becomes a problem. There is a lag of about 4 frames between receiving input data and producing output data. This is not propitious for a lot of real-time applications. On the other hand, it does optimize throughput since new images are constantly

fed into the system, and results are continually displayed. This helps to minimize the loss of information.

- Data structures must be dismantled and reconstructed for communication between processors.

- It is not clear how the system can be expanded if more processors become available. A pipeline cannot be "lengthened", and adding processors to aid in the completion of subtasks may hinder the efficiency of the algorithm, although certain processing modules are amenable to spatial decomposition.

- The linking and matching procedures have variable execution times since they depend on the complexity of the scene, amount of spatial disparity, and the total number of edges detected in the image. This could lead to unbalanced processor loads.

It is important to note here that the temporal lag problem does not mean that the system is tracking on "old", outdated information. The system is constantly fed new data, but the results are not available until the pipeline computations have completed.

The pipeline structure implemented takes as input a stream of images, and each iteration of loop produces a single element in the output stream of model poses. The following sections will now address the construction of the sub-tasks on the individual nodes. Figure 4.2 displays the ordering of sub-tasks in the pipeline.

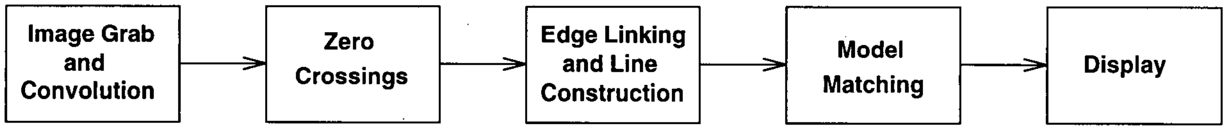
Image Grab and Convolution
Zero Crossings
Edge Linking and Line Construction
Model Matching
Display

Figure 4.2: The task ordering in the pipeline structure.

Processor 1 - Image Capture and Convolution

The initial processor in the pipelined structure is responsible for receiving the input images and for the convolution of these images with a convolution kernel. The images are received from the frame grabber in a compressed format; four pixel values are packed into each word. Hence images are entered into a large table, row by row, leaving space for the subsequent expansion of the data. The images are then stored as sequential arrays with an indexed lookup array which stores the address of the first pixel of each row. This provides an efficient method for image pixel access. The image size used was 512x460 pixels. This image size is a lot larger than the standard 256x256 used in most previous applications.

The input image needed to be modified because of the interlaced nature of the camera data. Since two scans are performed for the capture of each image, the first for the odd rows and the second for the even rows, it is important to only use every second row of the input image. This is especially crucial for moving objects since the object could be in a different position for the second set of scan lines, resulting in a jagged and incoherent image. Hence an image compression or vertical scaling operation was performed by discarding every second horizontal row. This has effects on later stages, as the image needs to be expanded again before the results are displayed. The edge endpoint locations are stored properly with the use of a scaling variable which holds the amount of vertical scaling applied to the image.

The convolution routines for this system are entirely new as the convolution was previously performed on special hardware. For speed, assembly language convolution routines were formulated so that the on-board memory of the C40s are utilized. For each row of the image, pixel values and data from the kernel are simultaneously stored in the on-chip memory of the C40 so that convolution operations can be performed without expensive memory accesses and indexing.

The method carried out is a Laplacian of Gaussian convolution operation, which applies a Laplacian operator smoothed with a Gaussian mask to the image. Hence the Laplacian of Gaussian operator may be shown to equal

$$log = \delta^2 G(x, y) = (1/\pi\sigma^4)(r^2/2\sigma^2 - 1)exp(-r^2/2\sigma^2)$$

where $r^2 = x^2 + y^2$. New 5x5 and 7x7 convolution kernels were designed in this manner.

The convolution routines were created with the ability to specify an image starting address, a row width, and an image width and height. This allows for the flexibility of convolving only a subimage if the object of interest has been tracked to a certain location. In this case, an output row width can also be specified so that the convolution result can be packed into a smaller output buffer.

Processor 2 - Zero Crossings

The zero crossings processor takes as input a convolved image, and from this delineates the boundaries between positive and negative intensities (zero crossings of the first derivative). This task is fairly straightforward and required few changes for use in a parallel system.

Zero crossings in the convolved image occur in places where there is a sign change between a pixel and its right neighbour or lower neighbour. The value of the zero-crossing pixel is then taken to be the local gradient - the difference between the two values, or the maximum difference if there is a sign change in both directions. All other pixels are marked with a zero. The result is an array of pixel values, non-zero every place where a zero-crossing occured in the convolved image. There may be gaps in some of these zero crossing segments, as any pixels below a certain gradient threshold level are dropped. This will help to reduce the number of less prominent edges. The gradient values themselves can be used in the next stage to remove shorter edges from the feature configurations.

Processor 3 - Edge Construction

The edge construction phase is composed of two tasks: the linking of edge points and line segment formation. These operations are performed consecutively on a single processor since together the amount of time required to complete these tasks is comparable to the work performed on the other processors.

The processor receives as input an image consisting of non-zero values everywhere a zero crossing occurred in the convolved image. These points are linked together into structures by tracking zero crossing segments and storing the points in a linked list. A Canny hysteresis

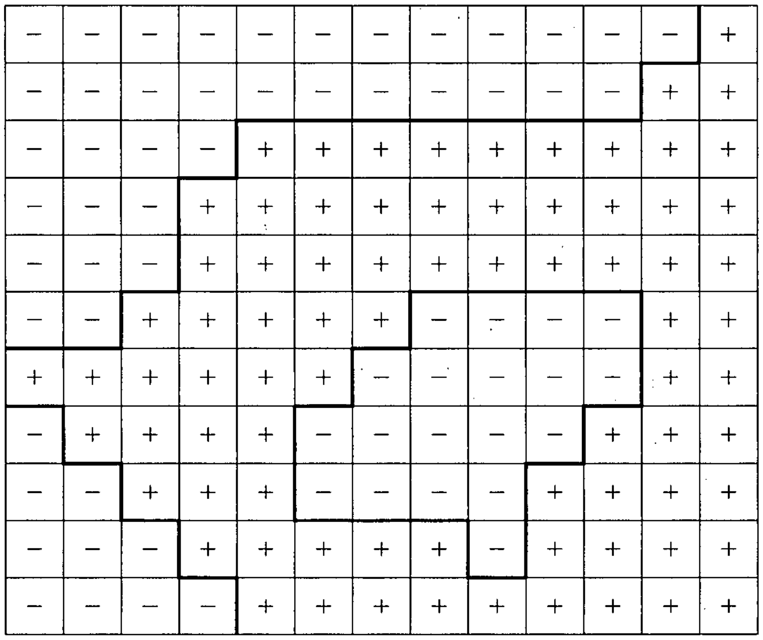

Figure 4.3: Example of zero crossing boundary creation.

thresholding technique is used [39]. The points are traversed looking for non-zero values, and if the gradient is above a certain selected high threshold value, a new edge is started. Since the initial pixel is likely in the middle of some edge, this edge is tracked in both directions from the first pixel.

As it is added to an edge, the pixel is removed from the image buffer by being zeroed out so that it is not included in another edge segment. Tracking on the edge continues until the gradient drops below a low threshold value. Further stipulations require that the endpoint gradients be above the high threshold level, and that the lines have a minimum length. The resulting edges include only sharp intensity discontinuities and shorter edges which would not aid in the tracking have been removed.

A change was made in the way in which these structures are created and stored. Once a complete edge segment has been tracked the number of points it contains is known, so instead of allocating an object for each point structure, an array of point values can be allocated. This implementation is likely more efficient than using pointers and is likely to result in better cache coherence. Each element contains the x, y coordinates of the point as well as the gradient value across the zero crossing. A linked list of edge segments is formed to hold the point arrays for each segment, and to enable the addition of subsequent edges.

Since a lot of memory is manipulated at this stage, an efficient method is required to handle memory allocation. This allocation is performed by a number of utility routines which grab small chunks of memory from a large storage pool. Memory can be taken as required, and rather than allowing each structure to be freed individually, it is only possible to free an entire pool of storage at once. This is not a limitation since each image will initially require an empty workspace. This method is a much more efficient means of allocating small objects, as there is no overhead for remembering all the information needed to free each object. Also, since objects are allocated in blocks, small chunks can be taken from these blocks without any allocation overhead by simply manipulating pointers in this block. Finally, it is possible to free all objects associated with some particular pool without having to link them all together and traverse a list to free them.

Once all of the linked edge structures have been constructed, the edge structures are used to construct straight lines. This results in a large reduction of data, since only the endpoints of the straight lines need to be stored and not every pixel, and this will also convert the data into a form which is consistent with the form of the model parameters so that matching can be readily performed.

A recursive subdivision procedure is used to find the best straight line segmentation for the input edges. These straight lines correspond to edges detected in the image. The procedure begins by subdividing the list of points into two segments at the point where the deviation from the line joining the first and last points is the greatest. The routine is then called recursively on each segment. The routine returns when the significance of a segment (based on a length to deviation ratio) falls below a predetermined lower limit, and combines smaller segments of lower significance into longer segments. Hence a number of straight lines can be created from one linked pixel list.

In addition to this line construction method, a least-squares line fitting capability was also added to the system, and an option file allowed the user to select which type of line fitting would be performed. A further option introduced to the system allows the tracking loop to halt at this stage and simply display the edges detected in each image frame.

Processor 4 - Matching

The next processor in the pipeline controls the matching operations. The matching procedure is used to identify correspondences between image edges and model edges projected from the current model position. The process begins by storing the current position of the model so that these parameters can be recovered if the matching process fails. Next, the edges that were detected in the image are partitioned into different bins depending on their orientation and the location of the midpoint. All image edges are processed and indexed in this way.

For each projected model segment, its current orientation and location bins are searched for potentially matching image segments. Match records are created if the candidate edges are within the appropriate bounds for the model edge. To aid in the classification of matches, probabilities that the image edge matches some model edge are estimated, based on the locations of the midpoints, the difference in orientations, and the difference in line lengths. A pair of edges are given higher evaluations if they have connecting endpoints when their corresponding model edges are also connected. These values are then used to rank the matches, and the "best" candidate matches are used in a least-squares solution. This best-first search procedure minimizes the time spent searching for correct sets of matches, and also eliminates the need to treat outliers. The most reliable initial matches are used to reduce the parameter variance on further iterations, minimizing the amount of search required for matching more ambiguous features. Initially, only enough feature matches are selected to constrain the degrees of freedom of the object.

A least-squares solution is performed, and the residual is examined to evaluate the consistency of matches. If the match is rejected, then at least one of the segments in the match set must be in error. The probabilities for all of these edge segments are reduced, which will cause new matches to be considered in the next search. Successful matching enables the viewpoint and model parameters to be updated, and subsequent iterations involving more model features are performed from this new viewpoint following Newton-Raphson convergence. A couple of iterations of this convergence are usually sufficient to accurately determine the object's position. Details of the least-squares solution and Newton-Raphson convergence are given by Lowe [41, 43, 44].

Processor 5 - Control and Display

The final processor in the pipeline is responsible for displaying the tracking results, and for the overall coordination of the system. Once the new position of the model has been determined, it is projected onto the image frame so that it can be displayed. The model is drawn in red if the previous match failed, and only visible edges from the current viewpoint are displayed.

Splitting the matching and display code onto separate processing nodes requires a lot of data to be passed between these nodes: model projection information (possibly one set for each convergence sequence), the edges that were detected in the image, and the edges which were matched in the matching process. To make this data transfer more efficient, a data structure was constructed consisting of a number of line entries, containing information about endpoints and line type. Thus model, match, and detected edge information could all be transferred via the same structure.

Hence the display node receives as input a list of lines which correspond to the new pose of the model and outputs a set of lines depicting the projected model edges at the object's new position. Optionally, this node is capable of displaying the edges detected in the image and the lines used for matching. Other information received from the matching node includes the portion of the original image which was processed and has been passed through the entire pipeline. This image is displayed in the background so that the validity of the tracking can be examined as the system is running. Once all of the pertinent information has been output, the new position of the model must be transferred back to the beginning of the pipeline to aid in speeding the computations. This is accomplished by forming a bounding box around the projected model position, and passing the coordinates of this box to the grab node. The dimensions of the box are increased by a few pixels in each direction to allow for possible model movement between iterations. All future computations will then be confined to this processing "window".

The display node also receives asynchronous status messages from all of the previous nodes. These messages can contain information such as the number of bytes transferred between nodes and timing statistics. This information was useful in constructing and debugging the system since communication with the user in the form of output to the host was unreliable.

4.2.3 Interprocessor Communication

Once a pipeline approach was decided upon, the most important implementation decisions that remained dealt with interprocessor communication. How should data be passed between nodes? What data should be communicated? What form should the communication take? How should communication between all of the nodes be coordinated?

Enter the worm-hole router. The TMS320C40 router is a software controlled message passing system. It provides the means to pass messages of arbitrary size between processors not directly connected. These communications follow a *worm-hole* approach. This means that the messages are passed through intermediate nodes to reach their destination, but unlike some routers (such as the *store-and-forward router*), these intermediate nodes do not store the message in local memory. Instead it is transferred immediately from an input port to the corresponding output port. This makes communication more efficient because it is faster and doesn't require any additional memory in the intermediate nodes.

Figure 4.4 shows the interprocessor communication that occurs in the motion tracking system. It is important to realize that if two nodes are adjacent in the system pipeline, it does not necessarily mean that they are connected physically. In fact, Figure 1.1 shows that the zero crossing and links nodes, as well as the match and display nodes, are *not* actually physically linked. Forming a complete, connected network would result in a chaotic mass of wires and would complicate later additions to the network. The router's efficiency is evident since communications proceed as if all of the nodes do have physical interconnections.

Since the router is software based, routing decisions and high level control functions are executed by the CPU. These functions are decentralized so that each intermediate node makes its own routing decisions. The C40 multichannel DMA is then responsible for handling the actual message transfer. A connectivity matrix is specified at the outset to select which communication ports on all of the nodes will be utilized.

The data transfer between processors in the pipeline is synchronous and uses a "handshaking" style. The destination node signals that it is finished its computations and ready to start communicating by sending a single byte to the source node. After receiving this status byte, the source node proceeds to transfer its results. If a receive request is issued before the source node is ready, the request is queued and the receiving node will wait, blocked from doing further work. Similarly, if the send request is issued before the destination node is ready, the message will be sent and queued in local memory at its destination until the appropriate receive request is issued.

The synchronous method was required to control the processing speed of certain nodes. Without this check, earlier nodes in the pipeline may execute too quickly, filling up the message buffers on later nodes, and ultimately crashing the system. Also, asynchronous methods could not perform efficiently when passing large image messages.

The type of data that is sent depends on the stage of the pipeline where the transfer is occurring. Data passed from the convolution node to the zero crossing node and then to the edge linking node is in image form (a large two-dimensional array of values) and is passed in a sequential array format since the receiving node knows the dimensions of the image. Data passed from the edge linking node to the matching node and then on to the display node is greatly transformed. It now is a large structure of line segment entries, and must be in

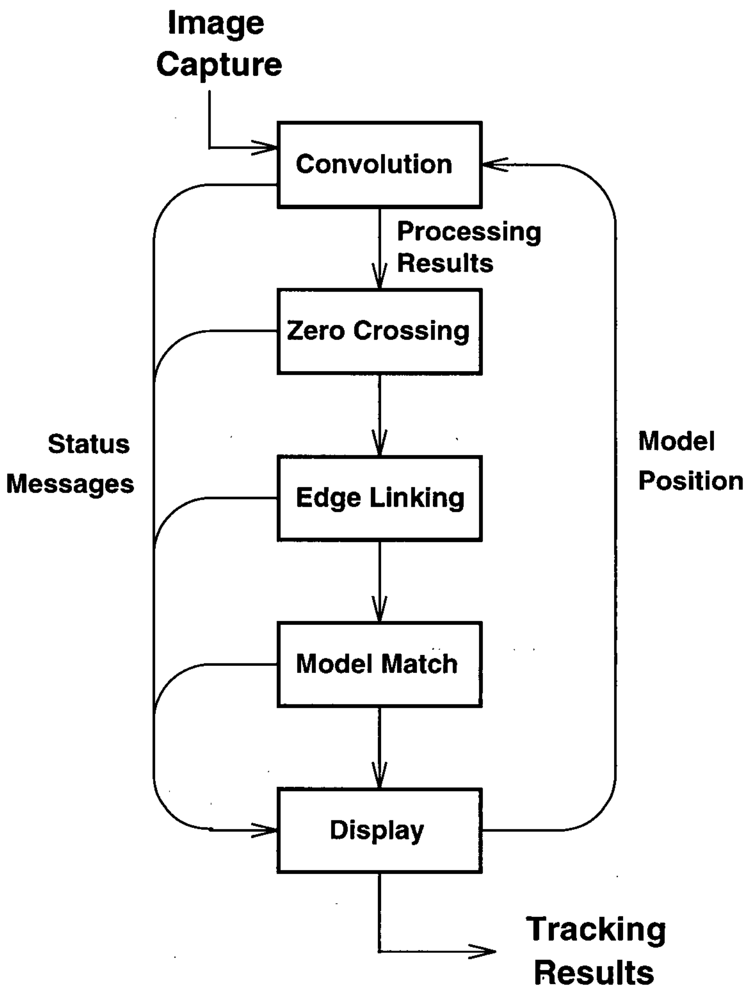

Figure 4.4: Interprocessor communication.

array form (each data structure entry a single element in the array) with no pointers so that the information is sequential and will be valid after transfer. (Obviously, if only pointers are passed, there is no correspondence to what will appear at the corresponding address in the destination node's memory!) The receiving node can determine how many line entries occur in the message since it knows the total length of the message and the size of each line segment record.

In addition to the previously mentioned data communication through the pipeline, a second piece of information is passed at the same time. The original image (or portion of) is transferred by way of a second set of send/receive pairs down the pipeline. A simpler and more effective method may have been to pass the image directly from the initial node to the display node. Reasons why this was not done are given in Section 4.3.

The final type of communication present in the motion tracking system is the asynchronous transfer of status messages from all nodes to the display node. This message passing is asynchronous since the sending nodes issue a single send request and then continue processing without waiting for the message to be received. At the end of each processing loop, the display node will then clean out its status message queue, displaying any necessary data, and then prepare to receive it's next set of match data. The originator of each status message is known since the source node id is attached to the message header.

4.3 Problems and Roadblocks

This section will address some of the problems which were encountered when parallelizing this system, and will focus on a number of alternative actions taken to overcome these difficulties. Various roadblocks encountered included communication problems, programming bugs, and hardware difficulties. Problems were even created by an optimizing bug existing on the C40 compiler which caused the compiler to incorrectly interpret a double-assignment with post-increment instruction.

When the system was initially being constructed, an easy way to get information about the validity of data that was being computed was to output intermediate results or status messages to the display of the host. This creates problems in this sort of parallel system since print statements will need to communicate information through the network. The problem is not as evident when the statements are issued by the node which is directly connected to the host, but when intermediate nodes are required to transfer the information to the host problems can arise. Not only are communication ports being occupied, but conflicts can occur with the regular data communications. This often caused the system to terminate prematurely.

As previously mentioned, it would have been desirable to send the captured image directly from the grab node to the display node, and thus avoid many extra transfers down the pipeline. Since the image would arrive at the display node before the processed data made its way through each subtask, a scheme was devised whereby each image would be given a unique identifier, and a number of images would be stored at the display node until the matching process is complete. This would greatly accelerate the communication between all nodes in the pipeline. However, much more memory would be required on the output node. The synchronization of communications turned out to be the demise of this plan. The image could not be passed synchronously without seriously hindering the ongoing synchronous pipeline communication. Asynchronous communication was attempted, but the size of this communication caused it to be too slow.

A great deal of effort was expended in attempts to speed up the tracking loop, and some of the improvements that were made can be found in Section 4.4. When an early version of the edge linking procedure was taking far longer than expected, it was discovered that the memory freeing routines were not very efficient. At the time, memory for the edge structures was being allocated as needed and linked to the end of the existing list. When the loop was completed, the list would be deleted, releasing all of the memory at once. This required a number of calls to the memory freeing routine which turned out to be very slow. The memory pool scheme turned out to be much more efficient, allocating and releasing large blocks of memory at once. A further improvement to this method was made when the memory in use was not released after each iteration, but the blocks were maintained, adding new elements to the front and keeping a pointer to the end of the list.

Many problems were encountered attempting to get reliably detected edges from the image in question. Camera focus and scene illumination turned out to be important factors in aiding the detection of edges, as well as the flicker effect produced by flourescent lights. For some reason, edges could be detected very reliably on the full image, but when the vertical scale variable was increased, many detected edges became either jagged, fragmented, or they disappeared altogether. An attempt was made to use only alternating horizontal pixels (horizontal scaling) to see if this, combined with the vertical scaling, would produce reliable edges, but a lot more information was lost this way.

By far the most important part of parallelizing the motion tracking system was getting the interprocessor communication working smoothly and faultlessly. At times the communication was suspected to be unreliable, as the system would halt execution seemingly waiting for data to be transmitted somewhere in the pipeline. Many different possible solutions were examined to facilitate more robust and efficient data transfers. Large messages were broken down into smaller pieces and processors were given different tasks (to correspond to correct physical pipelining) in attempts to improve communications. Another programming language, Parallel C, was even investigated to see if its communication primitives would be of use in this system. However, Parallel C does not support communication between nodes which aren't physically connected, so it proved unusable. These efforts aided in the understanding of the factors involved, but did not improve the overall communication performance.

The biggest hindrance to establishing dependable communications and enabling the expedient implementation of the system was the absence of a reliable debugging utility. It is extremely important in parallel programming to have a reliable debugger working on all nodes simultaneously to help solve problems. This debugger must have the capability to set breakpoints at any stage of execution on any node, and display different variables and data structures. Most nodes in any multiple processor system will not have any display capabilities to show their current status, so debuggers are invaluable in this capacity.

Unfortunately, debuggers are also very difficult to institute into parallel processing systems. There are a large number of synchronization issues to deal with, such as displaying variable information when blocked and waiting for communication. The C40 debugger was available for use, but could not be set up to work faultlessly. The alternative to using a debugger was to output as many status messages as possible to learn about the state of all nodes on the system. This was a very difficult and inefficient way of obtaining information, especially since execution had to be halted to gain much of this knowledge. Also, the variable nature of the incoming data caused each execution to be different, leading to problems in different stages of execution for every trial.

To aid in the gathering of information about the system, many routines were added to enable partial results to be displayed. For example, the system has the ability to show all of the edges that were detected in the image, as well as all line segments used in the matching process.

4.4 Performance Enhancing Modifications

This section will summarize the changes to the system which resulted in drastically improved overall system performance. Most modifications were performed in attempts to relieve bottlenecks in the system since attaining more efficient performance in these stages would have immediate effects on the execution of the system as a whole.

After a new convolution routine was written and optimized to be as efficient as possible, full image convolutions were still taking about 6 seconds to complete. The institution of a fast and efficient assembly language convolution routine using on-board memory, along with the use of smaller convolution kernels, improved the performance of the system to the point where the convolution routine was no longer the computational bottleneck (see results in Chapter 5). The use of vertical scaling further improved the convolution performance.

The edge linking process effectiveness was enhanced through the addition of the memory allocation routines. Also, the way in which data structures were constructed aided the performance. The points are completely gathered for each edge segment before the edge structure is allocated in case the sequence of points is too short to merit a new edge. Also, an array structure is used since the number of points is now known at allocation time. This also precludes the necessity of disassembling the data structure before communication.

A major decrease in execution time was gained through the use of the C40 library's cache routines. These routines allowed the use of the on-chip memory caches, and drastically improved performance on large loops and computationally intensive sections of code. The caches store often-repeating sections of code, thus greatly reducing the number of needed off-chip accesses. The external buses are also freed for other DMA operations or memory fetches. The CACHE_ON() call sets the *Cache Enable Bit* allowing the cache to be used. The CACHE_OFF() call in turn disables the cache. The cache routines were utilized with the convolution, linking, and zero crossing routines, and any other code sections where data manipulation was performed on the image. These changes resulted in improved task performance, sometimes by a few orders of magnitude.

Another large increase in performance resulted from selective processing of only the relevant portion of the input image, namely that region which contains the object. The most recent object position information can be obtained from the display node. When the model is projected for display, the horizontal and vertical extrema can be noted, and a "window" created in the image outside of which no processing will be performed. This window is enlarged slightly to allow for some object movement between frames.

Convolving a smaller region of the image will obviously speed up the convolution and zero crossing stages of the process, and a smaller image also means fewer line segments so that the edge linking, line creation, and matching processes are improved as well. The amount of improvement depends on the size of the object and its proximity to the camera, but speed-ups of 4x were not uncommon. Communication performance was also improved as a result since only a portion of the previous image needed to be transferred. This also meant that the sub-image needed packing to remain sequential and occupy less space. The zero crossing sub-image is unpacked at the edge linking phase so that edge tracking can be performed gathering the correct pixel coordinates for the edge data. The hitchhiking original sub-image does not need to be unpacked until the display stage.

Following the same idea, a further improvement was garnered by initially uncompressing only the relevant rows in the input image. The remaining rows are ignored at the convolution stage. Further improvements to the motion tracking system included the addition of least-squares line fitting, and the introduction of an option file to change run-time parameters.

Table 4.1 outlines the major improvements and their effect on the system's performance. Many smaller fine-tuning improvements were made but are omitted from this table. Some of the earlier entries are only approximate since not all of the improvements and features had yet been added.

Improvement	*Resulting loop time (sec)*
Dynamically allocate edge structures (160x160 image)	18.0
Cache routines in linking	14.2
Addition of free memory pool (512x460 image)	20.5
Assembly convolution routine on a 2nd node	3.68
Vertical scaling	3.24
Separate zero crossing and linking nodes	2.52
Window processing for zero and link	1.59
Split matching onto 5th node	0.75
Window processing - convolution and zero	0.420
Cache routines for zero crossing	0.400
Communicate only relevant subimage	0.265

Table 4.1: History of system improvements.

4.5 The Data Parallelism Approach

A completely different approach could have been taken in parallelizing the motion tracking system. Instead of splitting the job into sub-tasks and distributing these sub-tasks among processors in a pipelined fashion, the data parallelism paradigm could have alternatively been employed.

To recap (see Section 3.1), data parallelism involves partitioning the image into parts or sub-images which are each processed by a different node. Each node performs the same tasks on the sub-images, and results are combined when complete. This method would have its advantages and disadvantages.

This type of implementation would be favourable for the following reasons:

- When more processing power became available in the form of additional C40 nodes, it would be a more straightforward task to incorporate these processors into the system to improve performance. The data parallel edge detection tasks could be supplemented simply by partitioning the image into more pieces and processing on all available nodes.

- More individual communications may be necessary to divide and recombine data, but these are all similar in nature, smaller in size, and simple to control. Analyzing these transfers would reveal that all image division and result collection would involve as much data as two individual communication stages in the task parallelism scheme (less if movement of the original image is not required). In the end, fewer total communication stages would be necessary and the total amount of data passed between processors would be decreased.

- More balanced dataflow would result.

- Temporal latency would be reduced, though total system throughput would also be reduced.

- Much less dismantling and reconstruction of data structures would be required.

The disadvantages of a data parallelism approach are summarized in the following list:

- Many more modifications to the algorithm would be required for this approach.

- Edge effects are present when combining results.

- Some form of pipelining must still be employed due to the high-level matching operations.

- It is not as easy to gauge implementation progress; results must be combined from all nodes after each modification is made to see if the system is working properly.

- Processing loads may be very uneven if different areas of the image are more interesting or complicated.

The most common data parallelism configuration would involve reserving two C40 nodes; one for image capture and distribution, the other for collection of results and output; and employing the remaining processors as worker nodes which could process the subimages. The edge detection processes could be performed on each node, creating a number of sets of line segments found in the image.

In attempting to exploit the data parallelism for as long as possible, one may be tempted to then index these line segments and select candidate matches in parallel. The Newton-Raphson convergence method requires the availability of all matched segments, so at this point, all of the candidate matches would need to be communicated to a single node which would complete the solving process. The problem in this approach is that line segments corresponding to model edges may be divided across subimage boundaries, and thus exist partially on two separate nodes. For valid matching to be performed, these segments must be joined before the matching starts. Hence, the results of the edge detection performed on all subimages must be combined on a central processor where matching can be performed.

A strict data parallelism scheme would see the edge detection results passed back to the same node which partitioned the image initially. An advantage of this approach would be that the original image would still be stored here and would not have to be transferred to another node for display. Otherwise, a separate transfer sequence would have to occur. Alternatively, the worker nodes could save their sub-images and pass them unchanged to the matching node where the original could be reconstructed.

As can be plainly seen, any data parallelism approach must necessarily embody some pipeline mechanisms, hence it would be more of a hybrid approach. The ultimate system would likely contain elements of both paradigms.

Chapter 5
Results

Processing time results were gathered by utilizing the timers on each C40 node. The execution times for various stages of computation would be compiled and passed to the display node in the form of a status message. This information could then be passed to the system user.

A file box model was used as the object to be tracked. This model is quite simple with straight edges, and a hinged, translucent lid. The dimensions of the box are 7x6x6 inches, and it was situated about 5 feet from the camera so that it subtended an angle of about 6 degrees. This resulted in a spatial extent of about 145x135 pixels in the image plane. The image size stored by the system is 512x460 pixels, so the model takes up about 10% of the total image area. This makes some tasks more difficult such as model edge detection. However, at this distance the model has more freedom of movement while still being within a trackable velocity range.

Figure 5.1 shows the setup of the experiment, and Figure 5.2 gives an example of the edges that are detected in the image. A 5x5 convolution kernel was used with least-squares line fitting.

Tables 5.2 through 5.6 give the timing results for the individual processing nodes, showing how long each step of the task took to perform. The timing data was collected and averaged over a number of iterations. Some operations, especially the linking and matching routines, have a quite high variance of $\pm$ 15 ms.

The results displayed in table 5.7 show the complete execution times for each processing loop, including interprocessor communication. The duration of each loop was approximately 265 ms, but this value is highly dependent on the proximity of the object and the complexity of the background and other nearby objects. The time spent completing the required sub-tasks is given, along with the communication time in receiving and sending data. These values also

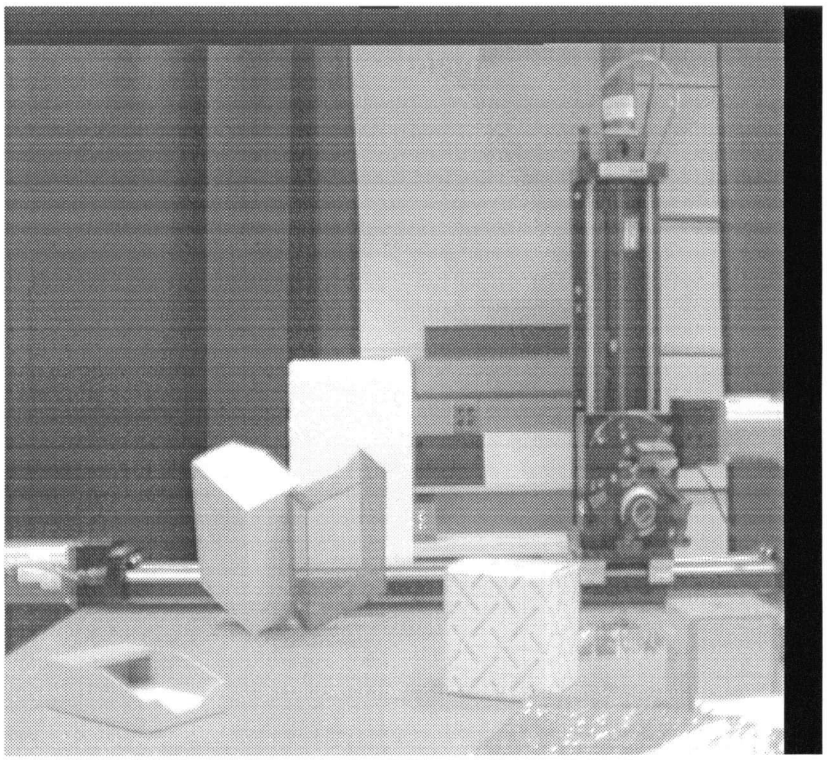

Figure 5.1: The motion tracking setup.

include the time spent waiting for other processors to complete computations. The "data in" time for the grab node refers to the time spent acquiring the input image. The "data out" time for the display node refers to all operations performed in order to display the results.

A sample of the display is given in Figure 5.3 which shows how the tracking system was able to "lock on" to the object model. Figure 5.4 shows another frame from the same tracking sequence. In this case, the convergent nature of the least-squares matching iterations is displayed. Each iteration produces a model position which more closely corresponds to the correct object position than the previous result.

It is a little difficult to determine what is happening on each processor in Table 5.7, so a slightly different layout is presented in Figure 5.5. This figure shows the processing and communications times, and also depicts the amount of time that the processor is "blocked" waiting for data from the previous processor or waiting for the next processor to finish its

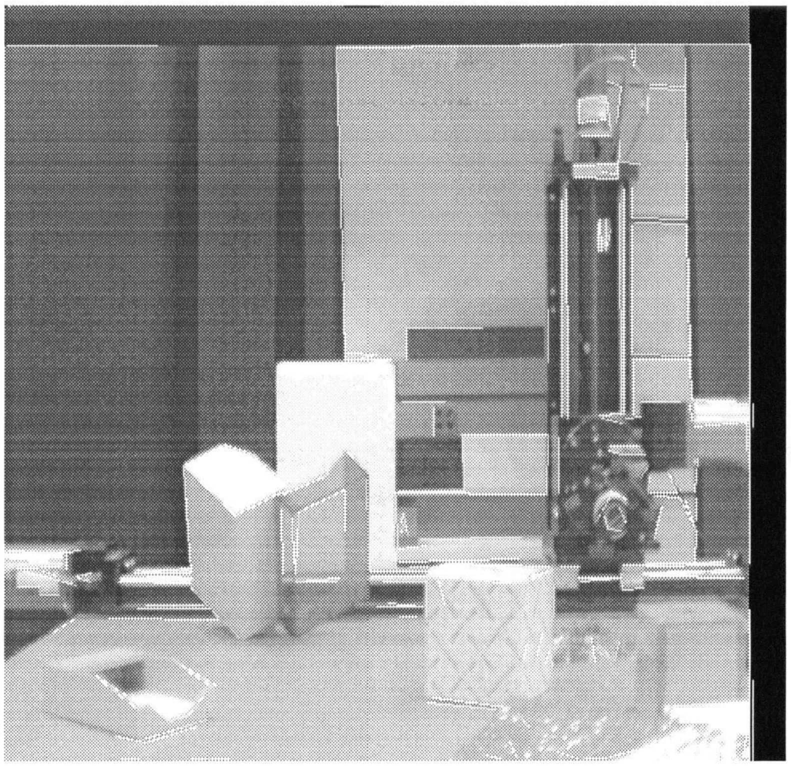

Figure 5.2: Sample of edges detected from input image.

work so it can pass its results down the pipeline. This type of diagram makes it easy to spot bottlenecks.

Another execution with a slightly different setup is given in Figure 5.6. This experiment is important because it shows that with only a few minor changes, the display node can become the bottleneck. The processors in this case are blocked at the completion of their work as they wait for the display node to become free. It is also seen that the display node seems to be taking far too much time for the tasks it performs. This is because it spends a lot of time preparing the original image for display. One way to speed this part of the processing up would be to give the option of completely suppressing the display of the image. This would speed the rest of the pipeline up as well since a lot less information would need to be communicated. The final display would then consist only of the model's edges projected from its current position, not overlaid on the original image.

task	*time(ms)*
Image capture	36
Vertical scaling	40
Decompression	20
Convolution (5x5)	140
Pack image	20
Total execution time	245

Table 5.2: Grab node timings.

task	*time(ms)*
Convert to float	10
Find zero crossings	27
Total execution time	37

Table 5.3: Zero node timings.

task	*time(ms)*
Unpack image	20
Link zero crossings	40
Show edges	10
Make line segments	27
Total execution time	97

Table 5.4: Link node timings.

task	*time(ms)*
Create edge structure	10
Create line array	10
Project model	20
Matching	75
Total execution time	115

Table 5.5: Match node timings.

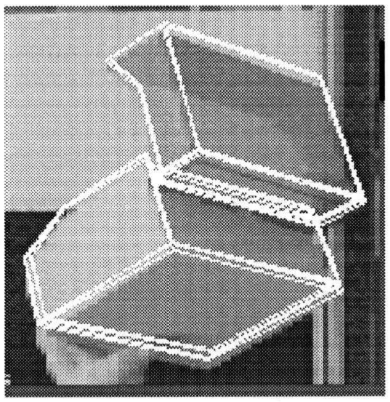
Figure 5.3: Tracking file box model - frame 50.

task	*time(ms)*
Unpack image	20
Show lines	20
Copy to RGB	60
Unscale image	83
DMA image copy	46
Clear message queue	11
Total execution time	240

Table 5.6: Display node timings.

task	*grab*	*zero*	*link*	*match*	*display*
Data in	105	207	161	145	23
Process	140	37	97	115	103
Data out	20	20	12	20	137
total	265	265	265	265	265

Table 5.7: System iteration timings. All values in ms.

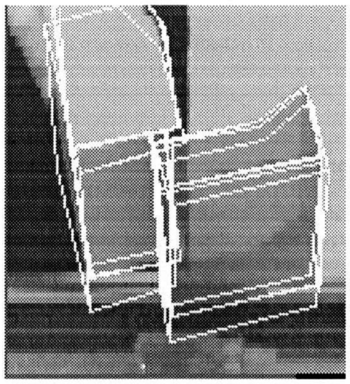

Figure 5.4: Tracking file box model - frame 100.

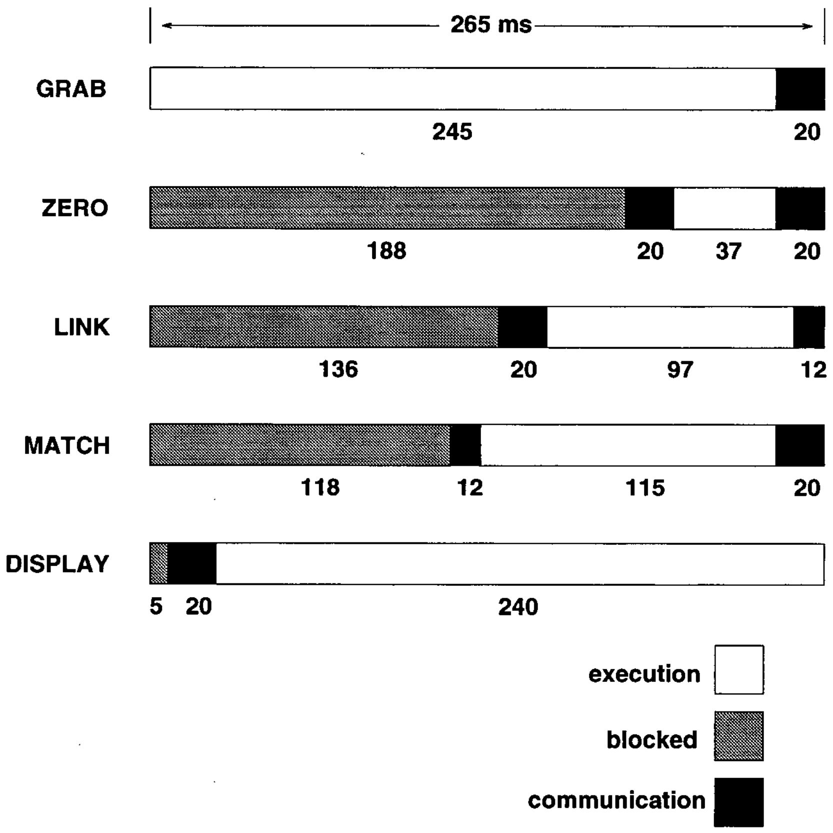

Figure 5.5: Individual processor timings - trial 1 (ms).

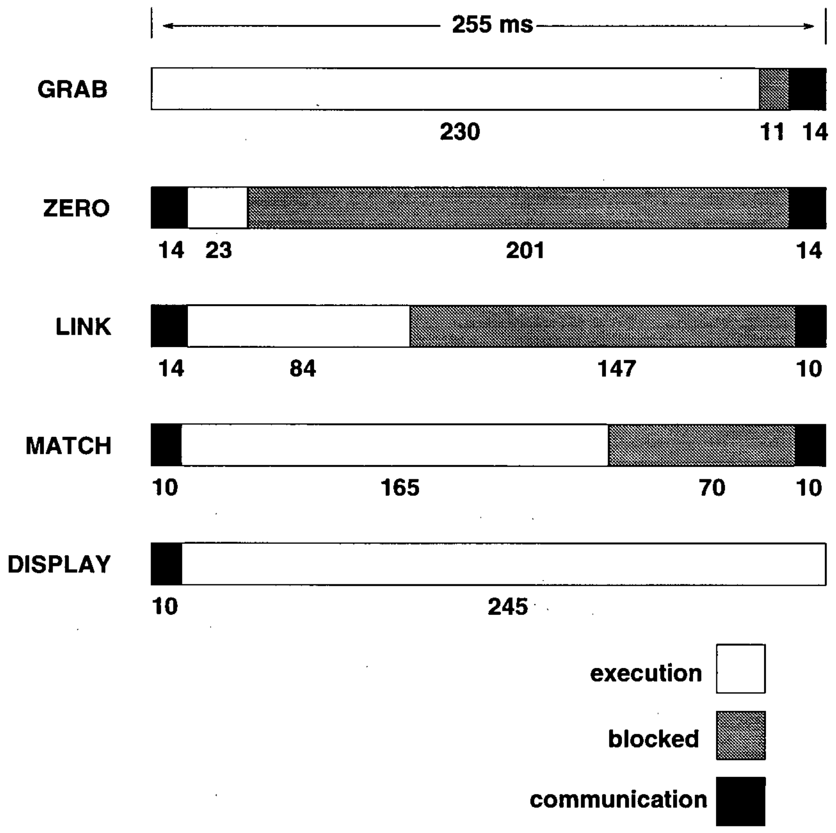

Figure 5.6: Individual processor timings - trial 2 (ms).

Chapter 6
Conclusion

This paper has described some of the factors and issues involved in converting a sequential system to a multiprocessor system. It did this using the concrete example of a motion tracking system, converting it to a real-time parallel implementation. This motion tracking system was described, discussing many implementation alternatives and decisions, and some of the problems that were encountered.

If anything, this experiment showed that the parallelization of sequential systems is not an automatic, uncomplicated procedure. Many issues must be considered, and a lot of unforeseen complications are bound to arise. It also stressed the importance of a reliable debugging utility, as many of the issues faced could have been simplified or even avoided with this type of programming aid.

The present performance of the motion tracking sytem may not be adequate for use in many current real-time applications; however, certain modifications could be performed to the system which would increase system effectiveness to the point where real-time operations are possible.

Possible Extensions

Some different approaches that could be taken include the use of specialized hardware, further distribution of the processing load among more processors, increased use of the DMA capabilities, and the employment of further vision techniques to make the motion tracking system more robust.

A large chunk of processing time is due to the convolution routine. A quick way to improve performance would be to split the image acquisition and convolution routines onto separate processors. The grab node would then perform all necessary image scaling and decompression and pass the image's processing window to the convolution node. This node would then be able to concentrate all of it's computation cycles on performing the convolution. This operation could in turn be improved by applying data parallelism and using a number of processors to convolve the image at this stage. The results would be combined and passed to the zero crossing processor with limited edge effects.

The other method of speeding certain operations, including the convolution, would be to utilize specially designed hardware. A "convolution engine" could process images in speeds approaching video rate.

The system could also make far better use of the capabilities of the DMA coprocessor. Currently, the DMA is utilized only during communication and image display. The input and some computation stages could also benefit, such as performing concurrent convolution and image acquisition operations. Perhaps the DMA could also be used to communicate partial results between processors. This could decrease the idle time on some processors and the overal system latency without adding too much communication overhead.

The latency of the system could be reduced by formulating a scheme in which partial results are passed along to the next processor. This processor can then begin computations without waiting for the rest of the data to arrive. Regions of the convolved image or partial zero crossing results could be processed in this manner. Also, as single edges are detected in the linking and line creation stage, they can be immediately transferred to the matching stage for indexing and ordering.

Another idea that has been discussed is to supress the display of the original image. The observable results would then involve only the projected model edges. The exact position of the object would still be known and the processing time of the final display node would be dramatically decreased. The processor's duties would be reduced to receiving and displaying the model pose information, handling status messages, and communicating model pose data back to the grab node. These steps could be executed in as quickly as 40 ms!

The image supression would also hasten communications throughout the pipeline, and would also alleviate the processing load on the grab node. Further DMA and scaling improvements to the grab node could then lead to the matching node being the system bottleneck, with loop times around 160 *ms*. This processing rate is becoming attractive for real-time systems.

If additional processors became available they could be utilized to speed the computations at some stages of the pipeline. Data parallelism could be employed at the convolution stage, or all of the edge detection tasks could be combined into one stage, utilizing a number of processors to detect edges in different regions of the image.

There are many other long-term research areas in motion tracking. Advances in these areas could result in systems which improved performance and tracking capabilities. More efficient methods of specifying object model parameters are required. Motion tracking could also be extended to track models with curved edges by finding smooth image curves [45]. Currently, the system uses only object edges as features, but vertices, cylindrical, and spherical surfaces giving rise to "horizon lines" could also be modeled.

BIBLIOGRAPHY

[1] R. Anderson, "The Design of a Ping-Pong Playing Robot", MIT Press, Cambridge, 1988.

[2] M. Asada, M. Yachida, S. Tsuji, "Analysis of Three-Dimensional Motions in Blocks World", *Pattern Recognition*, 17, 1984, pp 57-71.

[3] N. Ayache, I. Cohen, I. Herlin, "Medical Image Tracking", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 285-302.

[4] A. Blake, "Computational Modeling of Hand-Eye Coordination", *Phil. Trans. Royal Soc. London*, 1992.

[5] Jon Bradley, "TMS320C40 Hardware Applications", *Digital Signal Processing Applications with the TMS320 Family*, vol 3, P. Papamichalis ed., Texas Instruments Inc., 1990, pp 333-363.

[6] Alistair J. Bray, "Tracking Objects using Image Disparities", *Image and Vision Computing*, 8(1), 1990, pp 4-9.

[7] T.J. Broida, R. Chellappa, "Kinematics and Structure of a Rigid Object From a Sequence of Noisy Images", *Proc. Workshop on Motion: Representation and Analysis*, IEEE, 1986, pp 95-100.

[8] C.R. Brown, M. Rygol, "MARVIN: Multiprocessor Architecture for Vision", *Applying Transputer Based Parallel Machines* (Proc. OUG-10), IOS Press, 1989.

[9] John Canny, "A Computational Approach to Edge Detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6), 1986, pp 679-698.

[10] Vipin Chaudhary, J.K. Aggarwal, "Parallelism in Computer Vision: A Review", *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar et al eds., Springer-Verlag, New York, 1990, pp 271-309.

[11] A.N. Choudhary, S. Das, N. Ahuja, J.H. Patel, "A Reconfigurable and Hierarchical Parallel Processing Architecture: Performance Results for Stereo Vision", *Proc. IEEE 10th International Conf. on Pattern Recognition*, Vol 2, 1990, pp 389-393

[12] R. Cipolla, A. Blake, "The Dynamic Analysis of Apparent Contours", *Proc. of the Third International Conference on Computer Vision*, Osaka, 1990, pp 616-623.

[13] R. Cipolla, A. Blake, "Motion Planning Using Image Divergence and Deformation", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, 1992, pp 189-201.

[14] Jill D. Crisman, "Color Region Tracking for Vehicle Guidance", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 107-122.

[15] J. Crowley, P. Stelmaszyk, C. Discours, "Measuring Image Flow by Tracking Edge-lines", *Proc. of the Second International Conference on Pattern Recognition*, Rome, 1988, IEEE Press, pp 658-664.

[16] R. Curwen, A. Blake, R. Cipolla, "Parallel Implementation of Lagrangian Dynamics for Real-Time Snakes", *British Machine Vision Conference*, P. Mowforth ed., Glasgow, 1991, pp 29-35.

[17] Rupert Curwen, Andrew Blake, "Dynamic Contours: Real-Time Active Splines", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 39-58.

[18] R. Deriche, O.D. Faugeras, "Tracking Line Segments", *Image and Vision Computing*, 8, 1990, pp 261-270.

[19] Sven J. Dickinson, Piotr Jasiobedzki, Goran Olofsson, Henrik I. Christensen, "Qualitative Tracking of 3-D Objects using Active Contour Networks", *Proc. Conf. Computer Vision and Pattern Recognition*, Seattle, 1994, pp 812-817.

[20] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, 23(2), 1990, pp 5-16.

[21] R.M. Endlich, D.E. Wolf, D.J. Hall, A.E. Brain, "Use of a Pattern Recognition Technique for Determining Cloud Motions from Sequences of Satellite Photographs", *Jour. Appl. Meterol.*, 10, 1971, pp 105-117.

[22] D. Gennery, "Tracking Known Three-Dimensional Objects", *Proc. National Conference on Artificial Intelligence*, Pittsburgh, 1982, pp 13-17.

[23] Donald B. Gennery, "Stereo Vision for the Acquisition and Tracking of Moving Three-Dimensional Objects", *Techniques for 3-D Machine Perception*, A. Rosenfeld ed., Elsevier Science Publishers B.V., 1986, pp 53-73.

[24] D. Gennery, "Visual Tracking of Known Three-Dimensional Objects", *International Journal of Computer Vision*, 7(3), 1990.

[25] A.L. Gilbert, M.K. Giles, G.M. Flachs, R.B. Rogers, Y.H. U, "A Real-Time Video Tracking System", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1), 1980, pp 47-56.

[26] J. Hallam, "Resolving Observer Motion by Object Tracking", *Proc. 8th International Joint Conference on Artificial Intelligence*, vol 2, 1983, pp 792-798.

[27] L. Hamey, J. Webb, I. Wu, "Low-Level Vision on Warp and Apply Programming Model", *Parallel Computation and Computers for Artificial Intelligence*, Ed. J. Kowalik, Kluwer Academic, Boston, 1987.

[28] Chris Harris, "Tracking with Rigid Models", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 59-74.

[29] W.D. Hillis, "The Connection Machine: A Computer Architecture based on Cellular Automata", *Physica*, 1984.

[30] Daniel P. Huttenlocher, Jae J. Noh, William J. Rucklidge, "Tracking Non-Rigid Objects in Complex Scenes", *Fourth International Conference on Computer Vision*, Berlin, 1993, pp 93-101.

[31] INMOS Ltd., "Transputer Technical Notes", Prentice-Hall, 1989.

[32] P. Jasiobedzki, "Adaptive Adjacency Graphs", *Proc. SPIE Geometric Methods in Computer Vision*, San Diego, 1993, pp 294-303.

[33] M. Kass, A. Witkin, D. Terzopolous, "Snakes: Active Contour Models", *International Journal of Computer Vision*, 1(4), 1988, pp 321-331.

[34] E. Kent, M. Shneier, R. Lumia, "Pipe - Pipelined Image Processing Engine", *Journal of Parallel and Distributed Computing*, 2(1), 1985, pp 50-78.

[35] D. Koller, K. Daniilidis, H.H. Nagel, "Model-Based Object Tracking in Monocular Image Sequences of Road Traffic Scenes", *International Journal of Computer Vision*, 10(3), 1993, pp 257-281.

[36] J.A. Leese, C.S. Novak, V.R. Taylor, "An Automated Technique for Obtaining Cloud Motion from Geosynchronous Satellite Data Using Cross-Correlation", *Jour. Appl. Meteorol.*, 10, 1971, pp 118-132.

[37] F. Leymaire, "Tracking and Describing Deformable Objects Using Active Contour Models", Technical Report TR-CIM-90-9, McGill Research Center for Intelligent Machines, McGill University, Montreal.

[38] J.J. Little, G. Blelloch, T. Cass, "Parallel Algorithms of Computer Vision on the Connection Machine", *International Conference on Computer Vision*, 1987.

[39] James J. Little, Guy E. Blelloch, Todd A. Cass, "Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3), 1989, pp 244-257.

[40] J.J. Little, R. Barman, S. Kingdon, J. Lu, "Computational Architectures for Responsive Vision: The Vision Engine", *University of British Columbia, Department of Computer Science, TR 91-25*, November, 1991.

[41] David G. Lowe, *Perceptual Organization and Visual Recognition*, Kluwer Academic Publishers, Boston, 1985.

[42] David G. Lowe, "Three-Dimensional Object Recognition From Single Two-Dimensional Images", *Artificial Intelligence*, 31(3), 1987, pp 355-395.

[43] David G. Lowe, "Fitting Parameterized Three-Dimensional Models to Images", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5), 1991, pp 441-450.

[44] David G. Lowe, "Robust Model-Based Motion Tracking Through the Integration of Search and Estimation", *International Journal of Computer Vision*, 8(2), 1992, pp 113-122.

[45] David G. Lowe, "Organization of Smooth Image Curves at Multiple Scales", *International Journal of Computer Vision*, 3, 1989, pp 119-130.

[46] David Marr, Ellen Hildreth, "Theory of Edge Detection", *Proc. Royal Society of London, B*, 207, 1980, pp 187-217.

[47] Stephen Marshall, "Parallel Edge Detection and Related Algorithms", *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, I. Pitas ed., John Wiley & Sons Ltd., Chichester, 1993, pp 91-122.

[48] R.F. Marslin, G.D. Sullivan, K.D. Baker, "Kalman Filters in Constrained Model-based Tracking", *Proc. Brit. Mach. Vis. Conf.*, Glasgow, 1991, pp 371-374.

[49] P. Matteucci, C.S. Regazzoni, G.L. Foresti, "Real-Time Approach to 3-D Object Tracking in Complex Scenes", *Electronics Letters*, 30(6), 1994, pp 475-477.

[50] L.H. Matthies, T. Kanade, R. Szeliski, "Kalman Filter-Based Algorithms for Estimating Depth from Image Sequences", *International Journal of Computer Vision*, 3, 1989, pp 209-236.

[51] A.D. Morgan, E.L. Dagless, D.J. Milford, B.T. Thomas, "Road Edge Tracking for Robot Road Following: A Real-Time Implementation", *Image and Vision Computing*, 8(3), 1990, pp 233-240.

[52] D.W. Murray, D.A. Castelow, B.F. Buxton, "From Image Sequences to Recognized Moving Polyhedral Objects", *International Journal of Computer Vision*, 3, 1989, pp 181-209.

[53] G.R. Nudd, T.J. Atherton, N.D. Francis, R.M. Howarth, D.J. Kerbyson, R.A. Packwood, G.J. Vaudin, "A Hierarchical Multiple-SIMD Architecture for Image Analysis", *Proc. IEEE 10th International Conf. on Pattern Recognition*, vol 2, 1990, pp 642-647.

[54] Joseph O'Rourke, Norman I. Badler, "Model-Based Image Analysis of Human Motion Using Constraint Propagation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(6), 1980, pp 522-536.

[55] I. Page, "The Disputer: A Dual Paradigm Parallel Processor for Graphics and Vision", in *Parallel Architectures and Computer Vision*, Clarendon Press, Oxford, 1988, pp 201-216.

[56] Michael J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, 1987.

[57] John W. Roach, J.K. Aggarwal, "Determining the Movement of Objects from a Sequence of Images", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(6), 1980, pp 554-562.

[58] M. Rygol, S. Pollard, C. Brown, J. Mayhew, "A Parallel 3D Vision System", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 239-262.

[59] J. Schick, E.D. Dickmanns, "Simultaneous Estimation of 3D Shape and Motion of Objects by Computer Vision", *Proc. IEEE Workshop on Visual Motion*, Princeton, 1991, pp 256-261.

[60] D.B. Shu, J.G. Nash, C.C. Weems, "A Multiple-Level Heterogeneous Architecture for Image Understanding", *Proc. IEEE 10th Conf. on Pattern Recognition*, vol 2, 1990, pp 629-634.

[61] Lenoel Sousa, Moisés Piedade, "Low Level Parallel Image Processing", *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, I. Pitas ed., John Wiley & Sons Ltd., Chichester, 1993, pp 25-52.

[62] R.S. Stephens, "Real-Time 3-D Object Tracking", , *Image and Vision Computing*, 8(1), 1990, pp 91-96.

[63] G. Sullivan, "Visual Interpretation of Known Objects in Constrained Scenes", *Phil. Trans. Royal Soc. London*, 1992.

[64] D. Terzopolous, K. Waters, "Analysis of Facial Images Using Physical and Anatomical Models", *Third International Conference on Computer Vision*, Osaka, 1990, pp 727-732.

[65] D. Terzopolous, R. Szeliski, "Tracking with Kalman Snakes", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 3-20.

[66] "TMS320C4x User's Guide", Texas Instruments Inc., 1991.

[67] "TMS320C4x C Source Debugger User's Guide", Texas Instruments Inc., 1992.

[68] D.W. Thompson, J.L. Mundy, "Model-Based Motion Analysis: Motion From Motion", *Robotics Research: The Fourth International Symposium*, R. Bolles and B. Roth eds., MIT Press, Cambridge, 1988, pp 299-309.

[69] J. Tsotsos, J. Mylopoulos, H. Covvey, S. Zucker, "A Framework for Visual Motion Understanding", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(6), 1980, pp 563-573.

[70] S. Tsuji, M. Osada, M. Yachida, "Tracking and Segmentation of Moving Objects in Dynamic Line Images", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(6), 1980, pp 516-522.

[71] R.J. Valkenburg, C.C. Bowman, "Kiwivision-2 - A Hybrid Pipelined / Multi-transputer Architecture for Machine Vision", *Proc. SPIE 1004*, M.J.W. Chen ed., 1989, pp 91-96.

[72] G. Verghese, C.R. Dyer, "Real-Time Model-Based Tracking of Three-Dimensional Objects", *University of Wisconsin, Computer Sciences TR 806*, November 1988.

[73] G. Verghese, K. Gale, C.R. Dyer, "Real-Time, Parallel Motion Tracking of Three-Dimensional Objects from Spatiotemporal Sequences", *Parallel Algorithms for Machine Intelligence and Vision*, Kumar et al eds., Springer-Verlag, New York, 1990.

[74] R. Wallace, M.D. Howard, "HBA Vision Architecture: Built and Benchmarked", *IEEE Trans. Pattern Analysis and Machine Intelligence*, 11(3), 1989, pp 227-232.

[75] J. Weng, T.S. Huang, N. Ahuja, *Motion and Structure from Image Sequences*, Springer-Verlag, New York, 1993.

[76] A.D. Worrall, R.F. Marslin, G.D. Sullivan, K.D. Baker, "Model-based Tracking", *Proc. Brit. Mach. Vis. Conf.*, Glasgow, 1991, pp 310-318.

[77] Alan Yuille, Richard Szeliski, "Deformable Templates", *Active Vision*, A. Blake and A. Yuille eds., MIT Press, Cambridge, 1992, pp 21-38.