A Generalised Parallel Architecture for Image Based Algorithms

G. J. Vaudin, G. R. Nudd, T. J. Atherton, S. C. Clippingdale, N. D. Francis, R. M. Howarth, D. J. Kerbyson, R. A. Packwood and D. Walton

Department of Computer Science
University of Warwick
Coventry, UK

1. Introduction

Real time image generation and image understanding require levels of computing power, that are beyond that available from conventional sequential machines. Current commercially available systems aimed at this area make use of special purpose hardware to achieve the necessary throughput, but these systems can only achieve their performance for a restricted set of algorithms that are implemented in the hardware. A programmable general purpose parallel machine offers the possibility to achieve the required performance without restricting the choice of algorithm.

Unfortunately it is by no means clear which parallel architecture should be used. Many general purpose parallel architectures have been proposed but none has proved universally applicable, their problem being that their performance tends to be highly dependent on the algorithms that are being used, and it is therefore difficult to claim any of them are truly general purpose. However parallel machines can still be highly effective in specific problem areas where the class of algorithm is known.

Our aim has been to design a parallel machine that is optimised for image based algorithms in both graphics and image understanding. The architecture is not limited to a specific set of algorithms, but is instead optimised towards a class of algorithms which we believe are representative of image based algorithms. This has not been a paper study, but has resulted in us implementing such an architecture. We have achieved this by making use of industry standard components and integrating them into a system level architectural design. Also we have where possible used industry standard programming languages to program our machine.

2. Existing Parallel Architectures

Various classifications exist for parallel machines. The most fundamental is Flynn's [5] which classifies machines by the number of instruction streams that they employ. Single instruction / multiple data (SIMD) machines have a single instruction stream which is common to all the processors in the system such that each instruction operates on many different data in parallel. Multiple instruction / multiple data (MIMD) machines have a separate instruction stream for each processor so that many different data may be acted on independently from one another.

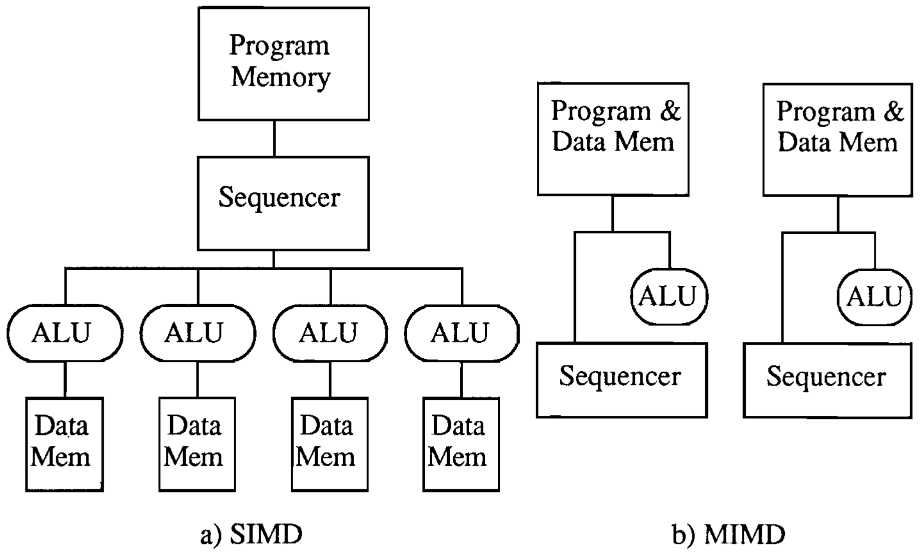

Figure 1: SIMD and MIMD Architecures

SIMD machines [1,2,3,4] excel at problems where large arrays of data are to be processed identically. They are also very good at communication intensive problems, provided that the communication is also identical for each processor. Since SIMD machines have only a single instruction stream they require only one program memory, and one instruction sequencer. This means that more of the machine can be given over to computational units and so makes them much more cost effective than MIMD architectures if the data is to processed identically. However SIMD machines become progressively more inefficient the more the processing begins to vary from one processor to another. Most SIMD architectures rely on being able to exclude certain processors from a particular piece of computation to implement different processing on different processors. This leads to processors lying idle for some of the time, and the more the processing varies the more time processors spend idle. The limiting case being when each processor is executing a completely separate instruction stream at which point the performance degrades to that of a sequential machine.

Conversely MIMD machines [13] excels at problems where each processor is completely independent of each other. Since it has instruction memory and instruction sequencing on every processor there is no overhead involved in every processor executing a different instruction. However because the processors have their own instruction streams there is no global synchronisation between processors, which means that communication between processors involves some form of rendezvous, which implies that one or other processor has to wait for the other. This leads to an increased communications overhead compared with the SIMD machines. Also it is characteristic of algorithms that run well on MIMD machines that because of the independent nature of the processing on each processor the communication tends to be less localised than with algorithms that run well on SIMD processors. This further increases the communications overhead, and makes MIMD machines less suitable for communication intensive problems.

In addition to the MIMD/SIMD classification parallel machines can also be classified into coarse and fine grained. Coarse grained implies that the machine is made up from relatively large processors, usually with floating point hardware, whereas fine grain implies the use of much smaller processors many of which can be integrated onto a single chip. Coarse grained machines are generally good at general purpose "scientific" computations which make heavy use of floating point calculations, and which therefore benefit from having these calculations done in hardware. Fine grained machines are generally better at tasks which involve fairly simple integer arithmetic often involving variable word lengths. For these tasks the use of a large arithmetic unit would be very inefficient since most of it would lie idle, a better use of the available resources is to implement more of the simpler processors.

In general most SIMD machines tend to be fine grained, whereas most MIMD machines tend to be coarse grained. It would be very inefficient to implement a fine grained MIMD machine since the overhead of providing an instruction sequencer for each processor would begin to dominate the system. There is no such restrictions with SIMD machines so it is reasonable to build systems with large numbers of small processors. It has been argued that it is in general more efficient to use a large number of simple processors than a smaller number of more complex ones [14], but as I have already mentioned this is in fact only true for a restricted class of problems. There may therefore be applications for which a coarse grained SIMD machine would be ideal, but at this time most available SIMD machines are fine grained.

3. Image Based Algorithms

To decide which of these architectures would be most suitable for the use in image based applications we need to look at the structure and composition of the problems and the kind of algorithms that are used in them.

3.1 Image Understanding

An image understanding system is required to take an image (typically consisting of an array of intensity values) as input and produce a high-level abstract description of the scene as output, in real time. To do this it would proceed in a number of stages, at each one refining the data into a more abstract form.

Initially low-level or iconic-to-iconic processing would be performed, which processes the input image and produces a modified form of the image as output (this is the image processing part of the problem). The result of this stage of processing will be a segmented and labelled image, where the various areas of interest within the image would have been isolated and uniquely labelled. These areas may be edges, enclosed regions or other more specific features.

The next level of processing consists of iconic-to-numeric processing where quantitative information about the isolated regions is extracted from the image. This information might include the centroid, area and variance of a particular region for example. The information required would depend on the type of feature being examined.

Finally numeric-to-symbolic processing is performed which takes this extracted data and uses it to form hypotheses about the scene represented by the image. This would involve using a priori knowledge of the possible contents of the scene to identify the relevant features and then match them with hypothesised objects. The system can then perform reasoning based on the interrelationship of these hypothesised objects in order to produce a full description of the scene.

One of the difficulties in building a machine to do this processing is that a wide variety of operations need to be performed, on widely differing representations of the data. An architecture which is good at one type of operation (eg. pixel-based iconic-to-iconic) is likely to be unsuited to other types (eg. symbolic).

For the pixel or iconic representation, and the corresponding class of operations such as thresholding, convolutions, and histogramming that one wishes to perform on this data a very high throughput is necessary to process the large amount of data. About 10 million pixels need to be processed per second for real time applications, and the throughput is particularly large when one considers that even for a simple linear operation such as a 5x5 convolution 25 multiplications and accumulates are required per pixel. For non-linear operations such as median filtering the throughput grows as N^2, and of the order of 1,000 MOPS may be needed.

To meet this throughput requirement requires a very high degree of parallelism, ideally one processor for every pixel in the image. However it should be noted that the operations are carried out globally on all pixels simultaneously and are therefore suitable for implementation on an SIMD machine. Also the operations tend to be numerically simple, with 8-bit integer multiply probably the most complex operation required. Thus an array of simple processors could be used which would be sufficiently small as to make such a massively parallel scheme feasible. Many such machines, mostly consisting of simple bit-serial broadcast SIMD array processors, have been built (CLIP, DAP, MPP [1,2,3,4] etc.) and have proved very able at this low level image processing type of problem.

The iconic-to-numeric stage, is responsible for analysing the results produced by the iconic-to-iconic phase. It needs the ability to extract results from the pixel array, allowing for the fact that the results required may vary depending on the feature that is being processed, and to process these and pass them on to the symbolic layer. The data extracted will be more complex in nature than the simple intensity values of the iconic stage, but will be less numerous, and the processing performed on it will typically be more varied than is the case with iconic data, since the data itself is more varied.

To reflect these needs a suitable processor would be a medium grain MIMD machine which would be able to influence the processing of the iconic data which it is to use as input. A conventional microprocessor would be a possible candidate for this purpose, with some additional hardware to allow it to communicate with and have some control over the iconic processors from which it receives its input.

The symbolic processing requirements are more varied in nature. By this stage any direct relationship with the image data is gone, and the processing deals with more abstract notions of shape and relationship. These structures will be be much more complex than in previous stages but will be correspondingly less numerous. For this purpose a powerful general purpose MIMD parallel machine would appear to provide the appropriate choice.

The conclusion to be drawn from this is that no one architecture will be suitable for all stages in the image understanding problem. The ideal image understanding architecture would consist of a variety of different processors arranged in such a way as to allow data to flow between the different sections.

3.2 Computer Graphics

Computer graphics can be regarded as the reverse process to image understanding, instead of starting with an image and producing a model, one starts with a model and produce an image which represents that model. Like image understanding, computer graphics systems proceed in a number of stages, in this case moving from an abstract representation of a scene towards a concrete pixel array representation.

The first step in processing consists of viewing transformations, where the model is transformed into the coordinate system of the viewer. This may involve different transformations for different objects in the scene, if the relationship between the various objects is not fixed. Working out these relationships may be a complex problem in itself if, for example, the different object were aircraft in a flight simulator. Then the illumination of the objects is calculated based on their positions with respect to the light sources. Most systems will perform some clipping at this stage, where objects out of view are removed from further calculations.

Next the objects are decomposed into their constituent parts, usually planar polygonal patches, and these are then projected onto the two dimensional screen coordinate system. Further clipping is carried out to remove parts of objects which fall outside the boundaries of the screen, and surfaces which are obscured by other objects may be removed. Finally the pixels which represent the 2D polygonal patches are evaluated, and each set to the appropriate intensity level. This stage may attempt anti-aliasing, and smoothing and may also be responsible for hidden surface elimination.

The first stage in the process involves a large number of floating point operations to be performed on the objects, which have no fixed interrelationship, and may have different representations and thus may require different processing. These operations can in general be performed entirely independently of each other, so that inter processor communications is not generally a problem. An MIMD array of floating point processors would appear to provide an appropriate solution to this stage.

Once the data has been converted to screen polygons the processing becomes arithmetically simpler, usually involving integer arithmetic only, and also tied more closely to the physical layout of the screen memory. This kind of problem is amenable to solution by an SIMD array of integer only processors. However there will typically be a many to one relationship between polygons and the pixels which make up the polygons, which implies that there will be fewer polygon processors than pixel processor, which in turn implies that a more coarse grained processor is used.

The final pixel tiling involves a very large number of simple operations to be performed identically over the pixels forming a given polygon, and so a large SIMD array of simple processors would be appropriate.

Similarly to image understanding it can be argued that no one of the available architectures is ideally suited to all the stages in the graphics process. What is required is a machine which consists of a variety of different processors arranged in such a way that data can be efficiently transferred between them.

We have designed and constructed such an architecture which provides four distinct "layers" of processors. These processors can communicate both within one layer or between adjacent layers. In this way data can be passed from one stage to the next. In the following sections we will describe in detail the design and implementation of this architecture, and then go on to describe the programming of it.

4. Warwick Pyramid Machine

The Warwick Pyramid Machine [6], has been developed to address the needs of a specific class of image based problems, namely image understanding and computer graphics. It consists of four distinct layers of processors, each optimised for one stage in the problem. Since data flows both within each layer and between adjacent layers, it is convenient to regard the layers as being stacked one above the other. Also as each layer contains fewer processors (but more powerful ones) than the layer below it, we regard the machine as being a pyramid of processors.

At the base of the pyramid is the iconic layer, which is a fine grained Multi-SIMD machine. This consists of an array of independently controlled SIMD machines called clusters. Each cluster contains its own controller which provides it with its instruction stream. These controllers are each associated with a symbolic processor which processes the results from the cluster and also determines the instructions generated by the controller. The symbolic layer is an array of coarse grained MIMD processors. The symbolic layer is connected to a conventional host machine which is used to interact with the pyramid.

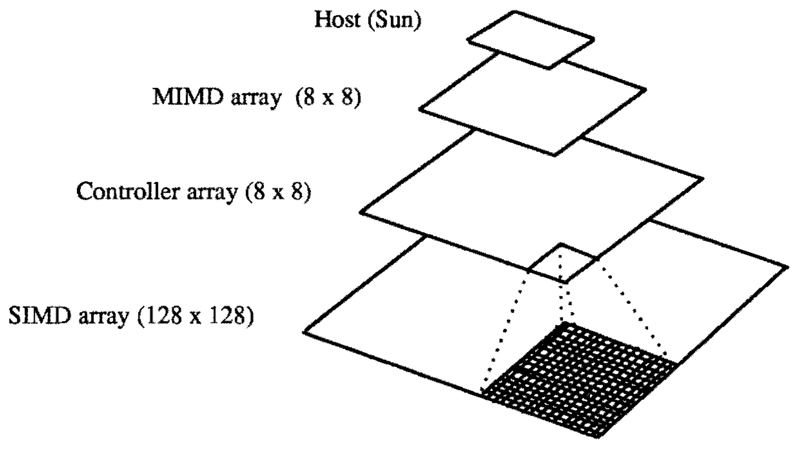

Figure 2: The Warwick Pyramid Machine

4.1 Host Machine

At the apex of the pyramid is the host computer, which performs no computations of its own but provides the user interface for the pyramid below it. The host machine provides mass storage and is responsible for downloading programs into each processor. It also provides a familiar user environment, such as editors and debuggers for developing programs to be run on the lower levels, displaying intermediate results, and so on. A graphics workstation such as a SUN is appropriate for the host, since it combines a powerful development environment with a suitable graphics capability to display results.

We have been using a SUN 3/280 server and SUN 3/60 workstation as the host machine for our prototype machine The SUN 3/280 contains a Transputer based interface card which communicates via a fibre optic cable to the symbolic Transputer layer.

4.2 Symbolic Processing Layer

The symbolic layer provides the top level of processing, dealing with abstract model data, and is an MIMD array of coarse grained processors. The processor we use for this is the INMOS Transputer, which is specifically designed with MIMD multiprocessing in mind. It includes hardware support for inter-processor communications in the form of four DMA driven serial links which can communicate at up to 20Mbits per seconds each without processor intervention. It also provides micro-coded support for multi-tasking which allows very fast context switching, and transparently handles interaction with the on board DMA controlled links.

The Transputer supports a model of concurrency known as Communicating Sequential Processes or CSP defined by Hoare [7]. This models breaks problems up into a set of independent sequential processes which communicate and synchronise only by means of channels. The channel is a unidirectional data stream which causes a rendezvous for every data item passed through it. Thus if a process writes to channel it is suspended until another process reads the channel. Conversely if a process reads from a channel it will be suspended until data has been written into the channel by another process.

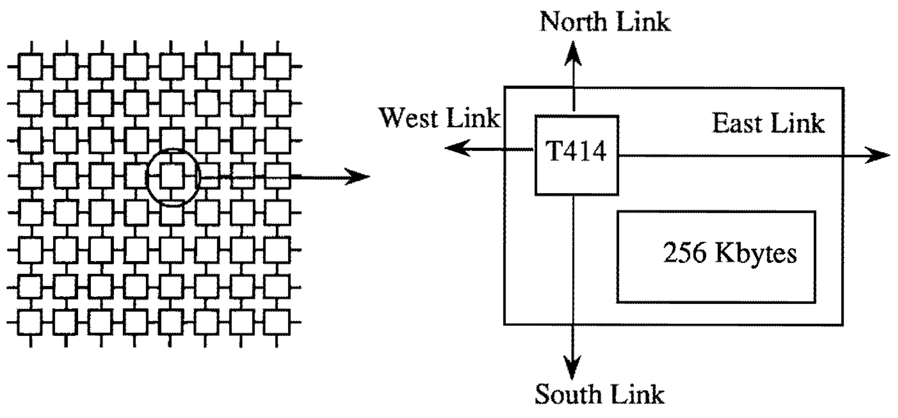

Figure 3: Symbolic Processor Array

The Transputer has instructions which directly implement these channel semantics, regardless of whether the channel is between two processes on a single Transputer or over a link between two processes on two different Transputers. Using this mechanism it is possible to write programs that are independent of the number of Transputers in the system provided that there are at least as many processes in the system as there are processors.

The Transputer is available both with and without hardware floating point support. Ideally we would like to have used the T800 floating point Transputer, but because of financial constraints our prototype hardware uses T414 non-floating point device. Our symbolic processing layer consists of an 8 x 8 array of N-S-E-W connected T414 Transputers each with 256Kbytes of RAM. We also have a Transputer based frame-buffer used for displaying processed images. This layer has been designed and constructed by us, and has proved a useful vehicle for software development, as well as providing the first stage in our prototype pyramid machine.

Each Transputer will communicate with the Cluster Controller below it by means of shared memory. In our prototype hardware this is 2Kbytes of dual port static RAM which can be accessed both by the Transputer and the cluster controller. This memory is used by the symbolic processor to send commands to the cluster controller, as well as being used for the bidirectional transmission of data between the symbolic processor and cluster controller.

4.3 Controller Array

The controller array is an 8 x 8 array of cluster controllers . Each cluster controller is responsible for generating the instruction stream for one cluster, which consists of the cluster controller and a 16 x 16 region of the iconic layer. Each cluster is equivalent to a conventional SIMD machine, and it is this that gives the machine its Multi-SIMD capabilities. The cluster controller is also responsible communicating data to and from the iconic layer.

The controller consists of a micro-instruction sequencer which fetches one micro-instruction at a time and broadcasts it to all of the processing elements (PE's) within the cluster. Each instruction specifies the operation that the PE's ALU is to perform, and also the source and destination of its operands. These can be either internal registers or locations in iconic layer memory. Since all the PE's operate on the same location in memory at the same time it can be thought of as acting on a plane of data.

The operation for the PE ALU is specified in the cluster controller micro-instruction, but the source and destination for the operands are generated dynamically which allows the same PE ALU operation to be performed on several different locations in memory. This is particularly useful for implementing multi-bit operations.To allow this the cluster controller includes a scaler ALU for address calculations. This ALU is also used to generate the conditions needed to control the flow of the micro-instruction sequencer.

To achieve maximum performance from the iconic layer it is necessary to provide one micro-instruction on every clock cycle. This proves very hard to achieve since for each micro-instruction several scaler operations may be required to calculate the operand addresses. However we have managed to approach this level of performance by allowing the address calculations to proceed in parallel with the execution of iconic layer PE instructions.

The cluster controller is also responsible for communication of data between the symbolic layer and the iconic layer. Information is passed between the controller and the symbolic processors using the dual ported memory, and between the iconic layer and the controller via a set of hardware registers that will be described later.

A functional diagram of the cluster controller element is shown in Fig. 4. It consists of a micro-programmable 16 bit processor consisting of an AMD 29116 ALU and an AMD 29331 Sequencer, with 16K x 68 bit micro-code store and 2K 16 bit dual ported data memory. These are connected together via a single 16 bit data path. The dual ported memory is shared between the cluster controller and the symbolic processor and is used for communication between the two. At boot time the host processor loads the cluster controller micro-code store with a standard set of micro-routines, which can later be called up through a software protocol using the dual ported memory.

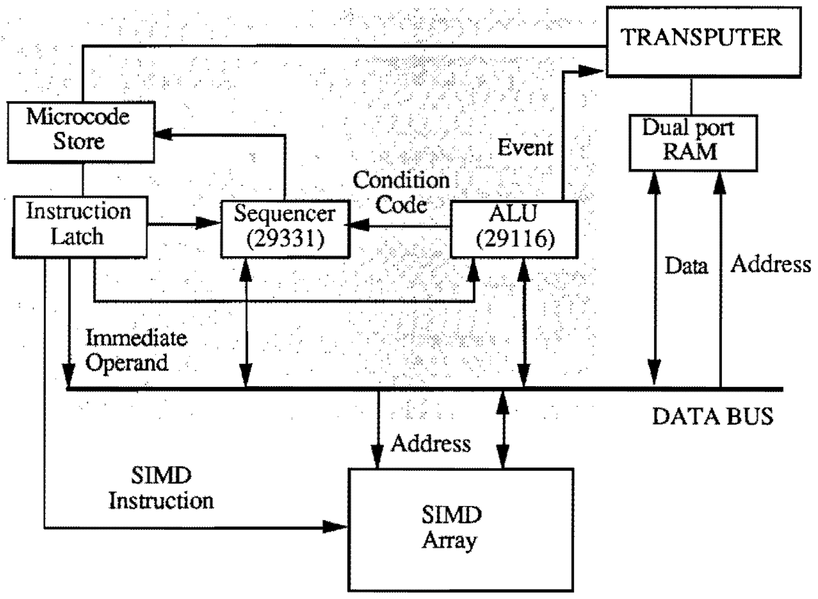

Figure 4: Cluster Controller

The cluster controller's micro-code word is divided into four fields one each for the ALU, sequencer, data path and the iconic processor array. Thus on each cycle the cluster controller can control all four elements in parallel. In this way it is possible to overlap address calculations performed by the 16 bit ALU and pixel instructions performed by the iconic layer, and so maintain the maximum possible iconic level instruction throughput.

The cluster controllers are capable of Multi-SIMD operation with each cluster operating independently from each other. However it is also possible to operate several clusters as a single entity if the iconic processors in adjacent clusters must be synchronised in order that communication across the cluster boundaries performs as expected. This synchronisation is performed by the cluster controller level, using inter-cluster controller communication links to rendezvous synchronised instructions.

4.4 Iconic Layer

The iconic layer performs the low level pixel processing, such as intensity interpolation, hidden surface elimination, convolutions etc. and consists of an SIMD array of bit-serial processors. The total array size is 128x128 processors, but this is broken up into an 8x8 array of 16x16 regions, each one controlled by its own controller. Each 16x16 region and its controller form one cluster. All the processors in one cluster execute the same instruction stream, the only exception being that some processors may elect to remain idle instead of executing a certain instruction. The processors are arranged in a N-S-E-W connected square array and can communicate synchronously over this network.

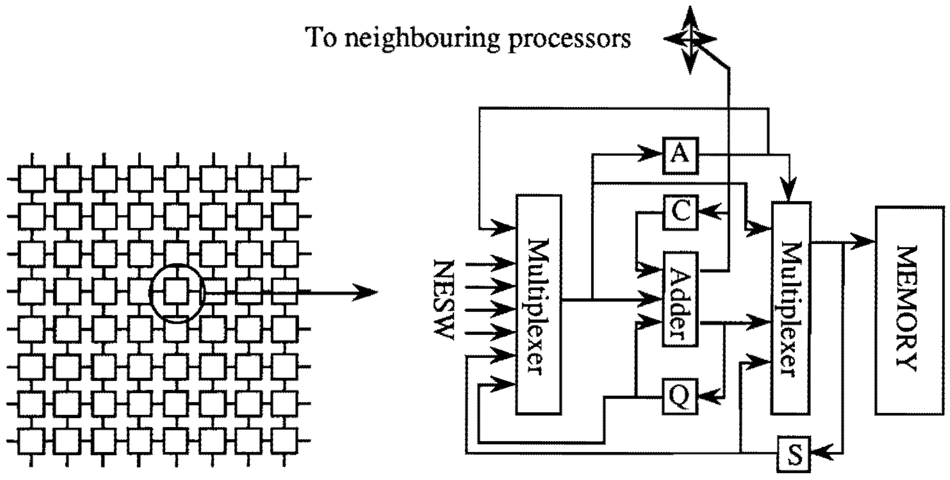

Figure 5: DAP Based SIMD Array

The processors used for this layer are the AMT Distributed Array Processors. The AMT DAP chip integrates 64 bit serial processors onto a single chip. Each processing element consists of a one bit ALU, four one bit registers, and some private external memory. The ALU provides a variety of one bit arithmetic and logical operations with add with carry being the most complex. Since the instruction stream is provided externally there are no control instructions in the DAP instruction set. Most instructions are arithmetic or logic operations, with a small number of communication instructions. Each processor can perform operations on registers and operations which require only a single memory reference in a single clock cycle. Operations which require two memory references take two cycles.

In addition to the processing array each chip incorporates an edge register, which can be used to access a whole row or column of processors in one go. This edge register is mapped into the register space of the cluster controller, and can be used under software control to transfer data to and from the DAP processing elements. Our design also features a mechanism for counting the number of processing elements that respond to a certain key, to allow us to implement associative algorithms efficiently.

In our prototype machine each 16 x 16 cluster is implemented using four DAP chips. Each processing element has 32Kbits of fast static memory making a total of 1Mbyte of memory per cluster.

4.5 Multiple-SIMD Design

A particularly important aspect of our architecture is its Multi-SIMD organisation, where the SIMD array is divided into independently controllable clusters. Other machines have been suggested that combine MIMD and SIMD parallelism [9] but they suffer from two major flaws which the Multi-SIMD approach overcomes.

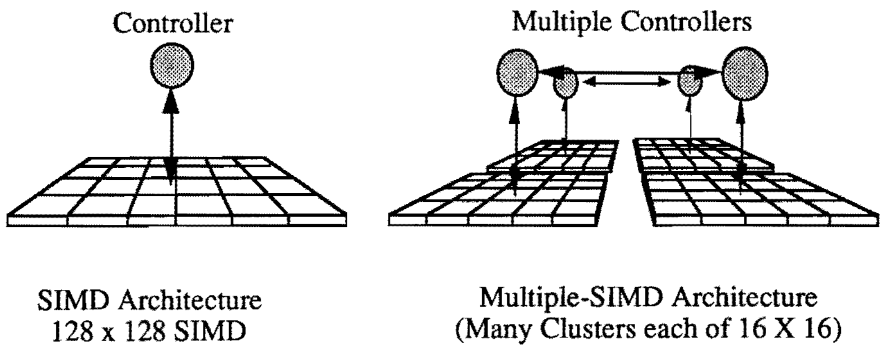

Figure 6: SIMD vs. Multi-SIMD

For a combined MIMD/SIMD machine to work effectively it must be able to pass data between its two halves quickly, which implies a high bandwidth connection. In a conventional SIMD machine the single controller becomes a bottleneck for communication and limits the bandwidth. In our design multiple controllers allow a far greater vertical bandwidth than would otherwise be possible. This approach also scales more readily since the number of controllers increases as the size of the array increases.

The other main advantage of a Multi-SIMD design is increased local autonomy. In a conventional SIMD machine if the processing is concentrated in a small area of the image the rest of the array has to lie idle. If the 128 x 128 array of processors were rendering a polygon, which probably only covers in the order of 100 pixels then only 1 % of the array would be active, or put another way the array would be operating at only 1% efficiency. With a Multi-SIMD design it is possible for each cluster to work on a different area of the image independently of all the others. Thus, taking the previous example, since the area of one cluster is of a similar order to the polygon, we might expect each cluster to be able to work on a different polygon, giving a factor of 64 (the number of clusters) improvement in performance. In practice of course this will not be fully realised but nonetheless a significant improvement is to be expected.

4.6 Current Status

We have built and tested a prototype machine. This consists of a full 8 x 8 array of T414 Transputers each with 256Kbytes of memory on multi-layer printed circuit boards. Connected to this is a single cluster. The cluster consists of a cluster controller which is a single wire wrapped board, and a 16 x 16 array of DAP processing elements implemented as four DAP chips with 1 Mbyte of fast external memory, implemented as two multi-layer printed circuit boards. The Transputer array runs at 20MHz while the cluster runs at 10MHz.

5. Programming

The pyramid architecture consists of three different processor types (Transputers, bit-slice, DAP PE's), each of would normally be programmed its own language. Ultimately it is hoped that a user will see only one language, namely Parallel C++, but at the moment two main languages are used, Parallel C and cluster assembly language.We will now describe these languages and then go on to discuss our future plans to integrate them.

5.1 Transputer Parallel C

This is a commercial product and so we will touch on it only briefly. It provides an ANSI C compiler and a set of function calls to implement the OCCAM style of concurrency, namely communicating sequential processes or CSP. This includes the ability to create processes and to communicate over channels. Processes and channels can be created dynamically and the language allows recursion.

Parallel C is used to program the symbolic processors. Typically the programs on the symbolic processor will coordinate the operation of their associated cluster. We have written a library which provides an interface to a set of routines written in cluster assembly language which run on the cluster controller. These routines perform a variety of functions from low level operations such as multi-bit arithmetic to higher level functions such as scan converting of a polygon. The C functions invoke a remote-procedure call mechanism which makes use the dual-ported memory to write into the cluster controller's instruction queue. The cluster controller takes instructions from this queue and executes the appropriate micro-code routine. Once it has completed this is writes the results into dual-ported memory and generates an event on the symbolic processor. The Transputer's scheduler receives the event and can then restart the process. This mechanism allows the symbolic processor to do other computations while the cluster controller is busy.

5.2 Cluster Assembly Language

The cluster assembly language provides a low level interface to the devices which make up the cluster, namely the scaler ALU, the micro-instruction sequencer, the data path and the DAP array . The cluster assembly language is effectively two assembly languages in one, namely AMD 29116/29336 bit-slice assembler and DAP assembler. Where possible the same mnemonics have been used in the assembler as used by the manufactures of the devices. This does lead to a somewhat irregular syntax, but it means that the manufacturers documentation still makes sense in the context of our machine.

The cluster controller design allows each micro-instruction to fully specify the operations of all four functional units, and the cluster assembly language reflects this by using four fields in each instruction, one for each functional unit, so that each line of assembly language corresponds to one 68-bit micro-instruction.

The use of a wide instruction word, with its inherent parallelism provides a worthwhile performance increase, but it does make programming the machine quite difficult. It is up to the programmer to work out which instructions can safely be carried out in parallel, and since they are all highly interdependent this is no easy task. We intend to automate this process in the future by providing a high level language compiler for the cluster controller but for the time being it is strictly a real programmers machine !

5.3 Parallel C++

We would ideally like to completely integrate the programming of our machine by providing a single language which can express both SIMD and MIMD models of concurrency. It is our hope that C++ will allow us to achieve this. Object oriented programming, which breaks programs into self contained units of data and program, has already been used effectively as a parallel programming technique for MIMD machines [10], and it is our feeling that it could be used equally well to implement the model of SIMD parallelism used in DAP FORTRAN. We intend to use the object oriented facilities in C++ to generate a suitable class library which will include both MIMD and SIMD objects, and thus provide the integrated programming environment we need.

6. Polygonal Mesh Rendering

Having described the architecture and its programming we will now look at mapping a specific graphics application onto it, namely the standard polygon mesh rendering pipeline.

A polygonal mesh rendering systems takes a model, defined by planar polygonal patches, and processes it in stages to produce an image of the model. The implementation has two main objectives. The first is to map each stage in the process onto the processor which can most efficiently implement it. To do this we need to look at the type of calculations that each stage performs, and also the structure of the communication at each stage. In addition to this we would like to use algorithms at each stage that are arbitrarily scalable. This means that as the number of processors increase so does the throughput of each stage.

The rendering is carried out in a number of separate stages, which can be arranged in a pipeline, and indeed this is generally how dedicated hardware implementations are arranged [12]. The early stages of the pipeline, which consist of database control, 3D transformations, clipping and shading calculations, are mapped onto the symbolic processor, since they involve complex operations, usually floating point, performed on relatively few data items. The later stages which consist of scan conversion, hidden surface elimination and intensity interpolation, are mapped onto the low level clusters, since they involve much simpler integer calculation which can be applied to all the pixels in each polygon simultaneously.

Previous attempts at mapping the rendering pipeline onto arrays of MIMD processors have involved using one processor for each stage in the pipe, emulating the approach used by custom hardware [8]. This approach has three disadvantages, first it does not scale readily, which is one of our primary aims, since the number of pipeline stages is fixed to the number of stages in the rendering process. Secondly this approach involves the polygon data being transmitted between each stage in the pipeline, which can be done in a hardware pipeline with no overhead, but in an MIMD machine where communications are a significant overhead this is a problem. Thirdly the pattern of communication implies that the processors be arranged as a pipeline, which does not correspond to our architecture.

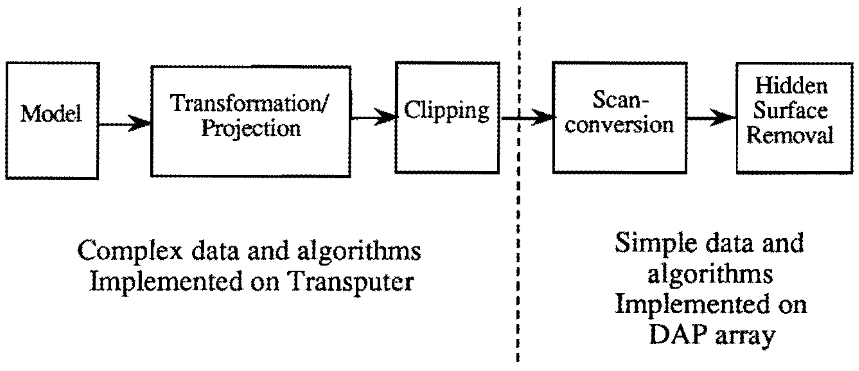
Model
Transformation/ Projection
Clipping
Scan-conversion
Hidden Surface Removal
Complex data and algorithms Implemented on Transputer
Simple data and algorithms Implemented on DAP array

Figure 7: Polygon Rendereing Pipeline

The approach we have taken is to use a static 'processor farm'. This involves distributing the polygon data base evenly amongst the available Transputers, such that each processor has an equal number of polygons. The viewing parameters and lighting details are then broadcast to the processors, which calculate all the transformations and shading for their set of polygons. The polygons are then clipped to cluster boundaries, which may involve polygons which span more than one cluster being broken into several pieces. These clipped polygon are then sent to the symbolic processor associated with the clusters which contains them. This processor then passes them onto the cluster controller for rendering. We have found that this technique provides nearly linear speed up with number of processors, and can be extended indefinitely, up to one processor per polygon. It is also topology independent.

This method relies on being able to send polygon data from one processor to any other in the array, so that clipped polygons can be sent to the appropriate symbolic processor for rendering. To allow this we have implemented a message based router, which can route addressed packets from one processor to any other. The existence of this router has other advantages, in that it allows any node to print data on the host machine, which is extremely useful for debugging purposes.

Once the clipped polygons arrive at the appropriate cluster controller they must then be scan converted. Clearly we wish to render all the pixels in each polygon in parallel, and so the conventional incremental algorithms are not suitable. Instead we use an algorithm developed by Fuchs [11] for his Pixel Planes machine.

The algorithm proceeds in a number of stages, taking one edge of the polygon to be rendered at a time. For each edge the pixel processors in the patch calculate the equation Ax + By + C, where x and y are the coordinates of the processor within the patch, and A, B and C are the coefficients of the line Ax + By + C = 0 which corresponds to the edge. The coefficients are calculated such that pixels outside the polygon return a consistent positive or negative result, so that all the pixels which do not fall inside the polygon can be eliminated. If this process is repeated for all the edges, those pixel which have not been eliminated must be inside the polygon.

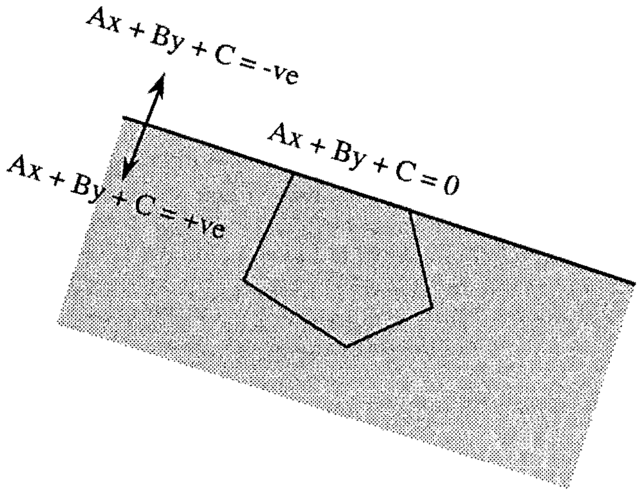

Figure 8: Interior Point Determination

Once the pixels within the polygon have been determined, the next step is to determine the distance of each pixel from the viewer, so that hidden surface elimination can be performed. To do this the processors evaluate the same Ax + By + C equation, but this time the coefficients are of the plane Ax + By + C = z, where z is the depth of the polygon from the view at the point x,y. Each processor then compares this value with any previously stored values for polygons already plotted in that pixel, and determines whether the new value is closer to the viewer than previous values. Finally all those pixels which are inside the polygon and closer to the viewer than previous polygons are set to the colour value for that polygon. This value may itself be calculated using a linear equation to achieve Gourard shading.

6.1 Performance

The Transputer array is expected to be able to process 400,000 triangles per second, assuming that T800 Transputers are used. A single cluster can render approximately 20,000 triangles per second. A full sized SIMD array could process up to 1.2 million triangles per second, but this assumes that all clusters are fully active the whole time which is optimistic. However we would hope that it would able to render at the a rate similar to that produced by the Transputer Array, giving a total system performance of 400,000 triangles per second.

7. Conclusions

We believe that a programmable parallel machine can provide the level of performance normally only available with custom hardware while not restricting the user to one specific set of algorithms. However this relies on achieving a good match between hardware and software, which can only be achieved by targeting the machine at particular problem area where all the algorithms share a similar problem structure. We have designed a machine which is oriented towards image based problems, such as image understanding and computer graphics. These problems are generally difficult to map onto existing parallel architectures because they contain many data structures which map most effectively on to widely different machines. Our architecture overcomes this by incorporating both coarse grained MIMD and fine grained SIMD processors.

Acknowledgements

This work is supported in part by contract N00014-87-G-0241 from the U.S. Innovative Science & Technology Office (addministered by Dr K.Bromley, Office of Naval Research).

References

1. K.E.Batcher, "Design of the Massively Parallel Processor", *IEEE Trans.* **C-29**, pp 836-840,1980

2. M.J.B.Duff, T.J.Fountain (Editors), "Cellular Logic Image Processing", Academic Press, New York, 1986

3. D.J.Hunt, "AMT DAP - a processor array in a workstation environment", *Computer Systems Science and Engineering,* 4 (2), pp107-114, April 1989

4. T.J.Fountain, "A Review of SIMD Architectures", in *Image Processing System Architectures*, ed J.Kitler.

5. M.J.Flynn, "Very High Speed Computing Systems", *Proc IEEE,* **V54** No. 12, 1966.

6. G.R.Nudd et al," WPM: A Multiple -SIMD Architecture for Image Processing", *IEE Conference on Image Processing and Applications,* Warwick, July 1989.

7. C.A.R.Hoare, "Communicating Sequential Processes", *Prentice Hall,* 1985.

8. T.A.Theoharis, "Exploiting parallelism in the Graphics Pipeline", *Technical Monograph PRG-54,* University of Oxford.

9. I.Page, "DisArray: A 16x16 RasterOp Processor", *Eurographics '83.*

10. W.Bronnenberg, "POOL and DOOM", *PARLE '89,* Lectures Notes in Computer Science, Springer-Verlag

11. H.Fuchs, "PIXEL-PLANES: A VLSI-oriented design for a raster graphics engine", *VLSI Design* **VII**, no.3.

12. J.Clark, "The Geometry Engine: a VLSI geometry system for graphics". *Computer Graphics (SIGGRAPH '82 Proceedings),* **vol 16**, no3, pp127-133.

13. E.F.Gehringer, "A Survey of Commercial Parallel Processors", *Computer Architecture News,* **Vol16**,. no4, 1988.

14. W.D.Hillis, "The Connection Machine", MIT Press, 1985.