[9]

Parallel Architectures for Fast 3D Machine Vision

Chris R Brown and Chris M Dunford

AI Vision Research Unit
University of Sheffield, Sheffield S10 2TN, UK

1. Introduction

The *TINA* software suite, developed within *AIVRU* during the period 1984 to 1987, has demonstrated an encouraging level of visual competence by delivering (from stereoscopic TV images of a relatively cluttered scene) three-dimensional edge-based geometrical scene descriptions sufficiently accurate to guide a robot arm in a pick-and-place task. Against this success must be offset the large computational effort required to pass even a single image through the *TINA* suite.

This paper describes work undertaken by *AIVRU* to develop a computing engine of realistic cost, yet powerful enough to provide three-dimensional machine vision at speeds commensurate with the real-time needs of an assembly robot, or an autonomous guided vehicle. Firstly, we make some observations about the opportunities for exploiting parallel MIMD architectures and other specialised hardware, including sequential pipelined frame-rate processors. Secondly, we discuss the results of a pilot study in which parallel implementations of the *Canny* edge detector and the PMF stereo matching algorithm were implemented on an array of eight transputers. Thirdly, we describe *'Marvin'* - a Multi-processor ARchitecure for VIsioN, currently being developed within *AIVRU*.

To give some idea of the extent of improvement in speed being sought, it is interesting to examine the CPU time required to process an image through the *TINA* suite at the time it was originally demonstrated, which for a typical scene at 256 by 256 resolution are shown in Table 1. These times were obtained on a Sun 2 with a *SKY* floating point accelerator.

Process Stage	Time (sec)
Canny (Edge detection)	428
Rectify (Convert to parallel camera geometry)	57
PMF (Stereo matching)	1118
Connect (Establish connectivity of 3-D edges)	214
GDB (Classify connected edges as lines & arcs)	1442
Model Matcher (Find 3-D match of edge model)	300
TOTAL:	3559

Table 1

Exactly what is implied by 'real-time' operation of the system is open to debate, but a throughput of 1 image per second is probably the minimum requirement. In addition, a move to higher resolution (say 512 by 512) images is desirable. We seek, therefore, a speed increase of order 10,000.

Much work is in progress within *AIVRU* to reduce the computational load by refinement of the algorithms; for example by restricting the areas of image which are processed to specific regions of interest, and to further reduce the amount of searching by predicting forward within a sequence of images. The work reported here however, is concerned with the provision of more raw computing power, through the use of parallel MIMD architectures.

The transputer is an ideal candidate as a processing unit in such a system, and our work centres around the use of this device. A powerful 32-bit processor in its own right, it has four high-speed serial links that enable the construction of highly parallel architectures without the system engineering problems and bus bandwidth limitations experienced by shared memory designs. The transputer's flexibility and price allow a modular system to be constructed for prototyping fast vision systems, without imposing a heavy financial burden.

2. The Exploitation of Parallelism

In this section we discuss the parallelism inherent in the *TINA* processing stages, and how this might be exploited. The discussion focusses mainly on the use of a transputer array, although some comments on the use of specialised pipelined hardware are also made. It is important to note that we seek opportunities for parallelism on quite a large scale. Schemes which allow us to exploit, say, two or three transputers are simply not worthwhile. Thus, the extent to which any given scheme can be extended to greater numbers of transputers, and the extent to which a linear increase in speed is achieved, are important issues.

Three types of parallelism can be identified within the processing stages: *spatial*, *featural*, and *temporal*. We will consider these in turn.

2.1 Spatial Parallelism

Each processor is allocated a patch of the image. This is appropriate for operators which require access to relatively small pixel neighbourhoods, such as Canny and *PMF*. Load-balancing (that is, arranging for each processor to have the same amount of work to do) can be performed by adjusting the size of the patches. Although in principle a decomposition into a 2-D array of rectangles could be used (see *Fig.* 2), in practice the use of 'slices' (rectangles extending the full width of the image) is easiest (*Fig.* 1). If 2-D decomposition is used, it is difficult to adjust the size of the patches to balance the processing load, and still have the patches tesselate the image. Also, because of the 'raster scan' order in which digital video busses operate, it is somewhat easier from an engineering standpoint to distribute data in complete rasters.

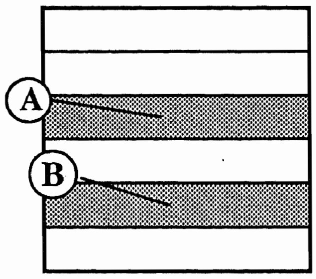

Fig 1. 1-D Spatial Partitioning

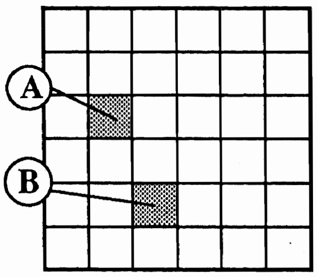

Fig 2. 2-D Spatial Partitioning

2.1.1 Data flow within spatially parallel systems

The data flow within spatially parallel systems is complicated by the fact that most algorithms require access to pixel neighbourhoods surrounding the pixels for which they are to deliver output. This can be handled in various ways:

(a) By migrating data at the slice boundaries across the transputer links. The order of processing may be important, especially in iterative algorithms such as *PMF*. For example, if all slices are processed top-to-bottom, and slice 4 needs complete or partial results from slice 3 etc., processing activity will ripple down through the slices and parallelism will be lost.

(b) By duplication of effort at the slice boundaries. That is, a slice's transputer will replicate some of the processing which its neighbour is performing on pixels near the slice boundary. The input slice will be bigger than the output slice.

These considerations lead to the following observation: if the width of the slice allocated to a processor is reduced, the processor spends a larger proportion of its time either talking to its neighbour or duplicating its neighbour's efforts. If the slice becomes narrower than the algorithm's neighbourhood size, this effect dominates and little speed increase is obtained. This yields a (very approximate) upper limit on the number of transputers which can be gainfully employed in this way of

```
image size/neighbourhood size
```

Taking *image size* = 512 and *neighbourhood size* = 8, yields an upper limit of 64 processors. At present, this value is larger than the number of transputers we have the money to buy -- though not by a large factor.

An alternative form of spatial parallelism could arise from multiple region of interest processing, in which a number of relatively small rectangular regions of the image have been identified as worthy of detailed processing. In this case the regions will not necessarily be horizontal slices, and there is no requirement for them to tesselate the image.

2.1.2 Load Balancing in Spatially Parallel Systems

Load balancing, as mentioned earlier, can be accomplished by adjusting the size of the slices. The computational load of some operations, such as *Canny*, is relatively insensitive to the actual content of the image. For others, such as *PMF*, cluttered regions take much longer. Since pictures (usually) have the important bits in the middle, a load-balanced partitioning will typically have wide slices at the top and bottom edges and narrower slices in the middle. Partitioning adjustments are ideally carried out before processing an image, if the 'cost' of processing each raster can be somehow evaluated before hand. For example, a simple count of the number of edges found by *Canny* could be used to guide a pre-partitioning of the data prior to *PMF*. Alternatively, if the system is being used to process a series of images, partition sizes can be adjusted after each image has been processed, on the assumption that each image in the sequence has a similar 'complexity distribution' to its predecessor.

2.2 Featural Parallelism

Each processor is allocated a subset of the geometrical features in the image (See *Fig.* 3). This is appropriate for

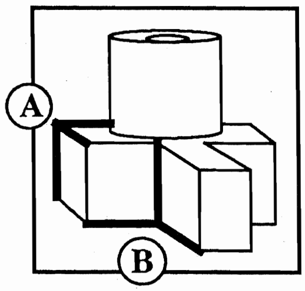

Fig 3. Featural Partitioning

operations on lines and regions which potentially could span large distances across the image. The rules by which features are allocated to processors are less obvious than for spatial parallelism, but could continue to have a spatial basis; for example, each processor is allocated a slice; features are allocated to processors on the basis of which slice their top-most point lies in. The real problems lie in the transformation of the data structures between spatial-parallel and feature-parallel stages, and in efficiently re-distributing the data around the processor array.

Load balancing in this scheme could be performed by using a 'processor farm', consisting of a master processor and a number of worker processors. The master keeps a list of busy and idle workers, and assigns 'work packets' to processors as they become idle. (A work packet might be, for example, an attempt to match a geometric feature in the image with a pre-computed object model). The fundamental characteristic of a processor farm is that the workers do not need to communicate or synchronise with one another. Processor farms are automatically load-balanced, and it is very easy to add more transputers; the master simply needs to be made aware of the increased number of workers. A desirable characteristic of a processor farm is a topology which provides short data paths between the master and all of its workers. Given that transputers have only four links, a ternary tree is perhaps sensible. The transputer is well-suited as a worker node as its design allows the passing of data from one link to another in parallel with the actual work process running on it. CPU time is required only to initiate the operation, thereafter DMA logic in the links carries out the transfer with minimal impact on the CPU. The master task should ensure that this overlap of processing and transferring data is exploited. Further, efficient processor farms reduce the CPU overhead and maximise the communications bandwidth of the network by sending small numbers of large messages in preference to large numbers of small ones.

2.3 Temporal Parallelism

Temporal Parallelism, or *pipelining*, refers to the use of several processing elements in series, with each element responsible for one stage of the processing. Typically this provides N-fold parallelism for only small values of N, simply because there are conceptually only a few stages in the pipeline. In practice each element could be a group of transputers which themselves employed spatial or featural parallelism. For example, one might envisage 4 groups as shown in *Fig.* 4. Different numbers of transputers are placed in each group to balance the processing loads of the various stages. Effective parallelism is achieved only if a continuous flow of images are to be processed, so that, for example, processor group *D* is processing image *N*, whilst group *C* is processing image *N+1*, and so on.

Insofar as it provides opportunity to use more transputers, pipelining increases the throughput of the system, but it does not reduce the overall latency from image acquisition to delivery of the geometry. For example, suppose each processor group in the pipelined system of Fig. 4 were able to perform its task in, say, 0.5 seconds. The pipeline would thus deliver fresh 3-D geometry every 0.5 seconds, but each one would be 2.0 seconds out of date. The same number of transputers employed without pipelining would achieve a four-fold increase in the fineness of the spatial or featural parallelism, thus (assuming the load was balanced) completing each processing stage in 0.125 seconds. It would still deliver results every 0.5 seconds, moreover, each would be only 0.5 seconds out of date.

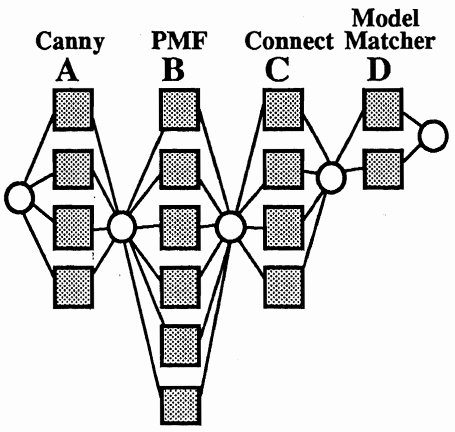

Fig. 4. Pipelining of Transputer Groups

These comments apply when general purpose processors are used at each stage in the pipeline, such that each processor is actually able to turn its hand to all processing stages. A very different picture emerges when we consider the use of specialised pipelined hardware which runs at video frame-rate. Such hardware offers extremely high performance for very specific tasks, of which convolution is a good example. Typical of this type of device is the *MaxVideo* range of modules manufactured by Datacube, Inc.

Whatever the hardware, pipelining also creates difficulties where feed-forward predictive techniques are used to reduce the computational load. It is difficult to see how results derived from image *N* could influence the processing of image *N+1*, if that image is already well through the pipeline.

3. Pilot Study

A pilot study (carried out in the Dept. of Computer Science at Sheffield University during 1986-87 with funding from GEC) implemented three of the algorithms at the front end of the *TINA* suite on a tranputer array. These are the *Canny* edge detector, the *PMF* stereo algorithm, and an edge-grouping stage. (These correspond to the 'Canny' 'PMF' and 'Connect' stages in the table of timings shown earlier). The system comprised a network of 8 T414 transputers each with 1 Mbyte of memory, with a Research Machines '*Nimbus*' PC as host. The algorithms were coded in *Occam II*, using the Inmos Transputer Development System (TDS).

3.1 Pilot Implementation of Canny

3.1.1 The *Canny* edge detector algorithm takes a 2-D intensity map as input, and identifies positions ('edgels') at which the intensity gradient is a local maximum. The algorithm has four stages: gaussian convolution, differentiation, non-maximal suppression, and thresholding with hysteresis.

3.1.2 Exploitation of parallelism within Canny

Canny is essentially a pixel based algorithm, which (apart from the thresholding stage) requires only local neighbourhood access to pixels. Therefore, a (1-D) spatial partitioning of the image was adopted. As with all algorithms which require access to pixel neighbourhoods, complications arise at the slice boundaries. In the case of the convolution and non-maximal suppression stages, these problems can be solved by supplying each transputer with input data which includes a few rasters adjacent to the slice for which the processor is responsible for generating output. That is, the input slice is bigger than the output slice.

In the case of the 'thresholding with hysteresis' stage, the problem is less easily solved. This stage is implemented by imposing two thresholds, T_l and T_h, on edge strength. Edges with strength above T_h are unconditionally accepted. Those with strength below T_l are rejected. Then for each marked edgel, a search for a neighbouring edgel with strength between T_l and T_h is made. If one is found, it is marked to be kept, and then in turn its neighbours are examined, and marked. Effectively this is a recursive line following algorithm. Finally, when all high contrast edges have been examined, (and lower contrast segments have been followed and marked), the thresholding stage terminates. All unmarked edges are then discarded. This algorithm allows isolated weak edgels to be discarded whilst allowing weak points in otherwise strong edges to survive.

Because of the slice partitioning of the image, provision must be made to allow the line follower to follow lines across slice boundaries. This is implemented by having each transputer request one raster from the transputers processing the neighbouring slices. The hysteresis procedure operates on the whole of the slice and each of these two rasters. At the end of the iteration, these two rasters (which may have been modified -- i.e. some edgels may have been marked) are returned to the appropriate transputers which compare them with the original rasters sent out. If they detect a new marked edge on this raster, they apply the line follower to it, which will mark the edgels comprising any line segment extending from it into the slice. In turn this may cause the slice edge raster to be modified again (if the line loops back towards the slice boundary), which will precipitate another iteration. These iterations continue until all transputers detect no change to the raster they previously transmitted. Typically the number of iterations will be one or two for nearly all images, unless a line wobbles across a slice boundary several times. In any event, each iteration consumes only a small amount of processing time as the line follower is being applied to only one or two points on the raster.

3.1.3 Pilot study -- Canny timings

Table 2 shows the time in seconds taken to execute Canny on a 256x256 image (similar to the one used for the timings in Table 1) using a network of 2, 4, or 8 transputers, both with and without load balancing. Due to the memory requirements of the program, no slice size adjustment was possible using two transputers, as each had sufficient memory to store only 128 rasters.

Condition	Number of Transputers		
	2	4	8
No load balancing	4.4	2.4	1.5
With load balancing	---	2.1	1.1

Table 2

3.2 Pilot Implementation of PMF

PMF is a stereo matching algorithm. Is takes as input a stereoscopic pair of edge maps, and attempts to match corresponding edgels in the two images. From the measured disparity of each matched pair, and the known camera geometry, a 3-D edge map is generated.

3.2.1 The PMF algorithm

The algorithm is not described in detail here. Essentially it consists of two distinct stages. The first establishes a matching strength for each potential match, computed from some measure of edge quality (for example, contrast strength), of every other potential match in the neighbourhood (typically about 15 pixels in diameter). Each contribution is weighted inversely by its distance away. This is the match generation phase, and is relatively fast, accounting for typically 5% of *PMF*'s total run time.

There follows the disambiguation phase which chooses the correct matches using a form of discrete relaxation. Matches which have the highest matching strength for both of the two image primitives forming them are immediately chosen as correct. To satisfy the uniqueness constraint, all other matches associated with these primitives are then discarded. Other matches that were not previously accepted or rejected are then considered again. Typically four or five iterations of this are needed to provide satisfactory disambiguation.

3.2.2 Exploitation of parallelism within PMF

PMF is essentially a local neighbourhood operation, albeit one in which communication with neighbouring pixels is frequent. The neighbourhood in the match generation phase is inherently anisotropic, as matches are sought only along corresponding horizontal rasters. These considerations again suggest the use of a 1-D spatial partitioning. Once again, problems arise at the slice boundaries.

Two methods of having the transputer acquire neighbouring rasters were investigated. These are the *slice overlap* method, and the *slice communication* method.

The simpler slice overlap approach requires each transputer to apply the match generation phase of *PMF* not only to the rasters within the slice to be processed, but also the *N* (typically 10) neighbouring rasters above and below. Thus, no transputer communication is required as all the data required for the disambiguation phase for the slice is already computed and held in memory. During the disambiguation phase, only the match strengths of edgels within the slice are altered, and matches are rejected or accepted accordingly. The match strengths of the neighbourhood rasters are not altered, nor are they rejected. This is not a true implementation of *PMF*, and it was anticipated that this could cause irregularities in the depth map at the slice boundaries. It transpired that although the disambiguation power was slightly reduced, the depth map was still of good quality, and not greatly different from the original serial implementation. Because there is no communication between the transputers, and hence no synchronisation, the time taken to process the slice varies considerably with complexity.

The slice communication method initially sends each transputer only those rasters within its slice. The match generation phase is then executed for each of the slice rasters. Before and after each iteration of the disambiguation phase, N neighbourhood rasters are copied from the neighbouring transputers. This approach ensures that matching strengths of the slice neighbour rasters are always updated and correct. This method gives a more satisfactory result than the slice overlap method. The timing variations are much less because the raster communication at the start of each iterations effectively forces the transputers to synchronise.

3.2.3 Load Balancing of PMF

A simple 1-D pre-partitioning of the data was performed by allocating roughly equal numbers of edgels to each transputer. This simple algorithm reduces the time taken by the slowest transputer by as much as 50%. A more complete algorithm to pre-partition the data would need to take into account other factors such as the distribution of edgels as well as their numbers, but no simple rule could be found which was effective. Further investigation might yield a more effective measure, but a requirement of the measure is that it is quick to compute, else more time could be lost than gained.

It would in principle be possible to re-partition the image after each iteration in the disambiguation phase. However, this re-distribution of rasters could be so substantial that large pieces of the image would need to be transmitted across several transputers. Instead, it was felt that load balancing would be best performed between images (assuming an image sequence), by controlling the initial distribution of the slices into the transputer array.

In this pilot study, only a single pair of images was available, so a re-partitioning of the *same* image was performed to simulate the effect of pre-partitioning the next image in the sequence.

3.2.4 Pilot Study -- PMF Timings

Table 3 shows timings for the same 256x256 image used in the pilot implementation of *Canny*, using 8 transputers.

It is particularly interesting to note how ineffective the pre-partitioned load balancing is (i.e. based on distributinq equal numbers of edgels to each transputer), as evidenced by the 3-15 second timing spread. The computational load of one raster of *PMF* depends not only on the number of edges but also their distribution. (A large number of edges close together take longer to disambiguate than the same number more evenly spread). This is reflected in the much wider range of slice sizes in the post-partitioned results.

3.3 Fixed versus Floating Point Arithmetic

This pilot study placed some emphasis on the avoidance of floating-point arithmetic by using scaled integer arithmetic wherever fractional accuracy was required. This was sensible in view of the fact that the T414 transputer has no floating point hardware. The T800 transputer being used in the new system can, remarkably, multiply two floating point numbers in less than one third the time taken to multiply two integers. This turns the tables -- it may be that in the future it should be integer arithmetic we should avoid!

	Pre-partitioned data		Post-partitioned data	
	Time	Slice sizes	Time	Slice sizes
Slice Overlap Method:	3-15	36,20,15,14,17,15,16,37	7-9	63,19,7,4,4,11,13,4
Slice Communication Method:	5.1-5.9	36,20,15,14,17,15,16,37	4.3-5.0	43,26,13,6,10,14,16,4

Table 3

3.4 Some comments on Occam II

The study did not find that the *occam* language is any better for program development than other languages with which the authors are familiar, such as *Pascal* and *C*, although the large number of compile time checks can reduce the number of potential runtime bugs. However, *occam* provides tighter control in a parallel processing environment and maps very efficiently onto the transputer instruction set.

Occam II has several shortcomings for the programmer used to procedural languages such as *C* and *Pascal*. Only the most basic data types are available: integers, booleans, and reals of different lengths, together with arrays of these types. Absent from occam are facilities such as pointers, record structures, and enumerated types. The transputer implementation of occam also requires static data allocation -- dynamically allocated structures such as record heaps or linked lists are not available. The lack of dynamic memory allocation further prohibits the use of recursion, requiring the programmer to resort to 'manual' simulations using loops and explicit parameter stacks.

Nonetheless, *occam* does offer considerable advantages to the parallel systems programmer. Networks of parallel processes can be set up very easily, and interconnected by 'occam channels' -- an abstraction of a serial inter-process communication channel which implements a *rendez-vous* between the two processes, and which corresponds directly to the behaviour of a transputer link. Networks of parallel processes can be implemented on a single transputer then later migrated onto an appropriate multiprocessor network by the addition of a very small amount of configuration information, and without changes to the code itself.

4. MARVIN

4.1 Introduction

The work currently in progress as part of the *Fast Vision Project* in *AIVRU* centres around the construction of a hybrid computing engine containing both pipelined frame-rate hardware and an MIMD transputer array, with fast data paths between them. We call this machine *Marvin*.

4.2 Marvin Hardware Architecture

4.2.1 Hardware Components

The hardware architecture of the system is shown in *Fig.*

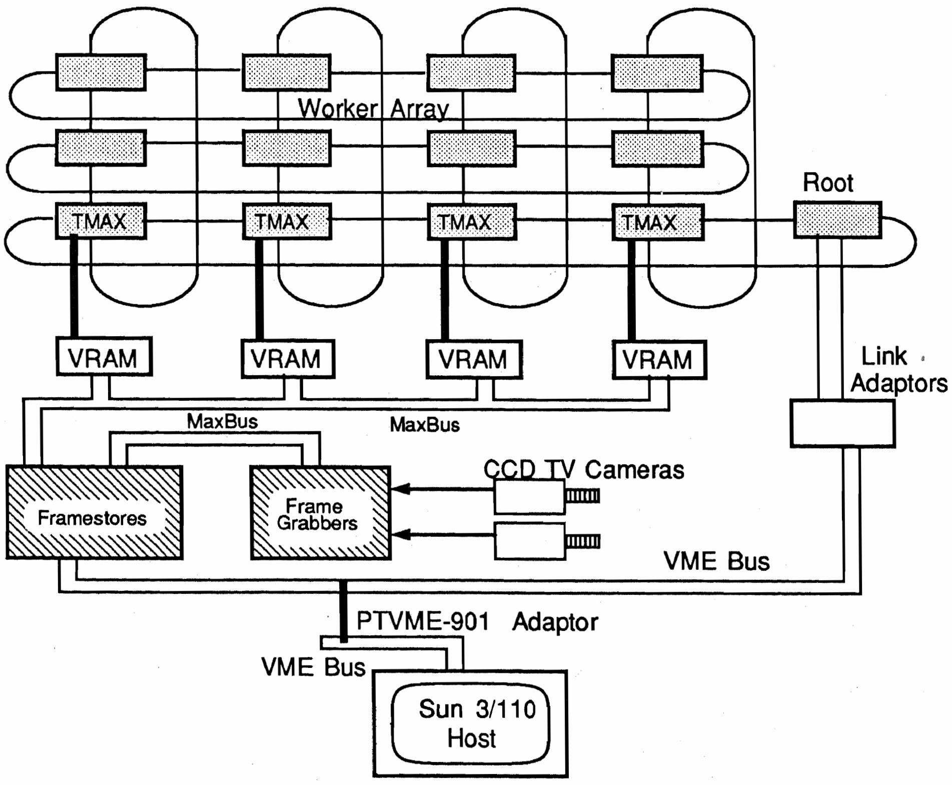

Fig 5. Marvin System Architecture

5. It consists of the following major components:

(1) A *Sun-3* workstation, referred to as the *host* machine.

(2) Datacube framestores and frame grabbers ('Digimax') providing for the acquisition, storage, and display of TV images using conventional interlaced TV timing. Two frame grabbers facilitate simultaneous acquisition of stereo images from two (synchronised) cameras.

(3) A network of transputers, referred to as *worker* processors, connected via Inmos serial links into a rectangular array. (The system will actually have an eight by three array. Only four columns are shown in the figure for simplicity). Each worker consists of a T800 floating point transputer with 2 Mbytes of memory. The workers in the bottom row of the array have an additional 1 Mbyte of video memory which is dual-ported between the transputer and the frame-rate digital video bus (MaxBus). In other respects they are functionally equivalent to the other rows. This bottom row are referred to as *TMAX* (Transputer-MAXbus) processors.

(4) One additional transputer module referred to as the *root* processor. This module is functionally equivalent to the workers and is special only as regards its strategic placement between the worker array and the host machine.

4.2.2 Datapaths

The datapaths within the system are as follows:

(1) Inmos serial links running at 20 Mbits per second provide interconnectivity within the worker array.

(2) The Datacube framestores are fully mapped into the Sun's address space via the VME (A24D16) bus. There are six 512x512 framestores, collectively providing 1.5 Mbytes of image storage.

(3) The framestores are dual ported onto a set of four *MaxBus* interconnects. Each interconnect provides an 8-bit parallel, byte-serial data stream at video frame rate. *MaxBus* is not a bus in the usual sense. It has no address lines, and no general arbitration scheme. Rather, it is a point-to-point data path. The spatial position of a pixel in the image is inferred from its temporal position in the data stream relative to frame sync, line sync, and pixel dot clock signals. The peak data rate (per *MaxBus*) is 10 Mbytes/sec; the average rate (over a 40 msec. frame) is 6.4 Mbytes/sec. The video memories on the *TMAX* processors are also dual-ported onto these *MaxBus* interconnects. Region-of-interest circuitry within this interface allows each *TMAX* to participate in *MaxBus* transfers (both in and out of the *TMAX* memories) within a software-selectable region of interest of the image. (See *Fig.* 6). This allows, for example, a pair of stereo images held in the framestores to be distributed across the row of *TMAX* processors within a single frame time. The architecture is clearly designed to exploit 1-D spatial parallelism.

(4) Two links on the root processor are connected to the Sun host via Inmos link adapters which are mapped into the Sun's VME address space. Each provides a serial byte stream between the Sun and the network of workers. The maximum data transfer rate acheivable here is limited by the rate at which the host 68020 processor can execute the loop which copies the bytes across, and is only about 120Kbyte/sec; however this is not crucial to the performance of the system as this data path is used only for initial downloading of code and for relatively short control messages.

4.2.3 Rationale

Our choice of network topology is made not as the result of any detailed study but simply because it is richly interconnected, regular, and (we believe) 'sensible'. We do not believe that there is some magic topology awaiting discovery which somehow resonates with the problem and offers supra-linear speedup. Of course, some topologies are demonstrably better than others, for specific applications. However, the adoption of a specialised topology is only appropriate if the process structure and patterns of dataflow are already well understood, and unlikely to change. Neither condition holds in our case. Indeed, the scenarios we envisage involve a variety of vision tasks, with different logical structures and operating on different time scales, executing on the

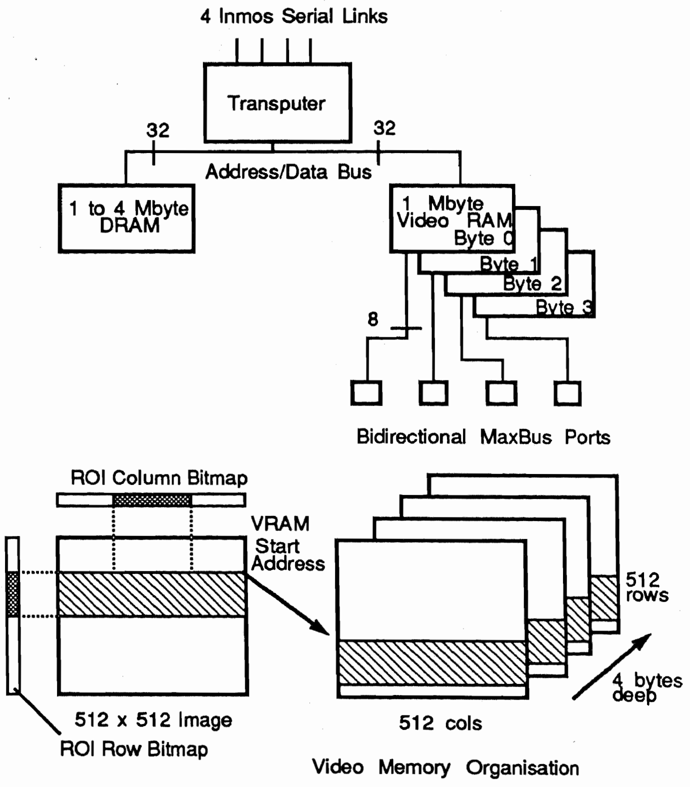

Fig 6. Architecture of TMAX Processor

system simultaneously. Thus, 'sensible' is the most we can expect of the topology -- 'optimum' is not realistic.

4.3 Marvin Software Architecture

4.3.1 Task Organisation

The root and worker processors are programmed in *3L*'s *Parallel C* language. In operation, each transputer is loaded with some configuration of both operating system and vision tasks. Operating system tasks include a router, providing message routing and multiplexing between tasks, and a memory manager task providing memory management facilities.

The vision tasks each implement some vision processing operation. Thus, the collection of vision tasks loaded onto a worker yields a repertoire of vision operations offered by that worker. There is no requirement that each worker carries the same set of tasks. Note, however, that unlike some of the more fully developed run-time environments such as *Helios* or Niche's *PSE*, *Parallel C* does not permit tasks to be dynamically added or deleted from a processor. Changing the task configuration requires that the entire network be rebooted, and this cannot be considered a real-time operation.

The host processor makes certain services available to the root processor, including a filesystem and console I/O, through the services of a 'host file server' program which runs on the host all the time the system is in operation. A program running on the root makes file I/O or console I/O requests to the server by sending messages via the host's link adapter. These messages conform to a tagged protocol developed by Inmos, and known as the *afserver* protocol. Locally developed additions to these protocols allow the root processor to request the server to spawn any unix program, and to obtain connections (via unix pipes) to either or both of that program's standard input and standard output streams.

4.3.2 Message Passing

Messages may be passed between any pair of tasks running on the system, using a store-and-forward mechanism within the router task running on each processor. Each processor is assigned a numerical identifier at the configuration stage, and a message is addressed to a specified task on a specified processor. This address information is included in an 8-byte message header which specifies the destination processor ID, the destination task ID, the source processor and source task IDs, and the message's length. The processors are numbered by simply counting off in 'raster scan' order, with fixed ID numbers assigned to the root transputer and the host. Each router knows the number of the processor on which it is running, and the topology of the network. From this it builds a lookup table which maps the destination process number onto the number of the link (effectively north, south, east, or west) to which the message is to be forwarded. Note that the routing is fixed; there is no attempt (for example) to dynamically balance message traffic within the network. Moreover, each router forwards messages in the order in which they were received. Thus, a sequence of messages from any one source processor to any one destination processor is guaranteed to arrive in the same order as it was sent. When the router receives a message addressed to its own processor, it forwards it via an internal (soft) channel to the appropriate vision task, as selected by the destination task field in the header.

The router has no compiled-in knowledge of which internal channel corresponds to which destination task, or even of how many such channels exist. Instead, each vision task passes a 'registration message' to the router when it starts up. This message includes the task's ID value, and effectively says 'I wish to receive messages addressed to this destination task ID'. A vision task could register more than once, using different identifiers, if desired. This registration scheme allows the set of vision tasks to be changed without recompiling the router.

Note that, with this routing software in place, and hence the ability to send a message from any task to any other, the actual physical topology of the link connections in the network is irrelevant, at least from a functional standpoint. The designer of an algorithm intended to be implemented as a set of communicating sequential processes can take a blank sheet of paper, and draw upon it whatever boxes and whatever interconnection paths he chooses. Of course from a performance standpoint it is rather important that a pair of processes which trade large volumes of data should be placed on nearby (preferably adjacent) processors. In some cases, it may be best to place two processes on the same processor, and have them share memory. 3L Parallel C does permit this. (It would be more honest to say that the transputer, lacking memory management hardware, is powerless to prevent it).

4.4 Marvin's Performance

It is too early to give trustworthy estimates of Marvin's performance as we have only recently begun to build any real vision software on top of the infrastructure described above, and also the full array of transputers is not yet present. We have, however, implemented *Canny* on a three-processor subset of this machine, from which we estimate that the complete system will be able to perform *Canny* on a 512x512 image in 1 second.

To improve further on these speeds we expect to have to exploit Marvin's potential as a hybrid system in which low-level processing is performed using frame-rate hardware interposed in the *MaxBus* datapaths. The frame-rate canny hardware developed at GEC (See "*A Pipelined Architecture For the Canny Edge Detector*" by Brendan Ruff) is the most obvious candidate for inclusion in this way.