A Unified Programming System for a Multi-Paradigm Parallel Architecture

Ph.D. Thesis

August 1991

Computer Science Department
University of Warwick

John Vaudin

ABSTRACT

Real time image understanding and image generation require very large amounts of computing power. A possible way to meet these requirements is to make use of the power available from parallel computing systems. However parallel machines exhibit performance which is highly dependent on the algorithms being executed. Both image understanding and image generation involve the use of a wide variety of algorithms. A parallel machine suited to some of these algorithms may be unsuited to others.

This thesis describes a novel heterogeneous parallel architecture optimised for image based applications. It achieves its performance by combining two different forms of parallel architecture, namely fine grain SIMD and course grain MIMD, into a single architecture. In this way it is possible to match the most appropriate computing resource to each algorithm in a given application.

As important as the architecture itself is a method for programming it. This thesis describes a novel multi-paradigm programming language based on C++, which allows programs which make use of both control and data parallelism to be expressed in a single coherent framework, based on object oriented programming.

To demonstrate the utility of both the architecture and the programming system, two applications, one from the field of image understanding the other image generation are examined. These applications combine some novel algorithms with other novel implementation approaches to provide the most effective mapping onto this architecture.

Declaration

The composition of this thesis is entirely my own work. The work on which it is based was carried out within the VLSI Architecture Group in the Computer Science Department at Warwick University.

All work relating to the design and construction of the Warwick Pyramid Machine was carried out jointly by the VLSI Group. I have been a member of the VLSI Group since it was first set up, and have been fully involved in every part of the design and some parts of the implementation of the Warwick Pyramid Machine.

This joint work has been published in:
[Atherton 90a] [Atherton 90b] [Francis 90] [Nudd 88] [Nudd 89] [Nudd 90a] [Nudd 90b] [Nudd 91a] [Nudd 91b] [Vaudin 89a] [Vaudin 89b].

The work relating to the design and implementation of Pyramid C++ was carried out entirely by myself.

The work relating to the model based alignment estimation system, and its mapping onto the W.P.M. is my own work. This work was carried out as part of a larger project in which a number of people were involved. The work described here constituted the module of this project, for which I was responsible.

The work relating to mapping the image generation system onto the W.P.M. is my own work.

Acknowledgements

I would like to thank all the members of the VLSI group past and present for providing the atmosphere in which this work could be carried out. In particular I would like to thank my supervisor Professor Graham Nudd for his support and assistance during my somewhat long stay in his research group.

CONTENTS

Chapter 1

Introduction

1.1 Background

While conventional computer architectures have achieved great improvements in performance up to now, there are still a number of areas where these sequential computers cannot provide sufficient processing power to meet the requirements of the application. Notable amongst these are real time image analysis and image generation, which have applications in robotics, defence, industrial inspection, design and simulation to name but a few.

While there are systems in use that can attempt some of these tasks, they are generally rather limited, often using specialised hardware which can only perform a restricted set of operations, which cannot be altered to meet changing requirements [Abram 85][Glassner 85].

Over the past six years the VLSI Group at the University of Warwick's Computer Science Department have been working on the design of a more general system based on a fully programmable massively parallel architecture which is sufficiently powerful to perform a wide variety of image based tasks in real time, and which is not restricted to a small set of algorithms.

The key to this design has been the development of an architecture which, while retaining complete programmability, is optimised to the class of algorithms that are typically found in image based applications. This optimisation has been based on a study of the structure of image based applications, in particular image understanding and image generation.

1.1.1 Image Understanding

An image understanding system takes an image as input, performs some processing on that image, and produces some form of high-level description of the contents of the image as output. To do this it would typically proceed in a number of stages, at each one refining the data into a more abstract form.

Initially low-level or iconic processing is performed, which includes such operations as edge detection and filtering. This produces a modified image as output, which forms the input to the next stage of processing, which is typically segmentation.

Segmentation involves subdividing the image into regions of interest which are associated with objects or parts of objects in the image. The exact method used will depend on the types of regions which are to be extracted. Typical systems use regions with uniform grey levels or textures, lines or other geometric shapes or corners.

Once the image has been segmented into regions of interest, information about each of the regions is extracted. Typically this information might include grey level, variance, length and end points of a line, or the angle of a corner. This process, known as feature extraction, performs an abstraction from the iconic image data to a more numeric representation which is passed on to the final stage of processing.

The last stage is the numeric to symbolic model based processing which takes the extracted data and uses it to form hypotheses about the scene represented by the image. This may involve using a priori knowledge of the possible contents of the scene to identify the relevant features and then match them with hypothesised objects. This matching process provides data on the position and orientation of the objects in the image.

Once a hypothesised model has been generated, the system can then perform purely symbolic high level reasoning based on the interrelationship of these hypothesised objects. This reasoning will typically form part of the wider system in which the image analysis system is being used, such as a robotic control system, and will allow the overall system to make decisions based on the visual input to the system.

1.1.2 Image Generation

Image generation is essentially the reverse process of image analysis. Instead of starting with an image and producing a model of the contents of the image, the system starts with a model and produces an image of that model. As with image analysis systems, image generation systems proceed in a number of stages.

The first step in processing typically 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 objects were aircraft in a flight simulator.

In the next stage the illumination of the objects is calculated based on their positions with respect to a set of 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, which 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.

1.2 Architectural Solution

The key point of similarity between both these image based applications, is that they both consist of a number of stages of processing, which perform different types of processing on different types of data. These range from globally uniform processing on regular arrays of integer data, to complex model based symbolic reasoning on complex structures of symbolic or floating point data.

The highly heterogeneous nature of these problems makes them particularly difficult to solve on any existing parallel architecture because the performance of parallel architectures is highly dependent on the structure of the algorithms used. Typically an architecture which performs well on a globally uniform problem would perform poorly on a highly irregular problem, and vice versa.

To overcome this problem the VLSI Architecture Group has developed an architecture which combines a massively parallel fine grain SIMD array with a coarse grain MIMD array, to provide a wide spread of parallel computing capabilities which can be matched to the different aspects of the chosen applications.

The design of this architecture has posed a number of interesting problems beyond its actual hardware construction. Most notable of these is how such a heterogeneous machine can be programmed, and how algorithms can be mapped in a coherent way onto such an architecture.

1.3 Thesis Outline

In this thesis this architecture will be described in detail, along with a novel programming system which has been developed to allow the machine to be programmed in a uniform and consistent way. Finally, how two applications, one from the field of image understanding the other from the field of image generation can be mapped onto this architecture using the proposed programming system will be discussed.

Chapter 2 - contains a review of existing parallel architectures, and discusses the suitability of a number of existing approaches to the different aspects of image based applications. It then goes on to discuss how a multi-paradigm approach can provide an appropriate solution to such problems.

Chapter 3 - contains a detailed description of the Warwick Pyramid Machine, a dual paradigm heterogeneous parallel architecture for image based applications.

Chapter 4 - contains a review of programming languages and techniques for existing parallel architectures.

Chapter 5 - contains a description of a novel dual paradigm programming model based on parallel object oriented programming. This model is discussed in general without reference to a particular implementation. Its appropriateness for use with multi-paradigm parallel architectures is also discussed.

Chapter 6 - contains a detailed description of Pyramid C++ a dual paradigm object oriented parallel programming language based on C++ for the Warwick Pyramid Machine. The implementation of this language on the Warwick Pyramid Machine is also discussed in detail.

Chapter 7 - contains two case studies, one in image analysis, the other in image generation. The mapping of these problems onto the Warwick

Pyramid Machine using Pyramid C++ is discussed.

Chapter 8 - contains a general discussion of the points raised in the rest of the thesis.

Chapter 2

Parallel Architectures

2.1 Introduction

Image based applications such as image analysis and image generation require very large amounts of computing power if they are to achieve the real time performance required by areas such as robotics, interactive design and simulation. This level of computing power is beyond that currently available from conventional sequential machines, and this situation is likely to continue since sequential architectures are inherently limited in performance.

If the trends in computer technology are examined (figure 2.1) [Ruechardt 87], it can be seen that over the past twenty years the number of transistors that can be fitted onto a single chip of silicon has increased by a factor of roughly two every eighteen months representing a total increase of four orders of magnitude in this period. On the other hand the clock speed of these chips has only increased by two orders of magnitude in the same period.

This difference stems from the different ways in which these two quantities scale with the minimum feature size (that is the size of smallest element that can be 'drawn' onto the silicon). Clock speed scales roughly linearly with reducing feature size, whereas total devices per chip scales as the square of linear feature size. Thus it can be expected that this trend will continue assuming no dramatic shift to alternative technologies [Nickel 87].

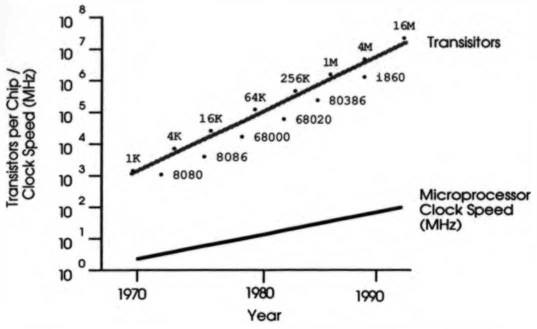


Figure 2.1: Trends in VLSI Technology [Ruechardt 87]

In a sequential machine, often referred to as a von Neumann machine after the mathematician Jon von Neumann, the processor performs an operation on one piece of data at a time. Each operation takes a number of clock cycles to perform, the exact number is dependent on the specific architecture, but for any given sequential architecture the total number of operations per second will be proportional to its clock speed.

Thus the performance of purely sequential designs scales roughly with the best clock speed currently available. As seen from the figure 2.1, clock speeds are increasing quite slowly when compared to the increase in the number of devices that can be fabricated on an integrated circuit. This fact is reflected in a correspondingly slow increase in the performance of sequential systems.

The one operation at a time limitation of these architectures is often referred to as the von Neumann bottleneck. To overcome this limitation computer architects have looked towards parallel processing techniques as a way of achieving increased performance. These techniques make use of a number of processors working together rather than a single fast processor and can therefore take advantage of the very rapidly increasing number of available devices. Parallel architectures can perform several operations at one time, and can therefore overcome the von Neumann bottleneck.

2.2 Parallel Architecture Classifications

At its simplest, the idea behind parallel computing is that a very powerful computer may be built by making use of a large number of cooperating processors. The computational power of each individual processor is not the primary concern, as the overall computational power results from the combination of the power of the individual processors.

Computational power is taken to be the number of arithmetic operations, typically on 32 bit integers, a system can perform each second. Thus if a single processor can perform M operations per second, then N processors taken together should be able to perform NM operations per second. In practice however this ideal figure is not generally achievable, because it is not always possible to arrange for all processors in a machine to be performing useful computations all of the time, and this reduces the computational efficiency of the system.

Computational efficiency can be defined as the proportion of the peak computational power (NM) that a system achieves. Peak efficiency can only be achieved if all processors are performing useful computations all of the time. The computational efficiency of a system is proportional to the number of processors that are active, that is performing useful computations, at any instant. This proportion may vary considerably during a computation, and will depend both on the architecture of the system, and the algorithm being performed.

There are a number of reasons for parallel systems not reaching peak efficiency. The most fundamental of these is insufficient opportunity for parallelism in the algorithm being executed to keep all processors busy. Fortunately there are many applications which have ample opportunity for parallelism to keep even the largest parallel machines fully utilised, particularly in the image analysis and image generation fields.

Another important reason for loss of efficiency is delays caused by processors communicating with other processors. Communication takes a finite time, and during this time any computation which depends on the result of the communication cannot proceed. This may result in processors remaining idle for a time while a communication is taking place, and during this time they are not performing useful computations. Typically the more processors there are in a system the more difficult it will be to keep all of them fully utilised at all times.

Another aspect of efficiency is resource efficiency, that is the number of operations per second that can be performed for a given amount of hardware resources. The main component of the hardware resources is the number of transistors used by a design. For a given implementation technology, the number of transistors will be proportional to silicon area used, and thus number of integrated circuits. The other main component of hardware resources are interconnections, these include the number of pins on an integrated circuit package, and therefore the size of the package. This in turn affects the complexity and size of the circuit board required, and also the number of connections between circuit boards. All these interrelated factors affect the physical size, power requirements and ultimately the cost of the system.

Many parallel architectures have been proposed which attempt to balance peak computational power, computational efficiency and resource efficiency to produce a viable parallel machine. All share the same basic principle of including multiple processors each of which is capable of performing one operation on one item of data at a time such that together they can perform many operations on multiple items of data in parallel.

There are a number of different architectures which each have certain advantages and disadvantages in particular problem areas. These architectures can be classified by three main criteria, their memory structure, that is whether they use shared or distributed memory, their control structure, whether they support multiple instruction streams with multiple data streams (MIMD) or a single instruction stream with multiple data streams (SIMD), and their communications structure, how the processors are connected together.

MIMD		
Shared Memory		
Bus	Sequent Balance	[Sequent 86]
	Encore Multimax	[Gehringer 88]
Multi-Stage Switch	NYU Ultracomputer	[Gottlieb 83]
Distributed Shared Memory		
Multi-stage Switch	BBN Butterfly	[Brooks 85]
	IBM RP3	[Pfister 87]
Distributed Memory		
Switched	Meiko Surface	[Chesney 87]
	SuperNode	[Refenes 90]
Mesh	AT&T Pixel Machine	[Potmesil 89]
HyperCube	Intel iPSC/2	[Intel 87]
	NCUBE	[NCUBE 88]
	DOOM	[Annot 90]
	FPS T Series	[FPS 87]
SIMD		
Distributed Memory		
Mesh	MPP	[Batcher 80]
	BLITZEN	[Davis 88]
	CAAPP	[Foster 83]
	Pixel Planes	[Fuchs 85]
	DisArray	[Page 83]
	DAP	[AMT 88]
	CLIP4	[Fountain 87]
	RPA II	[O'Gorman 89]
HyperCube	Connection Machine	[Hillis 85]
Other	WASP	[Lea 86]

Table 2.1: Classification of Representative Parallel Machines

Table 2.1 shows a sample of representative parallel machines classified according to memory structure, control structure and communications structure. Pipelined architectures are not considered as parallel architectures in this discussion, since they do not in general involve multiple functional units, rather they use parallelism within a single functional unit. In the following sections these architectural structures will be described in more detail, and the various design tradeoffs associated with them will be discussed.

2.2.1 Memory Structure

As mentioned above all parallel architectures include multiple processors each of which can perform one operation at a time. Ideally each of these processors would have its own path to memory, such that all processors could access any piece of data anywhere in memory in the same amount of time.

This idealised model is similar to the P-RAM model [Gibbons 88] used by the theoretical community. In practice however it is not a feasible arrangement, because as the number of processors increases the amount of hardware resources required to implement it becomes uneconomic.

2.2.1.1 Ideal Shared Memory Machine

To implement an ideal shared memory machine, there must be a separate path from every processor to memory, so that there is no contention for memory paths, and thus no delays which might affect computational efficiency. The memory must be able to provide simultaneous access for every path. This involves the use of memory devices with one port for every processor.

A memory port is the channel through which data is accessed from a memory device. If more than one access is required to a memory device at one time, then each access must use a separate port. Typical memory devices support just one port, although some may support perhaps as many as six ports, allowing up to six accesses at a time. To support simultaneous access by all the processors in a large parallel machine which might have a thousand processors, would require a memory with a thousand ports.

Each port into a memory device requires its own set of input/output pins. A one thousand port memory would require at least one thousand pins, which is well beyond the current state of the art. Within the memory device each port requires access to every cell within the memory array. This means there must be separate data busses to every memory cell for every port. As the number of ports is increased the amount of silicon area taken up with wires will at some point overtake that used for memory cells. These factors make the construction of memories with very large numbers of ports uneconomic, and this makes the idealised shared memory machine impractical.

Hillis [Hillis 85] argues that it is unreasonable to expect that such an idealised system could ever be constructed, since it ignores the fundamental laws of physics, in that it assumes information can travel in unit time from any point to any other, regardless of the size of the system. He argues that any useful model must take these fundamental truths into account.

The shared memory model is however conceptually elegant and simple, and many existing designs do provide such a model. In the absence of memory devices with very large numbers of ports designers have developed compromise solutions which attempt to balance hardware resources and computational efficiency. These solutions can be divided into three categories, shared memory, distributed shared memory and distributed memory systems.

2.2.1.2 Shared Memory

Shared memory systems are an attempt to implement a model as close as possible to the ideal shared memory machine, where all processors can simultaneously access any parts of memory with equal latency, but without the use of memory devices with very large numbers of ports. Figure 2.2 shows a typical shared memory system. It consists of a number of processors, connected to a number of separate memory banks by a memory switching network.

The switching network is responsible for routing memory requests from each processor to the appropriate memory bank, so that each processor has access to all of memory. In this diagram the number of memory banks equals the number of processors, but this is not necessarily the case.

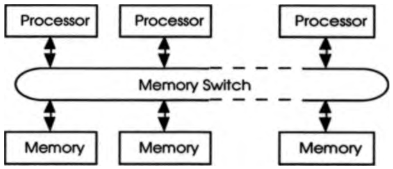


Figure 2.2: Shared Memory System

In such a system each memory bank can typically support one access at a time. Thus if all the processors access a different memory bank all will be able to perform their accesses simultaneously. However if two processors attempt to access the same bank, the two accesses will be performed sequentially. This introduces a delay, which will reduce the computational efficiency of the system compared to an ideal shared memory system.

A number of problems are encountered when very large shared memory systems are constructed. Most important of these is that the size of the switching network becomes very large as the number of processors increases. If the designer uses a very fast network the hardware resources used to implement it can dominate the system cost for large systems.

The highest performance switching network is the crossbar switch (described in section 2.2.2.2) which can provide a separate path from every memory bank to every processor. This network requires hardware resources which scale as the square of the number of processors. Since the hardware resources used to implement the processors themselves scale linearly, it can be seen that for a large number of processors the resources used to implement the memory switch will dominate the system cost.

If a less resource intensive but slower network is used, it will increase the delays introduced by the communication system, which will in turn reduce the computational efficiency of the system. Whatever network is used it will inevitably introduce some delay into each memory access and this can reduce the sequential instruction rate of each processor and thus reduce the peak computational power of the system. This becomes increasingly true as the number of processors, and the size of switch and/or the contention for the limited available switch bandwidth increases.

Because of these factors shared memory systems of this kind are generally limited to a relatively small number of processors. Typical commercially available systems of this type use less than fifty processors.

2.2.1.3 Distributed Shared Memory

The key to allowing more processors to be used in shared memory systems is to reduce the bandwidth requirement of the interconnection network, and thus allow slower networks, which use fewer resources, to be used. The key to achieving this is the observation that access to memory is not random, but structured.

Processors spend much of their time accessing relatively few memory locations, which are often close together, a phenomenon known as locality of reference. A rule of thumb is that programs spend 90% of their time accessing 10% of their memory, known as the 90/10 rule [HenPat 90].

If the commonly accessed code and data for each processor was stored locally to that processor then it would spend most of its time accessing its local memory. By allowing the processor to access its local memory directly without needing to go through the memory switch, most accesses would be speeded up and thus the overall throughput increased. This is the essence of the distributed shared memory architecture.

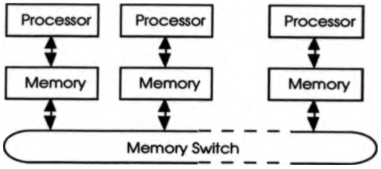


Figure 2.3: Distributed Shared Memory

A distributed shared memory system is shown in figure 2.3. Each processor is connected to its local memory, so that it can access data stored within it without the overhead of going through the memory switch. However the memory switch is still provided to handle memory references which fall outside the processor's local memory. When this happens the memory switch routes the access to one of the other memory banks, which is one of the other processor's local memory.

These systems scale better than the pure shared memory systems, because the memory switch is not required to be as fast. This allows the use of switching networks which do not require as much hardware resources to implement. Even with these slower memory switches the average memory access time does not necessarily increase dramatically, so long as most memory accesses are performed on local memory.

The main problem with these architectures it that the time to access different parts of memory becomes unequal, since accessing local data is much faster than non-local data. This makes the correct distribution of data very important, in order to ensure that most of the data used by each processor resides in its local memory.

The distribution of code and data may be performed by sophisticated system software which dynamically migrates segments of code and data to processors which require them. Alternatively complex caching hardware can be provided which performs the same function. Most commonly however it is the job of the programmer to ensure the programs are divided up appropriately amongst the different processors.

The necessity for programmers to distinguish between local and non local memory goes against the basic principle of the idealised shared memory machine. However, as noted by Hillis [Hillis 85], it is probably an inevitable feature of any large real machine.

2.2.1.4 Distributed Memory

Moving to larger numbers of processors, the requirement to reduce the communications bandwidth required by each processor becomes even more important, if the amount of resources given over to communications is not to dominate the system cost.

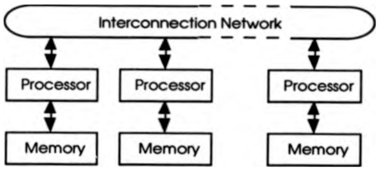


Figure 2.4: Distributed Memory

The necessary reduction in communication bandwidth can be achieved by moving to a distributed memory system, such as shown in figure 2.4. In these systems each processor is connected only to its local memory, there is no hardware support for non local memory accesses. Instead processors cooperate on problems by passing messages over a separate processor interconnection network.

The key to distributed memory systems is the assumption that the application programmer can pre-distribute programs in such a way that very little communication will be required between processors. Assuming this is the case, the interconnection network can afford to be relatively slow, since it does not have to support memory accesses, and thus does not have to offer the low latency this implies.

The message passing approach provides the potential to hide the long intercommunications latencies by scheduling another process while the communication is taking place. The Transputer [Shepard 87] for example makes use of so called parallel slackness or excess parallelism, by running more than one process on each processor. Ideally there will always be at least one process ready to run, while others are waiting for communications.

The assumption of distributability restricts the classes of applications that can be successfully mapped onto distributed memory machines. However for applications which do map, distributed memory machines are very resource efficient architectures, even for systems with relatively small numbers of processors, since they require less resources to be used for interconnection.

Distributed memory systems can scale very readily, because the resources needed by their communications networks tend not to increase as quickly as higher performance networks. This makes such systems ideal for massively parallel applications where hundreds or even thousands of processors are to be connected together.

The disadvantage of such systems is the necessity for total distribution of data and code onto each processor, because of the very slow access to non-local data. This places a particularly heavy burden on the programmer whose responsibility it is to perform the necessary distribution. This has given distributed systems a reputation for being highly specialised and difficult to program. Despite this however, the scalability of distributed memory systems has resulted in them constituting the bulk of existing parallel architectures.

2.2.2 Interconnection Schemes

All parallel architectures require some form of interconnection mechanism, either to route data from memory to processors, in the case of the shared memory systems, or to route data from processor to processor in the case of the distributed memory systems. A wide variety of designs have been used to implement these networks, each exhibiting different performance characteristics and resource requirements.

There are three main characteristics associated with these communications networks: throughput, latency and complexity. The throughput refers to the maximum rate that data can be passed through the network. Latency refers to the time that each packet of data takes to complete its journey from source to destination. The complexity of the network is the amount of resources required to implement it, and perhaps more importantly how those resources increase as the size of the machine increases.

Initially it might appear the throughput and latency are simply two ways of measuring the same thing, since if each piece of data is delivered more quickly, the total amount of data delivered in any given time must also be higher. However most systems transfer many pieces of information simultaneously with each one spending some time in transit.

The maximum throughput T in packets per second is given by

$$T = P_T \ / \ L$$

Where L is latency in seconds per packet of data, and P_T is the maximum number of packets in transit at any instant.

An ideal system will maximise throughput while minimising latency, but in practice many systems will trade throughput for latency or vice versa. The network used in a given system will depend on the relative importance of these parameters for that system.

Shared memory systems place the highest burden on the network system because it is used for all memory references. This means that the network needs to be capable of transferring data at a rate comparable to the total data transfer rate of all the processors combined. Also since the processor will usually be suspended waiting for each memory reference, the network must have a latency comparable to a single processor memory cycle, otherwise processors will lie idle, and reduce the computational efficiency of the system.

Distributed shared memory systems reduce the total required throughput of the network, since not all memory references go through the network. However they retain a requirement for fairly low latency, as while non local memory accesses are performed the processor will usually be idle.

Fully distributed systems place least burden on the interconnection network, since they remove the need to support memory references at all, using software controlled data passing instead. In particular distributed systems can tolerate quite high latency, by utilising excess parallelism to hide communication delays. However because distributed systems tend to make use of large numbers of processors, the total throughput of the network must be high, and must scale with the number of processors attached.

2.2.2.1 Bus Based Systems

The simplest and most common interconnection scheme is the bus. This consists of a set of shared lines that connect all the devices together as shown in figure 2.5. Bus systems have very low minimum latency, since data can be transferred directly from source to destination without going through any intermediate stages. They are also very straightforward to construct, requiring very few resources, which grow linearly with the number of devices connected. This low complexity allows the construction of extremely high speed busses, using relatively few resources, which can sometimes outperform theoretically superior systems.

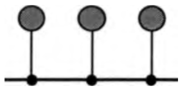


Figure 2.5 Bus Interconnection

However busses ultimately have a fixed, even if potentially quite large, total throughput. As the number of processors increases, and the total network traffic approaches the available throughput, contention for the bus becomes an increasing problem, which in turn increases the average communication latency.

The very low minimum latency of busses makes them ideal for shared memory systems, where latency is particularly critical. However their fixed total throughput means that the number of processors that can be supported is relatively low. Typical currently available commercial systems use up to thirty processors.

2.2.2.2 Connection Networks

One way to increase throughput over the bus is to move to a system that can perform more than one transfer at one time. This requires the network be made up of many isolated sections, each of which are capable of transferring data independently of the others. These sections must then be interconnected either statically or dynamically to allow data to flow from one section to another, and so move from source to destination.

Directly Connected Networks

The simplest class of such networks are the directly connected networks. In these each node is permanently connected to a subset of other nodes. These nodes are in turn connected to other nodes such that all nodes are connected either directly or indirectly.

There are a number of possible topologies that can be used in this class of networks, and some of the most common of these are shown in figure 2.6. There are two principal classifications used for directly connected networks, namely their order and their diameter. The order of a network refers to the number of nodes that each node is directly connected to. The diameter of a network is the maximum distance, measured in nodes, between any two nodes.

The networks can be broken down into two main sub-categories, fixed order networks, that scale by increasing their diameter, and variable order networks which scale by increasing their order (usually accompanied by an increase in diameter). The most common form of variable order networks are hypercubes, whose order is equal to their number of dimensions.

Different topologies may be used to optimise the complexity and latency of the network. Latency is approximately proportional to the average number of stages that the data must pass through en route from source to destination, which in turn is related to the network's diameter.

For example a sixty four node system might be arranged as an eight by eight mesh or a six dimensional hypercube. The diameter of the mesh is fifteen, whereas the diameter of the hypercube is only six. Thus the latency of data transfers for the hypercube will be significantly less than for the mesh. However the hypercube contains 196 wires compared to the mesh's 112 wires, so the hypercube will require more interconnection resources to construct.

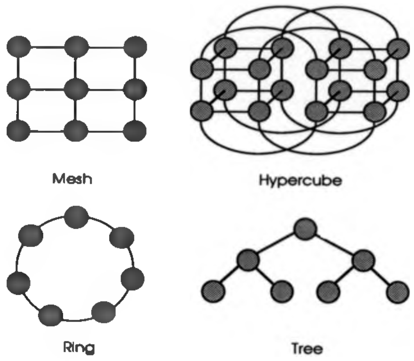


Figure 2.6: Directly Connected Networks

Each node of the hypercube must be directly connected to six other nodes, whereas each node of the mesh need only be connected to four others. More importantly if the number of nodes in the hypercube is to be increased, the number of connections to all the other nodes must be increased, since the order of the network must be increased. Increasing the number of connections made from a node can be awkward, since it involves altering the existing nodes. Avoiding this usually involves the use of designs which have a fixed maximum order, and thus a fixed maximum number of processors.

A mesh can be constructed easily in two dimensions, making it straightforward to lay out as an integrated circuit or printed circuit board. The hypercube does not lend itself to easy implementation in two dimensions and this may add to the construction complexity.

All the directly connected networks have a total throughput, which scales with the number of nodes, O(n) in the case of the mesh and O(n log(n)) in the case of the hypercube. This makes them ideal candidates for massively parallel distributed memory systems which can tolerate their relatively long latency. Constructional complexity varies from very low in the case of the mesh to moderate in the case of the hypercube, but even these are used in existing systems to connect many thousands of processors.

Switching Networks

The final class of interconnection schemes attempts to produce a compromise between the low latency of the bus with the scalability and high throughput of the nearest neighbour systems. These are switching networks, which consist of single and multi-stage switches.

Single Stage Switching Networks - A fully connected single stage switch also known as a crossbar switch can directly connect any of a number of inputs to any of a number of outputs. Once connected data can flow unhindered from source to destination, making them ideal switch networks. Crossbars can provide throughput that scales linearly with the number of processors, however they have $O(n^2)$ complexity and are therefore expensive to build for large numbers of inputs.

Multi-Stage Switching Networks - can be designed in many different configurations [Feng 81], a typical example being the butterfly network shown in figure 2.7. In such networks data is routed from source to destination through a fixed number of intermediate stages, each of which consists of a number of crossbar switches.

The network in figure 2.7 consists of three stages each of which contains four switches. Each switch is a two by two crossbar, which can connect either of its two inputs to either of its two outputs. At each stage a packet of data can be routed to one of two possible destinations. By using multiple stages the number of possible destinations is increased, in this case by a factor of two at each stage. Thus after three stages a total of eight (2^3) destinations may be selected.

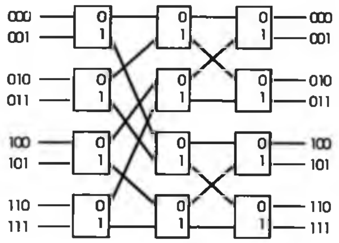


Figure 2.7: Butterfly Interconnection Network

Data is routed through the butterfly network using the data's destination address. Using the destination 101 as an example, and starting from any start point on the left, the path through the intermediate switches is given by the bits of the destination address, starting with the most significant bit. Thus the first switch selects its 1 output, the second its 0 output, and the third its 1 output. In this way the data will always arrive at node 101 no matter which node it started at.

Multi-stage switching networks can potentially provide latencies which are almost as low as a bus, but with best case throughput which scales linearly with the number of processors. This, combined with resources which scale less quickly than the crossbar switch (O(n log(n)) in the case of the Butterfly) makes them ideally suited to large shared memory machines. However they do suffer from contention if two packets both require routing along the same connection which can reduce their throughput compared to the more resource intensive crossbar switch.

2.2.3 Control Structure

The other principal classification for parallel architectures regards the distribution of control amongst the processing elements. This classification is due to Flynn [Flynn 66] who categorised all architectures into one of four groups.

These are shown below

Single Instruction Single Data	(SISD)
Single Instruction Multiple Data	(SIMD)
Multiple Instruction Single Data	(MISD)
Multiple Instruction Multiple Data	(MIMD)

These classifications are divided into two parts, the first is concerned with the number of instruction streams, the other with the number of data streams. In a conventional von Neumann architecture the processor fetches one instruction at a time, providing a single instruction stream. Each instruction may perform an operation on one piece of data, providing one data stream. Thus a conventional sequential architecture would be classified as an SISD or single instruction single data.

All parallel architectures perform many operations at once, on many pieces of data, thus they all provide multiple data streams. However they may or may not allow multiple instruction streams. This provides two alternative approaches to building parallel systems, namely SIMD and MIMD.

2.2.3.1 SIMD Machines

Single instruction multiple data (SIMD) machines fetch one instruction at a time, in the same way as a conventional sequential architecture, but each instruction can operate on many pieces of data simultaneously. This is achieved by the provision of multiple arithmetic and logical units, each of which has its own path to memory, allowing them all to simultaneously perform an operation on a piece of data stored in their local memory.

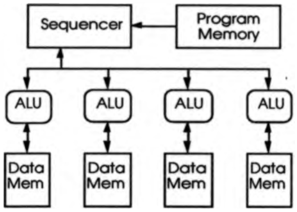


Figure 2.8: Single Instruction Multiple Data (SIMD)

Figure 2.8 shows a simplified diagram of a distributed memory SIMD machine. The Sequencer is responsible for fetching instructions from the instruction memory, where the user program resides. When the sequencer has read an instruction it is broadcast to all of the ALU's, which all perform the appropriate operation on their own local data.

The sequencer also generates the address of the data within the ALU's local memory that is to be operated upon. In this way all the ALU's operate on the same location within their own memory. This can be thought of as operating on an array of data where one element of the array resides within each of the ALU's local memory, and these machines are often referred to as array processors .

2.2.3.2 MIMD Machines

Multiple instruction multiple data architectures provide a separate instruction stream for each data stream. This arrangement is shown in figure 2.9. In this system each sequencer and ALU is grouped together to form what is essentially a conventional sequential processor. The sequencer reads the instructions from a combined program and data memory, and then instructs its local ALU to perform the appropriate operation. Again the sequencer is responsible for generating the address of the data to be operated upon.

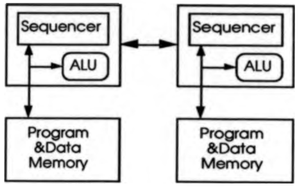


Figure 2.9: Multiple Instruction Multiple Data (MIMD)

An MIMD machine can be thought of as a collection of sequential machines, and they are often referred to as multi-computers. The important distinction is of course that they are connected together using one of the communication techniques described in the previous sections. In this way they can cooperate on the solution to a single problem.

2.2.3.3 Granularity

SIMD architectures have a significant resource efficiency advantage over MIMD architectures, in other words for a given amount of hardware resources an SIMD architecture can achieve more operations per second than an MIMD architecture. This is because each processor in an SIMD machine does not need its own sequencer and the resources that this requires. This means that more resources can be allocated to implement arithmetic units, rather than control logic.

In addition to allowing more resources to be used for arithmetic units, SIMD architectures also allow the arithmetic units to be more resource efficient. It would be inefficient to have a processor whose arithmetic unit was much smaller than the control logic associated with it. In an MIMD architecture each processor must contain logic which implements its instruction sequencer. Because of this it is generally inefficient to use MIMD processors with very small arithmetic units. SIMD architectures do not require logic to implement an instruction sequencer for each processor, which makes very fine grain architectures viable.

The finest grain processors are one bit or bit serial processors, and this is the type used by most SIMD architectures. Fine grain processors are arguably more efficient than coarser grain ones because their arithmetic units can be fully utilised on a wider mix of operations. If for example each processor were equipped with a floating point arithmetic unit, it would lie idle for all operations that did not involve floating point operations. The silicon area used by the floating point unit would therefore not be being used effectively. By comparison the arithmetic unit of a bit serial processor will be used on every arithmetic operation.

Another advantage of bit serial processors is that their arithmetic units do not suffer from the carry propagation delays associated with multi-bit designs. This allows single bit arithmetic units to operate at higher clock frequencies than multibit designs. The combination of these factors has led to the suggestion that single bit designs are inherently more efficient than other configurations [Hillis 85][Jesshope 89].

However it is a complex trade off. In a uni-processor architecture the resource efficiency of the processor is relatively unimportant, since silicon area used to implement the processor cannot, in general, be more effectively used elsewhere to improve the performance of the system. In the case of a parallel architecture however any silicon area used to improve the performance of an individual processor could have been used to implement more processors. It therefore only makes sense to improve the performance of the individual processor in a parallel design where the performance increase provided outweighs the disadvantage of having fewer processors.

The optimal solution for any particular architecture will depend on the application for which the architecture is intended. The higher the proportion of long word length or floating point operations that are expected the more likely it is that support for multibit operations will yield the best performance. Also if the total amount of potential parallelism is limited, it may not be possible to achieve improved performance from increasing the number of processors, so again coarser grain processors may produce the best performance.

So far single bit processors have dominated the SIMD field being used in the CLIP, DAP, MPP and Connection Machine. However recent trends suggest that multi-bit ALU's may become more popular in future, with the CLIP7a [Fountain 88] Connection Machine 2 [TMC 89] and the AMT DAP/C8 [AMT 90] all featuring multi-bit ALU designs.

2.2.3.4 Communication

Communication overheads have a large impact on the overall performance of a parallel system, and become increasingly important the more processors there are in the system. SIMD architectures have a clear advantage in this respect because they do not have to synchronise when communicating. For two processors in an MIMD machine to communicate, they must cooperate, such that each send operation performed by one processor is matched by an equivalent receive operation by the other. For this to take place the two processors must synchronise so that both are in the required state.

Synchronisation can be achieved using a mechanism such as a rendezvous, which involves the first processor to reach the point at which a communication is to occur signalling its readiness to communicate, and then waiting for a signal from the other processor. When the second processor reaches the appropriate state, where it is also ready to communication, it signals its readiness and both are then allowed to proceed with the transfer.

This synchronisation process takes time, which adds to the overall communications delay, and so potentially affects the overall performance of the system. By comparison all processors in an SIMD system are controlled centrally by the sequencer, so that all processors are guaranteed to be in the same state at the same time. Thus when communication is required it can be performed without any need for synchronisation.

A restriction associated with many SIMD machines, particularly fine grain ones, is the limited number of efficiently supported communications patterns. This is caused partly by the simple directly connected interconnection networks they generally use, and partly by the requirement to perform the same operation on all processors. The combination of these factors makes it very difficult to efficiently implement algorithms which require complex non-local communications patterns.

The main exception to this rule is the Connection Machine a fine grain SIMD machine which does allow arbitrary routing from one processor to any other. This increases the class of algorithms that can be efficiently processed significantly. Unfortunately the routing process is quite slow, and this can offset the efficiency advantage usually afforded to SIMD systems by not needing to synchronise.

2.2.3.5 Local Autonomy

The principal disadvantage of SIMD architectures is that they only operate efficiently for algorithms which process all their data identically. If an algorithm operates differently on different pieces of data the performance of the machine drops dramatically. In general the only way to implement algorithms that are not globally uniform on an SIMD machine is to disable certain processors so that they remain idle while a set of operations is being performed by the other processors. This clearly has a direct impact on the computational efficiency of the system.

Disabling processors works satisfactorily if there are a small number of alternatives, but becomes progressively less efficient as the number of alternatives increases. At the extreme where an algorithm wishes to perform a different operation on every pixel the performance of the whole system drops to that of a sequential machine.

Some SIMD architectures such as the Connection Machine [Hillis 85] do allow a certain degree of local autonomy for each processor. This autonomy allows each processor to modify the globally broadcast instruction in certain ways to allow the processors to operate partially independently. In the Connection Machine the aspects of the instruction which can be altered are the address on which the instruction is to be performed, and the source of any non local data used in the operation. In general though most SIMD architectures do not allow this level of autonomy.

MIMD machines by comparison allow each processor to operate completely independently of all the others. This local autonomy means they can support algorithms which perform a different operation on each piece of data just as efficiently as one which operates identically on all pieces of data.

In summary SIMD machines are potentially highly efficient and scalable, but tend to only operate efficiently on a restricted class of algorithms. MIMD architectures are rather less efficient but can operate effectively on a wider variety of algorithms.

2.3 Existing Parallel Architectures

In this section a number of existing parallel architectures will be described. These architectures use all of the architectural features described above, and serve to illustrate the advantages and disadvantages of each of the various architectural features that have been outlined above.

2.3.1 MIMD Systems

There are a large number of existing MIMD architectures. The reason for the proliferation of such machines is most probably due to their ease of construction, which results from them being able to take advantage of existing microprocessor technology.

This use of existing components has allowed the designers of these machines to avoid the considerable expense and difficulty associated with the development of custom VLSI experienced by the designers of SIMD machines [Fountain 87]. It also allows the machines to take advantage of the vast investment in mainstream microprocessor technology.

The IPSC/2 for example has been upgraded from the Intel 80386 to use the new Intel i860 RISC processor, which provides almost an order of magnitude improvement in performance. It is unlikely that any of the relatively specialised producers of SIMD processors will have access to such up to date technology. For example the newest AMT DAP processor contains about 50,000 transistors compared to the i860's 1.2 million.

2.3.1.1 Shared Memory Systems

Sequent Balance - [Sequent 86] allows up to 30 National Semiconductor 32032 processors to share up to 32 Mbytes of central memory as shown in figure 2.10. The processors are connected to the memory system via a single 64 Mbytes per second bus.

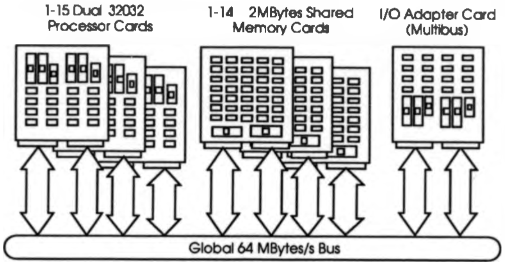

1-15 Dual 32032 Processor Cards
1-14 2MBytes Shared Memory Cards
I/O Adapter Card (Multibus)
Global 64 MBytes/s Bus

Figure 2.10 : Sequent Balance : A Shared Memory System

Encore Multimax - [Gehringer 88] uses up to twenty National Semiconductor 32332 processors, which access up to 128 Mbytes of central shared memory via a single 100 Mbytes per second bus.

Both the Balance and the Encore are general purpose multi-user UNIX machines oriented towards supporting many users. Their shared memory arrangement allows them to easily schedule multiple tasks between the pool of processors.

Both machines use a bus to provide the low latency access required to the central shared memory. However as discussed previously the use of a bus restricts the maximum number of processors that can reasonably be accommodated to around thirty.

2.3.1.2 Distributed Shared Memory Systems

BBN Butterfly - [Brooks 85] allows up to 256 nodes to be connected together. Each node consists of a Motorola 68020 processor, up to 4 Mbytes of local memory, and a memory control unit.

The memory control unit is responsible for handling non local memory references. Whenever the processor accesses a location in memory the memory controller checks to see if that location falls within the bounds of local memory. If so the request can be satisfied immediately from local memory. Otherwise the controller sends a request to the appropriate processor/memory node for the data.

The BBN uses an eight stage butterfly network to route requests and data between nodes. A local memory access requires about 500ns to complete, while a non local access takes around 5μs.

IBM RP3 - [Pfister 87] can support up to 512 nodes, each of which consists of a 32 bit proprietary RISC processor and up to 8 Mbytes of memory. Each node contains a memory manager that handles non-local memory accesses.

The nodes are connected by an Omega network [Feng 81] . Local memory references take 100nS while non-local references around 500ns. The total memory connection network throughput is approximately 12.8 Gigabytes per second.

The distributed shared memory machines, as expected, allow significantly more processors to be supported than the pure shared memory system. This however is only achieved by using resource intensive multi-stage switching networks, that are required to provide the necessary bandwidth requirements of the large number of processors. Thus while these machines are more scalable than pure shared memory systems, they are large and expensive systems.

2.3.1.3 Distributed Memory Systems

SuperNode - [Refenes 90] is built up of modules, each of which contains 16 Transputers with 256 Kbytes of local memory, expandable to 4 Mbytes. Each Transputer provides four 20 Mbit/s serial links for communication. Within each module these are connected through a full 16 x 16 crossbar switch which allows any link to be connected to any other link, all communicating at 20 Mbit/s.

Up to 64 of these modules can then be connected together via a second level switch, giving a maximum of 1024 processors, with a total of 4 GBytes of memory.

Meiko Computing Surface - [Chesney 87] is built from an arbitrary number of boards known as elements. These elements can be one of a number of different types. Compute elements consist of four Transputers each with 256 Kbytes of memory, expandable to 4 Mbytes. The display elements contain a single Transputer, with up to 1.5 Mbytes of display memory, allowing 500,000 24 bit pixels. Other elements include application specific elements, and host interface elements.

Inter-processor communications are provided by an expandable single level switch. There is no fixed maximum number of processors, and systems of up to 1000 processors have been constructed, although a few hundred is more common. Of particular interest is that the system provides a separate pixel bus to connect display elements, this allows very high speed image output, and allows real time image generation to be performed.

FPS T-Series - [FPS 87] is built up of modules, with eight processors per module, and up to 16 Mbytes of memory per processor. The processor used is a Transputer with an additional floating point vector processing unit.

Interprocessor communication is provided by an order 14 hypercube network, which allows up to 16000 of the modules to be connected together giving a theoretical limit of 128000 processors and 4 TeraBytes of memory.

Phillips DOOM - [Annot 90] is a message passing architecture optimised for the support of object oriented programming. It consists of up to 256 nodes, where each node consists of a 68020 with floating point co-processor and a memory management unit. Each node can contain up to 4 Mbytes of local memory. The communication network is an order eight hypercube, based on Inmos links, and thus sharing the Transputer's 20 Mbit/s speed.

Intel iPSC/2 - iPSC/2 [Intel 87] consists of up to 128 nodes. Each node consists of an Intel 80386 processor, and up to 16 Mbytes of memory. The nodes are connected by an order seven hypercube. The interconnections are implemented using Ethernet technology and operate at 10 Mbit/s.

NCUBE - [NCUBE 88] contains up to 1024 nodes, each of which consists of a single custom VLSI processor chip, and up to 512 Kbytes of memory. The compact design allows up to 64 processors to be fitted on a single circuit board. The processors are interconnected via an 8 Mbits/s order ten hypercube network.

AT&T Pixel Machine - [Potsemil 89] is divided into two halves, processing nodes and display nodes. All nodes contain an AT&T DSP32 processor, a RISC like 32 bit microprocessor optimised for digital signal processing tasks by the inclusion of fast multiply accumulate hardware, and 4 Kbytes of memory.

The display nodes contain 512 Kbytes of video memory, which forms part of a distributed frame buffer, providing high speed image output similar to the Meiko Computing Surface.

The distributed memory systems can support many more processors than the shared memory systems described previously, although in the case of iPSC/2 and the DOOM they do not actually do so. Both these machines as well as the NCUBE and the T-Series are hypercube machines, and therefore have a fixed maximum number of processors. The SuperNode also has a fixed sized communication network, but other machines use scalable interconnection networks, which do not suffer from the same problem.

All the MIMD machines described are coarse grain designs using conventional microprocessor technology.

2.3.2 SIMD Machines

There are a number of SIMD architectures in existence, all of which are distributed memory designs, and all of which adopt radically different design options than the MIMD machines described above.

CLIP4 - [Fountain 87] or Cellular Logic Image Processor is a 96 x 96 square array of bit serial processors, each with 32 bits of local memory. The processors are connected by an eight connected mesh.

DAP - [AMT 88] or Distributed Array Processor consists of a 32 x 32 array of simple bit serial processors connected in a four connected mesh. The processors are connected to external memory, which is typically 32 Kbits per processor.

MPP - [Batcher 80] or Massively Parallel Processor consists of a 128 x 128 array of bit serial processors connected by a four connected array. Each processor is connected to up to 1 Kbits of local memory.

CAAPP - [Foster 83] or Content Addressable Array Pixel Processor is a very large 512 x 512 array of bit serial processors. A block diagram for the processors, (which is fairly typical of this type of machine) is shown in Figure 2.11. Each processor has 32 bits of local memory.

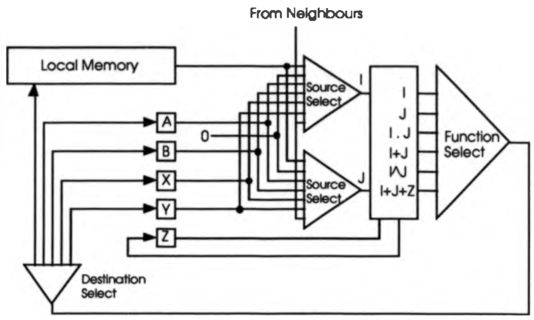


Figure 2.11: Simplified Block Diagram of CAAPP Processor

A key feature of the CAAPP is its associative capability. This allows individual processors to set flags in their memory, as the result of global tests. These flags can then be tested globally using a some/none line, which indicates the presence or absence of responders.

DisArray - [Page 83] is a 16 x 16 bit-serial array, optimised for the raster operations used in many graphics applications, such as windowing systems. Each processor is associated with 16 Kbits of local memory, which also forms part of the video refresh buffer. This allows the results of computations to be displayed immediately, without need to transfer the results to a separate video buffer.

Connection Machine - [Hillis 85] is a large 128 x 128 bit serial machine. Each processor has 4 Kbits of local memory. The processors are connected in two ways, both in a straightforward four connected mesh, and also by a sophisticated message passing network which takes the form of an order 12 hypercube.

This message passing network, which allows any processor to communicate directly with any other, combined with sophisticated programming tools makes the Connection Machine far more suitable as a general purpose computing machine than other SIMD array machines which are highly optimised for image processing.

These SIMD architectures are in many ways of very similar construction, with the partial exception of the Connection Machine. All are massively parallel fine grain machines, using bit serial processors. All are distributed memory machines, and all (except the Connection Machine) use a two dimensional directly connected mesh interconnection topology.

Whereas the MIMD machines were mostly aimed at fairly general purpose computations, most of the SIMD machines are heavily oriented towards image processing or graphics applications. Most use very simple nearest neighbour directly connected communications networks, which are highly oriented to the localised pixel operations found in low level image processing algorithms.

2.4 An Architecture Optimised for Image Based Applications

A key element involved in determining the correct design for a parallel architecture, must be its intended application area. As mentioned in the introduction the areas of image understanding and image generation are particularly demanding applications, since they exhibit a wide variety of different data and algorithm structures. With this in mind an architecture is proposed which is optimised for exactly these types of applications.

2.4.1 Architectural Features

2.4.1.1 Memory Organisation

Any machine that is intended to perform real time image analysis or generation will have to provide a very large amount of computational power. Such power is not only well beyond that currently available from sequential machines, it is even beyond that achievable by current parallel machines. Clearly then the maximum possible performance is going to be crucial, which implies the use of massive parallelism.

Massive parallelism implies the use of a fully distributed memory architecture. Such a system allows a relatively simple interconnection network to be used, while still allowing systems with very large numbers of processors to be constructed economically.

Distributed memory systems do however rely on all applications being broken down into autonomous modules which can operate with little interaction with other modules which reside on other processors. This requires that a suitable software strategy be used which encourages programmers to arrange their code into modules which require as little communication with other modules as possible.

2.4.1.2 Communication Network

A suitable communications system for such a machine would be a fixed order directly connected network. These provide communication that scales linearly with an increasing number of processors, and equally importantly require resources that also scale linearly with the number of processors, in other words they have a fixed overhead per processor. Also given the two dimensional orthogonal image data to be processed, a square two dimensional mesh would seem appropriate.

2.4.1.3 Control Structure

A fundamental dichotomy is the choice between an SIMD and an MIMD architecture. While SIMD architectures are potentially more efficient than MIMD ones, this is only true for algorithms that map effectively onto SIMD machines. Thus the choice of SIMD or MIMD will depend on the exact algorithms that are needed for the given problem. As mentioned previously image based applications involve a wide variety of algorithms, from globally uniform image operations to highly complex and irregular high level model based symbolic operations.

Low level image operations appear to be an ideal match for SIMD architectures since they exhibit all the qualities necessary for efficient solution on an SIMD machine. They operate on homogeneous arrays of data (images), and most of the operations they perform, such as convolutions, thresholding etc. are globally uniform. Also most of the communications performed are highly local and regular.

In addition most low level image operations use mainly short integer data, making them ideal for implementation on fine grain SIMD machines, arguably the most efficient configuration for this class of machine.

By comparison high level model based operations do not exhibit any of the characteristics associated with efficient SIMD implementation. They operate on model data, which will often be represented as floating point numbers or symbolic relationships, which may have a complex and irregular structure. The algorithms used also tend to be more complex, and perform less regular operations on their data. This would strongly suggest the use of an MIMD architecture for the higher level model based processing.

An optimal solution would therefore appear to be a dual paradigm machine, which consisted of both an SIMD array which performs low level image operations, and an MIMD array which performs the high level model based operations. Based on the arguments put forward above, it would be appropriate to make both arrays distributed memory systems with a fixed order directly connected communications network in a two-dimensional mesh configuration.

2.4.2 Existing Multi-Paradigm Architectures

A number of architectures have been proposed which roughly conform to the arrangement described above. The principal examples of these are the University of Massachusetts Image Understanding Architecture, the Oxford DisPuter, the Purdue PASM and the University of North Carolina Pixel-Planes 5 architecture. Other machines have been proposed, such as Uhr's 2 layered array/net [Uhr 81] but the examples chosen represent those that have had significant detailed design work carried out on them, or have actually been implemented.

2.4.2.1 IUA (Image Understanding Architecture)

The IUA [Weems 89] shown in Figure 2.12 consists of three layers of processor arrays, two of which are MIMD and one of which is SIMD. The bottom layer is a 512 x 512 array of bit-serial SIMD processors known as CAAPP (Content Addressable Array Parallel Processor), in a four connected two dimensional mesh. Above this is a 64 x 64 array of TMS320C25 digital signal processing devices called ICAP (Intermediate Communications Associative Processor), which are configured as an MIMD array. The CAAPP and ICAP layers communicate via 1 GByte of dual ported memory, which consists of groups of 256 KBytes shared between one ICAP chip and 64 CAAPPs.

Control for the CAAPP array is provided by a separate ACU (Array Control Unit) which consists of two parts, a 68020 macro-controller, and a custom micro-controller. CAAPP programs are targeted for the 68020 processor, which performs micro-procedure calls to the micro-controller which performs the actual instruction issue for the CAAPP array at a 10 MHz rate.

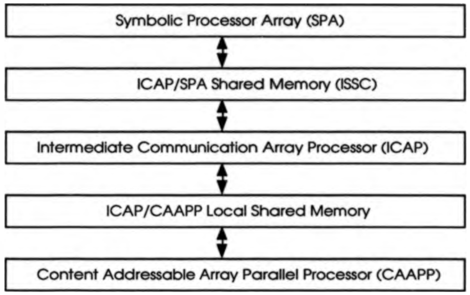

Symbolic Processor Array (SPA)
ICAP/SPA Shared Memory (ISSC)
Intermediate Communication Array Processor (ICAP)
ICAP/CAAPP Local Shared Memory
Content Addressable Array Parallel Processor (CAAPP)

Figure 2.12: Image Understanding Architecture

The top layer of the machine consists of 64 Motorola 68020 processors known as the SPA (Symbolic Processor Array). These are arranged as a shared memory MIMD array. The single block of 512 MBytes of shared memory is both globally shared by the SPA and also locally accessible by the ICAPs. Each ICAP has access to a single 128 KBytes segment of the ISSM (ICAP / SPA Shared Memory). This segment is accessible by the ICAP via an I/O port, rather than being mapped into the memory space of the processor, so it is not strictly shared by the ICAP.

The IUA is intended to tackle demanding real time image analysis problems, and is based on the functional partitioning outlined in the previous section. Thus the CAAPP array is intended to perform low level image processing tasks. The ICAP's are intended to extract numeric data from the results produced by the low level processing, and perform intermediate level processing on that data. The result of this processing would then be sent to the SPA which would perform high level model based processing.

2.4.2.2 The Disputer

The Disputer [Page 87] consists of two parts, a 5x6 MIMD array and a 16x16 SIMD array. The MIMD array is implemented using Inmos Transputers. Communications is via a full crossbar switch allowing arbitrary topologies to be constructed dynamically. The SIMD array is a bit serial design implemented from MSI and bit-slice components. It is arranged as a four connected 2D mesh. Control for the SIMD array is provided by a separate array controller consisting of a T800 Transputer and an AMD bit-slice processor.

The intended application of the Disputer is real time graphics, and it includes a facility to allow selected contents of the SIMD array memory to be continually displayed on a monitor, which effectively eliminates the need for any explicit I/O facilities. Similarly to the IUA, the Disputer performs high level model based functions, such as transformation and projection on the MIMD array, and low level pixel based operation such as scan conversion on the SIMD array.

2.4.2.3 Pixel Planes 5

Pixel Planes 5 [Fuchs 89] is similar in many ways to the Disputer, it too contains two parts, an MIMD array and an SIMD array. The MIMD array is responsible for high level processing while the SIMD array performs low level pixel based processing. However in the Pixel Planes device the SIMD array is a more special purpose device optimised for implementing polygon rendering algorithms.

To achieve this a conventional square SIMD bit serial array is augmented by two adder trees, one connected to each edge of the array. These trees can calculate all the values of Ax, where A is an arbitrary constant and x is the coordinate of the row (or column) of each pixel. Each pixel is connected to row and column busses, along which the values from the adder trees are passed. Each pixel then sums the values being sent along the busses, such that each pixel calculates the expression Ax + By + C. In this way a line can be drawn in the image plane by calculating the coefficients A,B and C for the line, evaluating Ax + By + C on all the processors, and then setting the pixels of all the processors whose result was zero to on.

The Pixel Planes 5 design incorporates a number of these 128x128 SIMD arrays, which are assigned to patches of the image. In addition to this is an array of MIMD processors, which communicate with the SIMD patches via a high speed ring. This ring allows any of the MIMD processors to communicate with any of the SIMD patches. This is particularly important in the graphics applications that the machine is intended for, where polygons processed by the high level processors must be passed on to the appropriate SIMD patch for scan converting. Because the mapping of polygons to patches will vary over time, it is not possible to pre-distribute the polygons in such a way as to avoid the need for global communication, and the ring network allows this to be performed efficiently.

2.4.2.4 PASM

PASM [Siegel 81] takes a radically different approach to the other machines mentioned here. Rather than use separate SIMD and MIMD arrays, it uses a single reconfigurable array, which can operate as either an SIMD or an MIMD array, or a collection of SIMD arrays.

This is achieved using a hierarchical approach where an array of controllers are each connected to a small array of conventional processors in this case Motorola 68010s, each with their own local memory. In SIMD mode the 68010s take their instruction stream from their designated controller, so that each patch of 68010s proceeds in exact instruction level lockstep. In MIMD mode each of the 68010's is allowed to fetch its own instruction stream from its local memory allowing fully autonomous behaviour.

The PASM architecture provides an interesting contrast to the multi-layered machines such as the IUA, but it does not provide the features outlined for an optimised image based architecture. It does not capitalise on the enhanced efficiency possible from fine grain SIMD systems, all its processors are conventional sequential microprocessors as normally found in an MIMD system. When operating as an SIMD machine the local control on each microprocessor is left unused.

Inter-processor communications are carried out using an asynchronous message passing scheme, so no advantage is taken of the higher efficiency possible with synchronous SIMD communications.

The principal advantage of the PASM is the potential flexibility it provides by allowing mixed MIMD and SIMD problems to be implemented, however no programming system is provided which supports this style of problem.

Chapter 3

The Warwick Pyramid Machine

3.1 Introduction

The Warwick Pyramid Machine (WPM) is an architecture optimised for image based applications based on the ideas introduced in the previous chapter. It incorporates both an SIMD array for low level iconic processing and an MIMD array for high level model based processing. The machine can be thought of as a pyramid of processors, with a large fine grain SIMD array at its base, a smaller coarser grain MIMD array above it, and a conventional sequential machine at the apex providing the user interface. This is shown in Figure 3.1.

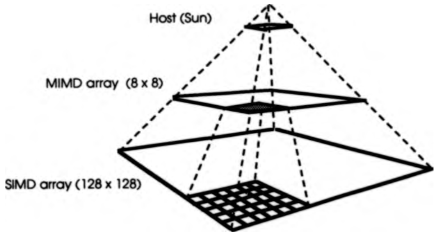


Figure 3.1: The Warwick Pyramid Machine

The aim of the design is to match each stage in the image analysis and image generation process to the most appropriate processor type. In this way it is hoped that the machine should be able to provide an efficient platform for most image based applications, allowing demanding applications to be performed in real-time while minimising the use of hardware resources.

Thus the architecture provides a massively parallel 128x128 fine grain bit-serial distributed memory SIMD array, connected by a two dimensional order two mesh, to perform low level iconic processing. This array is ideally suited to the globally uniform integer operations typical of low level iconic processing.

The SIMD array is connected by multiple high bandwidth paths to an array of coarse grain distributed memory MIMD processors which perform model based symbolic processing. These processors are more suited to the heterogeneous floating point intensive operations associated with the high level processing.

The connections between the two arrays provide not only data transfer from low to high level arrays, essential for the iconic to numeric processing, but also provide a control path from the high to the low level array. Each control path is associated with a 16x16 patch of the SIMD array. This arrangement allows local autonomy for each patch of the iconic array.

The WPM has been designed to make use of industry standard components, to avoid the requirement to develop custom VLSI components. This has allowed a fully functional prototype machine to be constructed and tested. This chapter describes the design and implementation of this prototype machine in detail.

3.2 Design

The building block of the WPM is a cluster. Each cluster can be thought of as an independent SIMD machine, several of which are connected together to form a complete pyramid machine. Any number of clusters may be connected together in any arrangement, allowing arbitrary sized rectangular or square arrays to be constructed as appropriate for any application.

3.2.1 Clusters

A cluster consists of a small SIMD sub-array which performs low level operations, a bit-slice controller which generates the instruction stream for the iconic processors and passes data from the iconic to symbolic processor, and, a single symbolic processor (Figure 3.2).

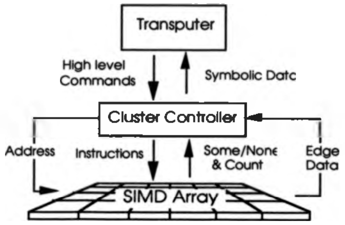


Figure 3.2 A Cluster of the WPM

Each cluster is responsible for a single patch of the image being processed. Ideally the image patch will be the same size as the SIMD sub-array, in which case there will one SIMD processor for each pixel in the image. Larger images can still be processed however, by allocating more than one pixel to each processor.

3.2.1.1 Symbolic Processor

The symbolic processor is responsible for high level feature and model based processing. It is also acts as the coordinator for the cluster, being responsible for initiating any processing that occurs on the other layers, and coordinating communication between neighbouring clusters. Typically the user's application program will run on the symbolic processors, and call pre-written library routines which run on the cluster controller and SIMD array. More details of this system are given in subsequent sections.

3.2.1.2 Controller

The controller provides an interface between the symbolic processor and the SIMD array. This interface is required because the symbolic processor is not capable of providing instructions directly to the SIMD array at the required rate. The SIMD array requires an externally supplied instruction stream at a rate of one instruction per clock cycle, 100ns in this case. The symbolic processor cannot generate instructions at this rate in software, so the cluster controller is necessary to perform this function.

The cluster controller receives high level commands from the symbolic processor, which are broken down into a sequence of low level SIMD array instructions which carry out the requested operation. These low level instructions are supplied to the SIMD array at the full clock rate. Because these higher level operations take many clock cycles to complete the symbolic processor has time to prepare the next operation before the array has completed the previous one. Using this technique the SIMD array can be kept almost fully utilised.

As well as providing the instruction stream for the SIMD array, the controller is also responsible for communicating symbolic data extracted from the array to the symbolic processor. The SIMD array provides facilities for extracting image data based on an associative response mechanism.

3.2.1.3 Iconic Layer

The iconic layer of each cluster acts as a conventional SIMD array, with all of the processing elements executing a single instruction stream generated by the controller. Non global operations are supported by the use of an activity flag which can be used to disable a given subset of the processors within a patch.

The processors of the SIMD iconic layer are connected to their four nearest neighbours, both within each cluster and from cluster to neighbouring cluster.

Associative Response

The array also provides a set of associative facilities which allow the programmer to test each pixel in the image for a logical condition. Those pixels which satisfy the condition are flagged as responders. The associative hardware provides facilities for the programmer to test for the presence or absence of responders, via a some/none line, or count the number of responders using a count network. The SIMD array hardware also allows the programmer to determine the coordinates of the responders, and read the image data from those coordinates using the array's edge register.

3.2.1.4 Cluster Interconnection

Adjacent clusters are connected at three levels, the SIMD array level, the controller level and the symbolic level. They can be connected in two different modes, cluster mode and array mode. In cluster mode each cluster operates independently, and can communicate with adjacent clusters only at the symbolic level. The boundaries of the iconic patch are wrapped round to form a torus, so that any shift operations cause the patch data to wrap round from east to west and north to south. Array mode allows communication at both the symbolic and iconic level. In this mode the edges of each SIMD patch are connected to the edge of the adjacent patch, and the boundaries of the whole array are wrapped round to form a conventional large SIMD array.

The cluster controllers are also connected together to allow them to synchronise. This is necessary because the SIMD processors, which they control, must be synchronised to allow them to communicate with adjacent clusters. Since the SIMD processors are locked to the instructions provided by the controller the controllers themselves must be synchronised.

This is achieved using a hardware semaphore between adjacent cluster controllers, on which the controllers perform wait operations. The first controller to perform a wait operation is suspended until the other controller also performs a wait, at which point both are allowed to proceed.

When two clusters wish to communicate, the sending controller, first moves a row of array data to its communication outputs, and then performs a wait operation on the semaphore. The receiving cluster first performs a wait on the semaphore, and then reads the data from its array inputs.

The flexible software controlled synchronisation of clusters allows many different configurations to be used. If all the clusters synchronise, then the whole iconic array acts as a single conventional SIMD array. Alternatively several subsets of the clusters may synchronise, forming a collection of SIMD arrays. This can be done dynamically under software control and allows the machine to match itself to the current structure of the data.

3.2.2 Multiple-SIMD Operation

A particularly important aspect of the architecture is its Multi-SIMD organisation, where the SIMD array is divided into independently controllable clusters. Other machines such as the IUA and the Disputer which combine MIMD and SIMD parallelism but which do not provide this Multi-SIMD organisation suffer from two problems which the Multi-SIMD approach overcomes.

For any 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 other MIMD/SIMD machines the single SIMD array controller becomes a bottleneck for communication with the MIMD array. The WPM's Multi-SIMD design uses multiple controllers allowing 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.

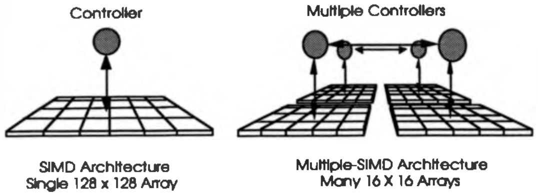


Figure 3.3: SIMD vs Multi-SIMD

The other 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 10x10 pixel polygon, then less than 1% of the processors in the array would be used, the others would be idle. Put another way the array would be operating at only 1% of peak performance.

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. Taking the previous example, each cluster would be able to independently render a different polygon. This would give up to a factor of 64 (the number of clusters) improvement in performance. In chapter seven this application is explored in more detail.

3.3 Implementation

The VLSI Architecture group has produced a working prototype of the Pyramid Machine. This prototype has been constructed using existing VLSI components which have been integrated into a system level design which provides the desired functionality. It should be stressed however that this prototype does not represent an optimal implementation of the Pyramid Architecture, and developments are underway to employ improved technology in subsequent implementations.

The current prototype uses a SUN workstation as the host and a Transputer as the symbolic processor. The SIMD array is implemented using AMT DAP chips, while the cluster controller is a custom built bit-slice design. A full description of the current implementation is given below.

3.3.1 Host Machine

The host machine used in our current prototype is a SUN 3/280. This machine contains a VME based Transputer interface card which communicates with the symbolic processor array via Inmos links.

The job of the host machine is to provide the user interface for the software development tools and the applications software. To this end a library has been written using the X Windows graphics system, which allows applications programs running on the WPM to display graphics and provide a user friendly interface. This and other software which runs on the SUN is written in C and should be able to run on any suitable UNIX workstation, which can accept a Transputer interface card.

3.3.2 Symbolic Processor

The processor used for the symbolic layer is the INMOS T800 Transputer. This processor was chosen because it is specifically designed to support distributed memory MIMD multiprocessing. Each Transputer contains a powerful 32 bit integer and floating point processor, 4 KBytes of fast static memory, and four DMA driven 20 Mbit/s serial links used for inter-processor communications. It also contains a micro-coded multi-tasking system which supports many time shared processes using fast context switching, and high speed inter-process communications.

The inter-process communication mechanism is based on channels, which are a communications and synchronisation primitive derived from the communications model used in Hoare's CSP programming system [Hoare 82]. Channels provide a unidirectional point to point communication mechanism with synchronisation based on the rendezvous principle. When a process writes to a channel it is suspended until the receiving process reads from the channel. Conversely, if a process reads from a channel it will be suspended until some data is written into the channel by another process.

Channels can exist either between two processes running on the same Transputer, in which case they are implemented by in core copies, or between processes running on adjacent Transputers, in which case they are implemented using the DMA driven links. The Transputer instruction set includes a number of special communications instructions which handle channel communications. These instructions, which are implemented in micro-code, select the appropriate type of communication, either in core or via links, and perform all the necessary synchronisation and scheduling completely transparently. In this way the same piece of code can perform communications without regard to the location of the destination.

The combination of the capability to run multiple processes on each Transputer, and the ability for these processes to communicate transparently, regardless of whether they are on the same processor or not, in principle allows programs to be written which are largely independent of the number of processors on which they are run. For example if a program were designed to run on ten Transputers, with one process per processor, it would run equally well on five with two processes per processor. The processes would not be aware of any change, and would not even have to be recompiled.

In practice however this ideal is seldom realised because of the fixed topology inherent in the channel system. A channel is a static entity connecting one process to one other process. Thus a program written using channels must also have a fixed communications topology. Since only one channel can be associated with each link, this fixed topology must map on to the topology of the hardware used. This causes severe portability problems since programs will in general only run on a machine with the same topology as the one for which they were written, and it is this that prevents programs from being mapped to different machines.

Thus although the channels implemented by the Transputer provide an excellent primitive communications and synchronisation mechanism it is often desirable to provide a layer of abstraction built on top of channels, which is topology independent. A number of systems have been implemented which do this such as Helios [Perihelion 89], and a similar system has been developed for the Pyramid Machine, details of which are given in chapter six.

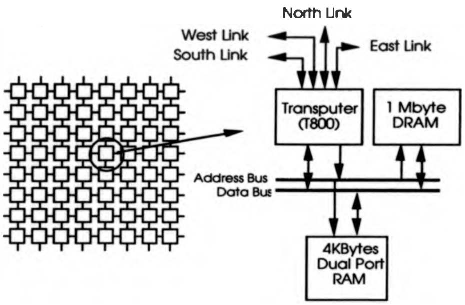


Figure 3.4 : Symbolic Processor Array

From a hardware perspective the Transputer is particularly suited to constructing coarse grain MIMD arrays, where any hardware required to implement each processing node must be replicated many times. The

Transputer includes an on chip memory controller, which allows it to be connected directly to a dynamic RAM array using no other chips except buffering. This dramatically reduces the chip count for a complete system.

The Inmos links used by the Transputer provide a convenient way of connecting multiple Transputers together. Again no extra interface chips are required except for buffers, and since the links are serial they require only four wires to provide bidirectional communications at 20 MBits/s. This greatly eases the problem of wiring a large array.

By utilising the Transputer it has been possible to construct a fully working MIMD array using surprisingly little hardware, and making use of extensive existing software support. The Transputer array which has been constructed is shown in figure 3.4. It consists of an eight by eight array of Transputers connected in a four connected mesh with one of the Transputer's links connected to each of its four neighbours.

Each node of the array consists of a Transputer connected to 1 Megabyte of dynamic RAM and 4 Kbytes of dual port static RAM which is shared between the Transputer and the cluster controller. The dynamic RAM is used to store all the code and data for the Transputer, while the dual ported RAM is used to communicate with the cluster controller. This communication is in the form of commands sent to the cluster controller and symbolic data received from it.

It should be noted that although a full eight by eight array of Transputers has been constructed, only two are currently equipped with dual ported memory, since only two cluster controllers have been constructed.

3.3.3 Cluster Controller

The cluster controller lies at the heart of the Pyramid Machine, and it is the most complex part of the design. Its function is to provide the instruction stream for the SIMD array at the full clock rate, which in the current design is

10 MHz, or one instruction per 100ns. Each SIMD instruction consists of an opcode, which specifies the operation to be performed, and an address, which determines which memory location the operation is to be applied to.

If the controller is to achieve one instruction per cycle, it must not only be able to generate opcodes at one per cycle it must also calculate the operand address during the same cycle. A typical piece of code, such as a multi-bit add operation, would require three different addresses to be calculated, one for each source operand and one for the destination operand. Each of these addresses would need to be incremented once for each iteration of the main loop.

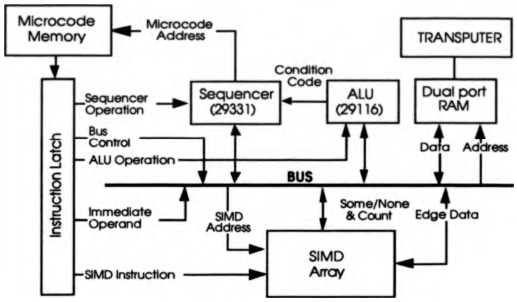


Figure 3.5: Cluster Controller

In addition to address calculations, the multi-bit add routine must also keep track of its loop counter. This will require at least one arithmetic operation per loop to increment or decrement the counter, plus another to test the value against a predetermined limit. This test will then be used to perform a conditional branch. If the SIMD array is to be kept fully occupied all these operations must be performed without reducing the rate of instruction generation.

To achieve this a custom bit-slice design has been used, a block diagram of which is shown in figure 3.5. The main components are a sequencer (AMD 29331) which fetches the micro-code instructions, a scalar ALU (AMD 29116) which performs address calculations, 16K x 64 bits of microcode memory, 4K Bytes of dual ported RAM used to communicate with the symbolic processor, and a central data bus which links the various components together.

In operation the cluster controller acts in a similar way to a conventional scalar processor. The sequencer generates the address of the next micro-instruction, which is fetched and latched in the instruction latch. The micro-instruction consists of a number of fields which specify the operation of the various functional units in the controller. The format of the micro-code word is shown below.

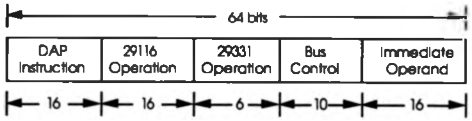


Figure 3.6: Micro-code word

The micro-code word is fully horizontal, that is every instruction contains one field for each of the functional units in the controller. This allows one operation to be performed by each unit every cycle. In addition every instruction includes a field which contains an SIMD array instruction. This field is sent directly to the SIMD array and provides the opcode, but not the operand address for the array instruction.

The purpose of this wide micro-code word is to achieve the design aim of generating the SIMD instruction stream including operand addresses at the full clock rate. By providing an instruction for all of the functional units every cycle, address calculations and loop operations can be overlapped with instruction generation. To see how this overlapping is performed it is necessary to understand the operation of each of the functional units that are controlled by the microcode.

3.3.3.1 Sequencer

The sequencer is responsible for generating the address of the next micro-code instruction. The device used is the AMD 29331 the internal structure of which is shown in figure 3.7.

It consists of four main parts, the program counter, the stack, the loop counter and an address multiplexer. The output of the multiplexer is used as the address of the next micro-code word. In normal operation the multiplexer will select the current value of the program counter plus one as input and this value will be fed both to the micro-code memory and back to the program counter which will therefore be incremented.

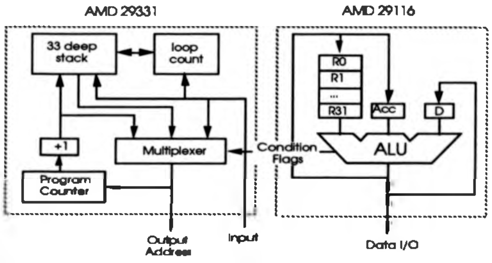


Figure 3.7: Bit Slice Sequencer and ALU

The multiplexer may alternatively select the next address to be provided by the direct address input. This comes from the controller bus, and the value on it will depend on the bus control micro-code field, but will usually be the immediate operand field. This has the effect of branching to the address specified by the immediate operand. Branching operations may be performed conditionally based on the condition flags that are generated by the ALU.

The final alternative is that the address be supplied by the on chip stack. This stack is used to perform subroutine calls. When the sequencer performs a subroutine call the current program counter address is pushed onto the stack. When a return operation is performed the address is popped back off the stack and used as the next instruction address.

The stack can also be used in conjunction with the loop counter to perform variable iteration loops. To start a loop a value is loaded into the loop counter using the direct input lines from the controller bus. When the loop is entered the current program counter address is pushed onto the stack. At the end of the loop the counter is decremented, and if not zero the loop address is read from the stack and used as the next instruction address. If zero the address is popped off the stack and execution continues.

This looping facility is particularly useful since it allows the loop counter incrementing and testing to be overlapped with the array instruction generation. This allows the controller to generate instructions at the full clock rate throughout the execution of a loop.

3.3.3.2 Scalar ALU

The ALU's principal purpose is for address generation, although it can also be used to perform general scalar operations if required. The ALU used is the AMD 29116 shown in figure 3.7. The main parts of the device are the 16 bit arithmetic and logic unit itself, the 32 register register-file and the accumulator.

The ALU has three inputs, one from the register file one from the accumulator and one from the D register which hold values inputted from the controller bus. The ALU can perform a variety of arithmetic and logical operations on either one or two of its inputs, and write the result to a specified destination. Each arithmetic operation generates a number of condition flags which are passed to the sequencer to be used in conditional branch operations.

An important aspect of its behaviour is that it allows a result to be sent both to its output (which is connected to the controller bus) and to a register. This feature is useful for address generation, since it allows an address to be read from a register, supplied to the bus and incremented and stored back into the register in a single cycle.

3.3.3.3 Central Bus

The central data bus connects all the parts of the controller together. All devices connected to the bus are given an address. The devices include all the bit slice units as well as a number of simple control registers. The bus control logic allows any one of the devices to put data onto the bus, and allows that data to be written to one of the devices. It is not necessary to specifically write to the ALU or sequencer as these can read the value that is currently on the bus.

The registers on the bus are as follows:-

Register	Source	Destination
Immediate operand	yes	-
Sequencer	yes	implicit
ALU	yes	implicit
Dual Port RAM Address	-	yes
Dual Port RAM data	yes	yes
Edge Register	yes	yes
Row/Column address	-	yes
DAP memory plane address	-	yes
Count	yes	-
Cluster mode control	-	yes

The cluster mode control determines whether the DAP operates in array mode, where the communication lines are fed to the neighbouring clusters, or cluster mode where the lines are wrapped round to form a torus within each cluster.

The Edge register, row/column address, DAP memory plane address and count registers are discussed in detail in the following section describing the SIMD iconic layer.

3.3.4 Iconic Array

The iconic processor array lies at the base of the cluster. It consists of a 16x16 bit serial SIMD array implemented using AMT Distributed Array Processor (DAP) chips. The DAP chip integrates 64 bit serial processors onto a single device arranged as an 8x8 four connected grid. Four such DAP chips are required to implement one of the 16x16 SIMD patches associated with each cluster. The basic structure of one DAP chip with its memory is shown in figure 3.8.

All operations performed by the DAP operate over a complete 16x16 array of memory, so it is helpful to consider the DAP memory and registers as planes. The DAP chip itself contains a processor element (PE) array and three register planes A, C and Q. Memory for the array is external to the chip, and consists of 32K planes of fast static memory, a total of 1 MegaByte of memory per cluster. The DAP chip provides a full 64 bit wide memory bus (one bit per processor) to interface to this external memory, allowing a complete plane to be accessed in a single cycle.

Each processor in the array is connected to each of its four neighbours which allows planes to be shifted horizontally and vertically within the array. The edges of the array may either be connected to the neighbouring array, as used in array mode, or wrapped round to meet the opposite side of the same array, to form a torus, as used in cluster mode.

In addition to this every processor is connected to two busses, one row line and one column line. These busses run vertically and horizontally across the array, and connect to two edge registers. These registers are mapped into the register space of the cluster controller, and can be used in two modes. In the first mode they can be used to allow a complete row or column of data to be read from the array, under control of the Row Select address register. This address specifies which row or column is presented to the row or column busses.

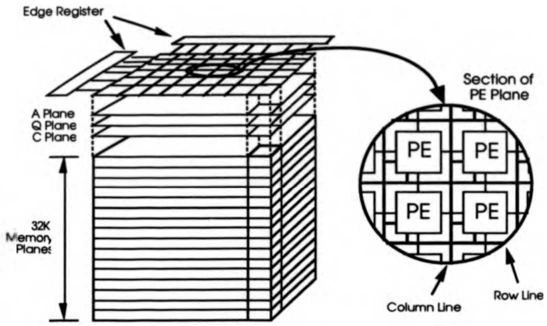


Figure 3.8: AMT DAP SIMD Array

Alternatively the row and column lines can be used to AND together all the values placed on them and present the result to the edge registers. This mode has a number of uses for extracting data from the array. It can be used to implement a some/none associative response over the whole array, by arranging for any responders to present a zero value on the row lines, and non responders to present a one. The cluster controller can then read the edge register, which will contain all ones if there are no responders and some other value if there are responders.

It can also be used to determine the position of a particular responder, by reading both the row and column edge register, and determining the intersection of the two zero bit positions. Thus if, for example, the responder were in cell (3,4) then bit 3 of the row register and bit 4 of the column register would be zero.

3.3.4.1 Processing Element

Each cluster also includes a mechanism in hardware for counting the number of responders in only a couple of clock cycles. This is achieved using a fast adder tree which is connected to the memory bus of the DAP chips, allowing it to count the number of active bits in any memory access. This count facility has been shown to be extremely useful for implementing associative algorithms such as calculating a histogram [Kerbyson 91].

A simplified block diagram of the processing element used in the DAP chip is shown in figure 3.9. The principal components are a one bit full adder, which performs all the arithmetic operations, three single bit registers used to store intermediate values, and two multiplexers which select the source and destination for the operands of each operation.

The programmer's model of the DAP PE does not lend itself to easy explanation. Its design dates back to the ICL DAP developed in 1972 when it was constructed from small scale TTL logic [Parkinson 90]. The design was optimised to require as few TTL packages as possible, and as a result the elegance of the programmer's model was relatively low priority. The current DAP which is implemented as a gate array retains the same PE design in order to maintain compatibility with the installed software base, particularly the various compilers that were developed for the ICL DAP.

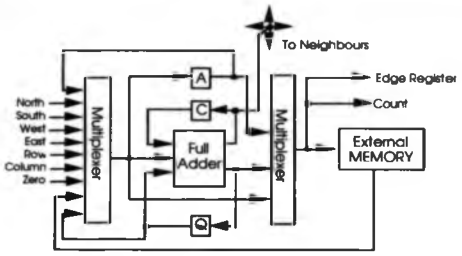


Figure 3.9: DAP PE functional diagram.

The ALU of the PE is provided by a single bit full adder, which has three fixed inputs, one from the input multiplexer, one from the Q register and one from the C register. The two outputs of the adder are also fixed. The sum output goes to the Q register and the output multiplexer while the carry output goes to the C register and is also broadcast to all four of the PE's neighbours.

The A register has a dual function, it can either be used as a simple accumulator, whereby new values can either be ANDed with the value already stored in it, or simply stored directly in it, or it can act as an activity control register, allowing the programmer to perform writes and additions to memory which are conditional on the value in the A register. This is the only facility provided to allow PE's to act in any way autonomously.

Although the connectivity of the PE is fixed, the programmer can specify which registers are written to in each instruction, and can generally specify that either the value stored in a register, or zero be used as the output for each register. It is also possible for the programmer to optionally invert the value provided by the input multiplexer. Unfortunately this level of control is not available in all combinations, since the instruction encoding causes certain clashes, but it does work in most cases.

The input and output multiplexers determine the routing of data through the PE. The input multiplexer determines one of the input operands, which can be either the Q or A register, one of the neighbouring PE's, the currently addressed external memory plane or the value zero. The output multiplexer determines what value is written to external memory. This can be one of either the A register, the input multiplexer's output, or the sum output of the full adder.

These multiplexers are not completely under the control of the programmer. Instead a fixed number of predetermined combinations of input and output multiplexer settings is provided. There are fourteen of these 'instruction groups' as they are referred to, using the terminology of the APAL language, AMT's assembler. These groups are shown below.

Group	Description	Input Multiplexer	Output Multiplexer
0	Memory to register	memory	-
1	XOR of memory with bit of edge register to PE	memory	-
2	Memory to edge register	memory	input mux
3	Edge register to PE	column	-
4	Zero to PE	zero	input mux
5	Bit of edge register to PE	zero	-
6	Edge register to memory	row	input mux
7	Edge register to PE	row	-
8	Q to PE/store/edge register	Q	Q
9	A to PE/store/edge register	A	input mux
10	Conditional add to store	memory	input mux
11	Conditional write to store	zero	input mux
12	Shift Q to nearest neighbour	neighbour	-
13	Ripple add (in any direction)	neighbour	-

All the instructions for the DAP are provided by the cluster controller at a rate of one instruction per clock cycle. Each instruction consists of an opcode, which specifies the instruction group and register operation described above and an address, which specifies the memory plane the operation acts on where appropriate. Only a single address is required since the DAP can only read or write one memory plane in any given cycle. Because of this all instructions that require two memory references (those in groups 2,10 and 11) require two clock cycles to execute. All other instructions can be performed in a single cycle.

3.3.5 Cluster Controller Programming Model

There are four programmable devices of interest, the ALU, the sequencer, the control bus and the DAP. Each of these has its own field in the microcode word, so that each can perform one operation in each cycle. In addition to this there is an optional immediate operand which can be supplied for use by any instruction.

To program these devices an assembler called CLASS (CLuster ASSembler) has been developed, which provides direct access to each field of the microcode word. Like the microcode each CLASS instruction has five fields one for each device plus an immediate operand. Within each field CLASS uses the manufacturers recommended opcodes, so that the DAP field is essentially the same as the APAL language, while the AMD parts use opcodes described in each devices user manual. The control bus uses a bespoke instruction format.

This approach allows very efficient programming of the cluster controller, and allows us to make use of existing documentation, but it does lead to a somewhat opaque instruction format. As an example take the following CLASS code.

```
DJMP_S;   INC SORR R10;        SQ;         PEADDR=ALU
```

The DJMP_S is a sequencer instruction which decrements the loop count and jumps to the address on the stack if it is not zero. The INC SORR R10 is an ALU instruction which increments register 10. SQ is a DAP instruction which assigns the Q plane to the store plane specified by the current PE memory address. This is set by the final field PEADDR=ALU, which sets the PE memory address to be the value output by the ALU, which is the result of the increment operation performed in the ALU field.

To examine this more closely a more full example will be presented , but instead of using the actual CLASS mnemonics which are somewhat obscure pseudocode notation will be used. This pseudocode uses a Pascal like notation where registers of the various devices are denoted by a three letter code for the device (ALU, SEQ or DAP) followed by the name of the register which corresponds to the descriptions given above.

The following example is an eight bit addition:

```
          // Set up the addresses of the two sources and the
          // destination in ALU registers. Values are minus one
          // because they are pre-incremented in loop
-----------------------------------------------------------------
ALU         ALU_R1 = source1 - 1
-----------------------------------------------------------------
ALU         ALU_R2 = source2 - 1
-----------------------------------------------------------------
ALU         ALU_R3 = destination - 1
-----------------------------------------------------------------
          // Clear carry flag ready for addition
DAP         DAP_C = 0

          // Set up loop return address and counter
SEQ         SEQ_push(here + 1) ; SEQ_count = number_of_bits
-----------------------------------------------------------------
          // START OF MAIN LOOP
          // Increment first source address
ALU         ALU_R1 = ALU_R1 + 1

          // Use source address as address in DAP memory
BUS         PE_address = ALU_R1

          // Load source data into DAP Q register
DAP         DAP_Q = memory[PE_address]
-----------------------------------------------------------------
          // Increment second source address and set as
          // address for DAP
ALU         ALU_R2 = ALU_R2 + 1
BUS         PE_address = ALU_R2

          // Perform addition with carry putting result into Q
DAP         DAP_Q = DAP_Q + DAP_C + memory[PE_address]
-----------------------------------------------------------------
          // Increment destination and use as DAP address
ALU         ALU_R3 = ALU_R3 + 1
BUS         PE_address = ALU_R3

          // Store result into destination
DAP         memory[PE_address] = DAP_Q
```

```
      // Decrement loop counter
      // If non zero repeat main loop
SEQ     SEQ_count = SEQ_count - 1
        if SEQ_count != 0 then goto SEQ_pop()
```

In the routine shown above the instructions are organised into groups (between pairs of horizontal lines), such that each group performs at most one operation on each of the functional units. Each instruction is preceded by a code which specifies the functional unit it applies to, thus within each group all the codes are different.

Each group represents a single cluster controller microcode word, and executes in a single cycle. Notice in particular that the main loop formed by the last three groups contains a DAP operation in every group, so that the DAP is kept fully supplied with instructions at one per cycle, even though three operand addresses and one loop counter are calculated for each iteration of the loop. This was one of the principal design aims, and it has been achieved very satisfactorily.

As can be seen from this example there is significant interaction between the various fields of each instruction. Managing this interaction is the job of the programmer, which can considerably complicate the programming task. This is particularly true for the numerous special cases such as being able to supply an immediate operand to the ALU and use the outputted value on the bus in the same instruction, or having to repeat PE instructions for operations that take multiple cycles, or having to wait three cycles before edge register data is ready and so on.

Using CLASS the applications programmer must be fully aware of this complexity, and this reduces the programmers efficiency considerably. To overcome this a software strategy based on the use of a high level language has been devised, whereby only a simple set of primitive routines needs to be written in CLASS. Once this is done the applications programmer will not be required to use CLASS at all, although the option will always exist for those applications which require optimal performance.

3.4 Conclusions

The Warwick Pyramid Machine is designed to provide an architecture optimised for image based applications. This is achieved through the use of a heterogeneous multi-paradigm arrangement where a variety of parallel processing resources are made available to allow each of the varied types of algorithms associated with this class of application to be mapped on to the type of processor most appropriate for it.

The main components of the machine are a massively parallel fine grain SIMD array matched to low level iconic processing, and a coarse grain MIMD array matched to high level model based processing. Of particular importance is the manner in which these two arrays are interconnected, and the manner in which control is distributed.

The machine is built up from independent clusters, where each cluster consists of a small section of the SIMD array, a controller and a single processor from the MIMD array. This arrangement allows each SIMD patch to be provided an independent instruction stream, and provides an independent path for data to be passed from SIMD to MIMD arrays.

The multiparadigm approach has been taken by a number of other architectures, most notably the IUA, the Disputer and the Pixel Planes 5 machines. All these are based on the same basic premise of matching fine grain SIMD arrays to pixel level processing and coarse grain MIMD arrays to model based processing. However a number of fundamental differences exist in the various approaches that have been taken.

Most notably the Multi-SIMD arrangement of the WPM contrasts with both the Disputer and the IUA, both of which rely on a single global array controller for their SIMD arrays, (although the IUA does provide multiple communications paths). This restricts the class of efficiently implementable algorithms to those which operate uniformly over the whole image. Also in the case of the Disputer the single communication path forms a serious potential bottleneck, although in the current very small implementation this is probably acceptable.

The WPM by comparison can efficiently implement algorithms which exhibit a high degree of local autonomy. Examples of such algorithms include recognition systems, such as described by Francis [Francis 90] and image generation systems such as described in chapter seven. Either of these typical example applications would be difficult to implement efficiently on a non Multi-SIMD architectures.

The Pixel Planes 5 architecture does provide for Multi-SIMD operation, which it uses to implement an image generation scheme similar to that outlined in chapter seven. However the Pixel Planes architecture is highly optimised for a very specific class of image generation systems and lacks support for more general algorithms, such as inter processor communications at the SIMD level, provision for associative response, and communication from the SIMD to the MIMD array. However it is nonetheless interesting to note that such a highly optimised architecture should include Multi-SIMD operation, and this seems to confirm that such an arrangement is very desirable for a number of real applications.

Possibly as important as the architecture is its chosen implementation. The WPM is based on industry standard components, namely the Transputer, the AMT DAP and AMD bitslice components. This has allowed a working scaled down prototype of the machine to be successfully constructed and tested. Apart from the WPM only the Disputer, also based on industry standard components, has actually been implemented to the point of having a working prototype.

Chapter 4

Programming Parallel Architectures

4.1 Introduction

One of the most important challenges for the designers of parallel systems, has been finding effective ways of programming them. Conventional programming languages are usually inherently sequential in nature, which reflects the sequential architectures on which they are intended to run. In a sequential language a programmer expresses an algorithm as a sequential series of operations, with each operation being completed before the next begins such that one operation is being performed at any one time. Parallel architectures, on the other hand, allow several operations to be carried out at the same time. To take advantage of this there must be some way of determining which operations may be performed in parallel and which may not.

There are essentially two approaches to this problem, one is to make use of implicit parallelism, the other to allow explicit parallelism. Systems based on implicit parallelism use languages which do not contain any mechanism to directly express parallelism. These systems are often based on conventional sequential languages, most commonly Fortran, and sophisticated compiler technology which can determine which operations may be performed in parallel while preserving the semantics of the original program.

Explicit parallel programming systems are based on new or extended programming languages which allow the programmer to directly express which operations are to be performed in parallel.

4.2 Implicit Parallelism

Systems based on implicit parallelism attempt to extract parallelism intrinsically present in an algorithm. Even if an algorithm is expressed in a sequential language it is often possible to analyse the dependencies within the program and work out which operations must be performed in sequence, and which can be performed in parallel. Implicit parallel systems are generally based on either an existing conventional sequential language, or on a pure functional language.

4.2.1 Parallelising Sequential Programs

The use of an existing sequential language has a number of advantages. Firstly it allows so called dusty deck programs to take advantage of parallel hardware. Dusty deck, is a reference to punched cards, and refers to existing programs, which are to be run without modification. It also removes the necessity for programmers already familiar with an existing sequential language to learn a new one. This is particularly important for those programmers whose principal job is not actually programming.

The most common language used for this purpose is Fortran, which is currently the most popular language used for scientific and engineering work, where parallel architectures have found most application up to now. All current super-computers provide versions of Fortran which can make use of vector parallelism. Super-computers such as the Cray and Alliant etc implement vector instructions which perform an operation on a vector (or array) of numbers in parallel and provide Fortran compilers which can generate these vector instructions in place of the inner loops of array routines.

For example consider the following code:-

```
do 10 i=1,100
        A(i) = B(i) + C(i)
10 continue
```

This loop adds together two arrays and places the result in a third array. On a conventional architecture this would be implemented as a sequence of one hundred loops each of which would add one element of each array together and place the result in an element of the third array. However it is clear that the calculations performed in each iteration of the loop are completely independent of the calculations in the previous loop, and so it would be possible to calculate all the one hundred additions in parallel.

A vectorising Fortran compiler would replace this loop with a single vector instruction, which adds two vectors of length one hundred together and places the result in a third. This is quite straightforward in this very simple case, but in practice programs are rather more complex than this, and compilers need to be very sophisticated to be able to determine which loops can and cannot be vectorised. Consider the following simple example.

```
do 10 i=1,100
        A(i+1) = A(i) + B(i)
10 continue
```

In this example the result of one iteration of the loop does depend on the result of the previous iteration, because the value A(i) in the loop is the value A(i+1) from the previous loop. This means it is not possible for the compiler to use a vector instruction in place of the loop, and so no use can be made of the parallel hardware of the machine.

So called vectorising compilers are extremely popular, and form the foundation for the vast majority of scientific supercomputing. They are however currently limited to vector processors, and comparatively little progress has been made in automatic parallelisation of programs for other architectures such as distributed memory MIMD systems.

Even on vector machines a state of the art parallelising compiler can only generate parallel code for certain specific cases, as illustrated above, which involve inner loops that operate on array data with no inter loop dependencies. Code of this kind is however very common in certain scientific applications, and it is in exactly these areas where supercomputers have found much application.

In many more complex programs the ability of the compilers to extract possible parallelism from the programs becomes limited. It is not uncommon for current vectorising compilers to be almost totally unable to vectorise normal scientific applications [HenPat 90]. This is often not because the programs do not contain any parallelism, but because the compiler is unable to extract what parallelism there is [Kuck 74].

Compilers are unable to extract parallelism when a program does not conform to the structure that the compiler is expecting. This may be because the algorithm used is not data parallel, that is it does not perform the same operation on many pieces of data, and therefore cannot use the vector operations of vector machines. Alternatively it may be because the algorithm, although data parallel, is expressed in a manner too complex for the compiler to recognise and thus vectorise.

In the latter case reorganisation of the program will often allow a version to be produced which is simple enough for the compiler to vectorise, but in the former case it will be necessary to rewrite the program completely using a completely different algorithm which achieves the same overall result, but which is suitable for vectorisation.

Unfortunately the fact that programs have to be rewritten rather defeats one of the main advantages of the parallelising compiler approach, namely that existing sequential programs written without parallelism in mind could take advantage of parallel hardware without modification.

It was also assumed that programmers already familiar with the sequential language used would not need to be retrained, but this is also not true. The task of designing an algorithm which can be parallelised, is equivalent to designing a parallel algorithm, since in both cases it is necessary to organise the computation such that multiple operations can proceed in parallel. Therefore it will still be necessary for the programmers to be fully conversant with parallel algorithm design. In addition the programmer must be able to express the parallel algorithm in a sequential language, taking into account any peculiarities in the particular compiler used.

The approach outlined above is exactly the reverse of that originally intended. Instead of the compiler taking a sequential program and converting it into a parallel one, the programmer takes a parallel program and converts it to a sequential one. This causes a number of problems due to the semantic gap it introduces. The semantic gap is the difference between the algorithm as understood by the programmer and the realisation of the algorithm in the form of code. If the language the algorithm is expressed in does not allow the concepts that the programmer was working with to be directly expressed, then there is said to be a semantic gap.

For example to take advantage of a vector machine a programmer may deliberately design an algorithm so that all the calculations on a particular array could proceed in parallel. If this were coded in a sequential language there would be no way of expressing that the operation should proceed in parallel, it would be the task of the compiler to determine that a vector instruction could be used. Subsequent programmers reading the sequential version of the algorithm would be given no clues as to the motivation behind this particular formulation of the problem, particularly if they were not familiar with the peculiarities of the machine and compiler for which the program was written.

This leads on to another problem, namely that to be able to express the algorithm in a sequential form the programmer must be aware of the limitations of the particular compiler which is to be used. There is no guarantee that one compiler will be able to parallelise the same code as another. This clearly has implications for portability of code, as although in general a program will still work on another machine it may not take full advantage of the available hardware.

Automatic parallelising compilers are a useful tool for users who have a large quantity of existing code which it is not feasible to rewrite and which falls into the general data parallel form. However as a general solution to the parallel programming problem they seem to be rather limited. They lack the expressiveness necessary to allow the programmer to capture the meaning of the algorithm being used.

4.2.2 Parallelising Functional Programs

The main difficulty associated with parallelising sequential programs is deducing the interdependency which determines which parts can be performed in parallel. All conventional sequential languages are based on the assumption that each statement in the program is completed before the next begins. A parallelising compiler needs to perform multiple statements in parallel, and therefore must break this underlying assumption. To do this while maintaining the correct semantics of the program, the compiler must determine whether each statement is in any way dependent on preceding statements, and ensure that if it is, those statements are completed before the next begins.

As discussed in the previous section determining interdependency is a difficult problem, and while it has been partially solved for certain cases, it remains problematic. An alternative approach is to make use of languages which do not contain any assumed sequentiality. The most likely candidate for this are the pure functional or declarative programming languages. These are not parallel languages, but they are also not sequential in the sense that they do not express algorithms as a sequence of statements where each is performed one after another.

A program in a pure functional language consists of a number of function definitions. Each function definition consists of a single expression, as opposed to a sequence of expressions as would be the case in a sequential language. An expression consists of a single function invocation whose arguments may be other function invocations or constants.

To illustrate this, an example functional program which sums the values stored in a tree is given below. The program consists of a single recursive function, which takes a node as its argument. If the node is an atom, that is a leaf node of the tree, it will return the value stored in it, otherwise it will return the sum of the two branches, whose values are determined by a recursive call to the sum function.

An important point to note is the difference between an if statement as used in a conventional language, and the functional if statement. In a conventional sequential language an if statement conditionally executes one of the two alternative code segments, whereas in the functional case it evaluates one of the two alternative functions, and returns the result.

```
        sum(n)
        {
                if (is_atom(n) ) {
                        n.value ;
                }
                else{
->                      add(sum(n.left),sum(n.right)) ;
                }
        }
```

In a conventional language a function may have side effects, which means it may alter the global state of the program. Also the value returned by a function may be dependent on the state of the program. This interdependence of functions via the global state of the program, means that function evaluation must occur in strict sequential order to ensure the correct functioning of the program.

Functional languages do not have global state, and do not allow side effects. A function will always return the same result, if called with the same parameters, no matter where or when in a program it is called. This property is known as referential transparency, and it guarantees that functions are only dependent on the values of their parameters. This in turn implies that functions may be evaluated in any order provided that their parameters are calculated first (even this restriction may be relaxed in the case of lazy functional languages).

Knowing that functions are only dependent on their parameters allows us to construct a dependency graph for a program, which has a simple tree structure. The nodes of the graph represent the operations to be performed, while the arcs represent dependencies. In this case each node will represent a function invocation, while each arc will represent one parameter. This graph is constructed by a process of repeated reduction. A reduction step involves replacing a function invocation with the corresponding function definition, with the actual parameters substituted for the formal parameters.

For example if we have two function definitions

```
    add(a,b) = a + b

    mult(a,b) = a * b
```

and an expression

```
    a = add(mult(2,3),mult(4,5))
```

one reduction step gives

```
    a = add( 2 * 3 , 4 * 5 )
```

another gives

```
    a = (2 * 3 ) + (4 * 5)
```

which represented as a graph is

```
          +
        /   \
      *       *
    /   \   /   \
   2     3 4     5
```

The form of the graph is a binary tree, so the two subgraphs associated with each node are disjoint, that is they have no interdependency, and can therefore be calculated in any order, or indeed at the same time. In this simple example above, the two multiplications may be calculated in parallel.

Taking the more complex example given earlier which summed the values in a tree, the addition on the line marked -> takes two parameters, which can be evaluated in parallel. The parameters are calculated as recursive calls to the same functions which calculate the sum of the values in the two subtrees of the current node. This bifurcation leads to a O(log n) solution instead of the sequential O(n) solution.

Thus it is possible to extract parallelism from a functional program by taking advantage of its property of referential transparency. This makes parallelisation more straightforward than for the conventional sequential case, where the compiler must attempt to keep track of interdependencies. Typical of this approach is the GEC parallel functional language project undertaken as part of ESPRIT project 415 [Burns 1989] which is attempting to produce compilers for lazy functional languages such as Miranda and Orwell.

Unfortunately functional languages have found only limited application, and are only used extensively by a relatively small community, which consists mostly of academics. The only significant exception to this is LISP, the oldest functional language. However modern versions of LISP no longer conform to a pure functional model, and do not guarantees referential transparency, making it no easier to parallelise than a conventional sequential language. Thus it appears that the very things that make functional languages relatively easy to parallelise, also prevents them from being accepted as viable general purpose languages.

4.3 Explicit Parallel Languages

Explicit parallel languages allow the programmer to directly express parallelism using special language constructs. This approach has several advantages over parallelising compilers, not least of which is its relative simplicity. Parallelising compilers must analyse programs, looking for any interdependencies and are very sophisticated pieces of software. By comparison a compiler for an explicit parallel language simply has to translate the chosen parallel constructs into parallel code.

However the main advantage of explicit parallel languages is their efficiency which derives from two aspects of their design. First the parallel constructs used in such languages are tailored towards the specific machine on which they are to be used, so as to allow the compiler to map them efficiently onto the underlying architecture. More subtly because the parallel constructs are visible to programmers, they are likely to attempt to find suitable algorithms which can take advantage of those constructs. Thus since the constructs are chosen because of their compatibility with the chosen machine, an algorithm chosen because it matches those constructs will in general be a good match to the architecture itself.

The parallel constructs in common use can be subdivided into a number of categories. These are data parallel and control parallel. Control parallel constructs can be subdivided into shared memory and distributed memory, while data parallel constructs can be divided into fixed topology and variable topology. It is interesting to note that these classifications are essentially the same as the architecture classifications used in chapter two, which supports the assertion made above that explicit parallel languages are targeted at specific machines.

4.3.1 Control Parallel Languages

Control parallel systems use a number of independent but cooperating processes to perform a computation. This style of computation is associated with MIMD architectures where each process or thread is situated on a different processor. In order to express control parallelism a language requires three essential constructs. Firstly a mechanism to create threads, secondly a mechanism for the threads to communicate, and finally a mechanism for threads to synchronise.

Synchronisation refers to the ability of one thread to wait for an event caused by the execution of a different thread. In general threads must synchronise in order for them to interact, such as communicating, or coordinating access to shared data (on shared memory systems these are often the same thing).

For example in a producer consumer problem, the producer is a thread which wishes to communicate a sequence of data values to the consumer. For this to function correctly, the producer must produce data at the correct rate for the consumer to consume it. To do this the producer must wait for the consumer to consume each data item before it produces another, and conversely the consumer must wait until the producer has produced a data item before it attempts to consume it.

As well as communication, threads may need to interact when accessing shared data values. A simple example is where a shared variable is being used to store a simple numeric value. If two threads wish to increment the shared value simultaneously the access must be sequentialised to avoid an incorrect result. This is because the increment operation will generally involve the thread first reading the value in the variable, then incrementing it and then writing the new value back. If both threads read the value simultaneously, and then both increment it and write it back, the final value will only have been incremented once rather than twice. Synchronisation is required to ensure that only one thread attempts to increment the variable at a time.

Reference is often made to parallel programming models. An example of such a model is CSP or communicating sequential processes proposed by Hoare [Hoare 78]. Many languages are based on this model, such as, Hoare's own languages CSP, OCCAM [Shepard 87] and Logical Systems C [Mock 86]. All these languages contain the same set of constructs for process creation, synchronisation and communication. There are a number of other models, each of which is used in a number of languages.

However there are also many constructs which are not associated with a particular model, and these may be mixed and matched with other such primitives to provide a set of constructs for a parallel language. These constructs are often found in existing languages which have been extended to allow concurrency control. An example of this is Logical Systems C for the Transputer, which is a version of the C language which has been extended with a variety of concurrency constructs, including message passing, semaphores, guarded selections, fork/join, and shared variables.

In the context of such a muddled situation it is difficult to arrange an overview of these different systems satisfactorily. The following section presents the most common constructs broadly grouped into programming models. The relevant languages which make use of those constructs and models are also presented. The review is divided into two parts, the first of which deals with constructs associated with shared memory systems, and the second with constructs associated with distributed memory systems.

4.3.1.1 Shared Memory Constructs

Many multiprocessor machines use a shared memory architecture, and these are often programmed in existing languages modified to allow access to the multiple processors. These extensions are often fairly low level. Communication is usually via shared variables, which require no additional support, since all threads share a single address space. Threads are typically created directly using either spawn or fork primitives, and synchronisation is usually by either semaphores, critical regions or monitors.

Process creation

Spawn - creates a new thread of control, which executes a specified piece of code. Spawn often allows parameters to be passed to the newly created thread.

Fork - creates two identical threads of control which both continue execution after the fork statement, and share the same code and data. The fork operation returns a value which is different for the original thread and the newly created thread. In this way the two threads can diverge.

Parallel Statements - statically associate particular statements with different threads. The total number of threads is fixed for a particular program, although it may change at different points in the program.

Synchronisation

Semaphores - make use of an atomic shared variable which is initially set to either zero or one. When two threads wish to synchronise they perform wait and signal operations on the semaphore. The wait operation checks the value of the semaphore. If it is one or greater then the operation decrements the value and continues. If it is zero the operation waits and the thread is blocked. The signal operation increments the semaphore and allows one waiting thread to continue.

Critical Regions - are regions of code which allow only a single thread to enter at one time. Typically critical regions are used to serialise access to shared data to guarantee its integrity.

Monitors - are collections of routines which are used to access shared data. Threads can only access shared data via these routines. The routines then carry out the necessary synchronisation to ensure that the data's integrity remains intact.

4.3.1.2 Distributed Memory Constructs

Distributed memory systems consist of a number of processors each with its own local memory. They are connected via some form of interconnection network which allows the processors to communicate. Most of the programming systems for these machines use combined communication and synchronisation primitives, rather than separate ones.

Communication and Synchronisation

Remote Procedure Calls (R.P.C.'s) - are modelled on conventional procedure calls, using familiar syntax, and similar semantics. R.P.C.'s are a two directional fully synchronous construct. When a calling thread executes an R.P.C. it is blocked, and remains blocked until the remote procedure finishes execution and returns a value.

At the receiver a new thread is created to service the procedure. This potentially creates multiple threads within the procedure, which means that an additional synchronisation mechanism must be included to ensure safe access to shared data within the procedure. When the remote procedure is complete the thread terminates, and control is passed back to the calling thread which can then proceed

Message Passing - refers to a mechanism which transfers a packet of data from one thread to another. Message passing may be synchronous or asynchronous, and either fixed topology or routed.

Synchronous message passing requires the sending thread to wait until the receiving thread accepts the message, at which point both continue. Asynchronous message passing allows the sender of the message to continue as soon as the message is sent.

Asynchronous message passing is potentially more efficient than synchronous, since it avoids the sender having to needlessly wait for its message to be delivered. However to implement this, the programming language must buffer the messages which are waiting to be delivered. Given that buffers must be finite, there may come a point at which no more messages can be sent until the receiver has accepted one. This creates a dependence which is not consistent with the asynchronous model, and may lead to the program unexpectedly deadlocking.

Fixed topology systems are based on point to point message passing channels along which messages are sent. Each channel can connect two predetermined threads, which are usually fixed for the duration of the program. Routed systems allow any thread to send a message to any other thread, the destination for each message is determined dynamically at run time.

Rendezvous - is a two way communication mechanism used principally in Ada, but also in Concurrent C and Concurrent C++ [Gehani 88]. A rendezvous is initiated by a calling thread, or task as it is called in Ada, by executing a rendezvous call statement. The syntax of the rendezvous call statement is similar to a procedure call, with a name for the rendezvous, and multiple typed arguments. The name and argument list must match the corresponding ones in the called task's rendezvous accept statement. Each task has a definition part, which states the name and argument lists of all the rendezvous calls the task may accept. Within the body of the task, the code may then select which rendezvous calls to accept at any instant.

When a rendezvous call is made the calling task is blocked. When the call is accepted by the called task a block of code associated with the call is executed. This block of code may read the input parameters passed by the caller, and may write to the output parameters of the caller. In this manner a two way communication is set up. Once the rendezvous is complete both tasks may continue execution in parallel.

Linda - actually encompasses a complete parallel programming model rather than just a communication and synchronisation mechanism, but since the principal feature of Linda is its communication model, it will be dealt with here.

Linda [Ahuja 86] uses a shared tuple space, which threads read from and write to, in order to communicate. A tuple is an ordered collection of atoms, where each atom is a single value of a particular type. Threads may write any number of tuples into the tuple space without blocking. A thread reads a tuple by specifying a template, which is a tuple of the same order as the tuple to be matched.

Each atom of the match tuple may either specify a specific value, or a type. If it specifies a type then it matches any atom with that type. If it specifies a value then it will only match an atom with the same value. The template will match all tuples in the tuple space whose corresponding atoms match those in the template.

For example if the tuple space contains the tuple ("fred", 2, b), i.e. a tuple containing a string, an integer and a character, then possible templates that will match it include, ("fred",**int**, b), (**string**, 2, b), (**string**, **int**, **char**) and ("fred", 2, b).

If when a thread attempts to read a tuple, no match is present in the tuple space, the thread blocks, until a matching tuple is written to the space by another thread. Thus the tuple space may be thought of as a high level shared memory system, which enforces synchronisation on data accesses, although it has been shown that Linda can be implemented on hardware which does not itself have a shared memory architecture.

The Linda programming and communications model has been incorporated into a number of existing programming languages including C and Pascal as well as in the language Linda itself.

4.3.1.3 Shared vs Distributed Memory

The above summary of programming constructs is given roughly in increasing order of sophistication. In general the later ones may be regarded as higher level constructs, which are more complex to implement but are generally easier for the programmer to use. In particular there is an important distinction between the shared memory with synchronisation model and the private memory with communications model.

With a shared memory system even the simplest statement such as A = A + 1 can behave unexpectedly without the proper synchronisation (if two threads simultaneously read the value of A then increment it and write it back A will be incremented once not twice as one would expect). Worse still the statement will typically perform as expected most of the time, and only fail in the rare case when two threads access A simultaneously. This non deterministic behaviour makes debugging extremely difficult.

By comparison the private memory systems, where each thread has its own local data (Occam, Ada, Linda etc.) , are simple to program in, because as each thread has its own local data the synchronisation problem discussed above can never arise. Also these languages support communications systems that include the necessary synchronisation to perform in the expected fashion. With these systems although the implementer may have to deal with the complex issue of synchronisation, the applications programmer will not have to. This tends to concentrate the difficult multi-threaded code into a small number of system routines, rather than being spread out amongst the application code, which improves maintainability.

4.3.2 Data Parallel Languages

Data parallel constructs are generally based on special parallel collection data types. These data types vary from language to language, but they all allow operations to be performed in parallel on all items of data in the collection at once. As with the control parallel constructs, data parallel constructs are usually chosen to match the architectures on which they are to be run. Existing SIMD architectures can broadly be divided into two categories, those which have a fixed topology, generally a square array, and those which allow a reconfigurable topology. Similarly the data parallel languages support either fixed array data types or more general collections.

4.3.2.1 Fixed Topology Constructs

The class of machines referred to as parallel array processors, have been mentioned in chapter two. Typical of these are the AMT DAP, the UCL CLIP4 and the MPP. All these machines allow a square array of data to be manipulated in parallel using special array operations, so that for example, a single addition operation may add all the corresponding elements of two arrays together, in parallel, to produce a third array. The languages used on these architectures provide equivalent array data types and a set of array operations which act on these data types in parallel.

ICL DAP FORTRAN-PLUS - provides two special data types, the vector and the matrix [AMT 88]. Vectors are one dimensional arrays, and matrices are two dimensional. Each element of the vector or matrix is mapped to a single processor in the processor array. This is based on the assumption that the matrix is the same size or smaller than the processor array. If the matrix is larger than the processor array then multiple elements of the matrix will be mapped to a single array processor (this is only supported on later versions of the DAP FORTRAN compiler).

Once a variable has been declared to be of a matrix or vector data type, then any arithmetic operation on that variable is interpreted as an array operation and is performed in parallel over the whole matrix. In addition to basic arithmetic operations, an extensive library of predefined routines is supplied to perform common operations, principally for image processing operations, but also for more general scientific applications.

Given below is a simple FORTRAN-PLUS program segment which illustrates these points.

```
1     integer A(64,64),B(64,64),C(64,64)

              :

2     A(C.GT.thresh ) = A + B
3     C = Sobel8(A)
```

Line 1 declares three variables A,B,C as matrices of size 64 by 64 elements (the indices may be omitted in some versions of FORTRAN-PLUS). Line 2 adds the matrix B to the matrix A conditionally on matrix C, so that only the elements in matrix A which correspond to the elements in matrix C which have a value greater than the chosen threshold will be assigned a new value. Line 3 calls a built in function which performs a Sobel edge detection on matrix A. This assumes that matrix A is being used to represent an image of 64 by 64 pixels.

Image Processing C - is an extension to the C language developed as part of the CLIP program [Reynolds 81]. IPC provides facilities equivalent to those of DAP FORTRAN, by providing an image data type from within the C language. Data for the image data types are stored in the CLIP's array memory, and operations performed on these images are performed in parallel by the array.

A comprehensive set of library routines are supplied with the system which perform common image processing functions such as convolution and edge detection as well as image input and output. In addition a run time system is provided which handles allocation of memory within the CLIP's array memory, thus freeing the programmer from these tasks.

Apply - [Hamey 87] takes a slightly different approach to the other data parallel languages. In Apply programs are written from the point of view of one pixel of an image. A program consists of a single procedure which is applied to all the pixels in an image potentially in parallel.

Conceptually the procedure is executed once for each pixel in an image, and the value of the current pixel is passed as a scalar parameter to the procedure. Within the procedure the code appears like an ordinary scalar program, the Apply system being responsible for performing the routine on all of the pixels in the image. In Apply arrays represent a section of the image whose origin is the current pixel. This allows the procedure access to its immediate neighbours, and potentially any other pixel, depending on the size of the array.

As an example the routine replaces every pixel (x,y) by the average of the pixels (x,y),(x+1,y),(x,y+1),(x+1,y+1).

```
procedure average(image : in array (0..1, 0..1) of byte,
                  result : out byte )
is
begin
     result := (image(0,0) + image(0,1) +
                           image(1,0)+image(1,1))/4;
end average.
```

Notice that the result is declared as a scalar quantity, but actually refers to an entire image. This provides a simple and elegant way of dealing with arrays, and is straightforward to compile onto a variety of SIMD hardware.

4.3.2.2 Variable Topology Languages

CmLISP - [Hillis 85] is the language used by Thinking Machine's Connection Machine. This is a conventional SIMD architecture, but rather than arranging the processors in a fixed square grid, they are interconnected using a dynamic routing system, which can route messages from any processor to any other processor. Physically the communication network is a hypercube, but the inclusion of a hardware routing mechanism allows the programming model of the machine to assume that every processor is connected directly to every other.

To take advantage of the architecture CmLISP has been developed, which allows the manipulation of arbitrarily complex parallel data structures. These data structures, known as xectors, are modelled on the LISP list construct, and use a mechanism similar to the LISP apply operation to process them in parallel. This corresponds to the conventional fixed topology SIMD arrays and array operations. In addition to this xectors may also be manipulated using a construct known as ß (Beta) reduction, a mechanism which allows xectors to be transformed and combined with other xectors. This is the mechanism which gives the programmer access to the arbitrary connectivity of the Connection Machine.

In CmLISP a xector is a list of mappings: each mapping takes a single LISP atom and maps it to another. In any xector there is at most one mapping for any given atom, although many atoms may map to the same atom. Thus the mapping specified by the xector is mathematically a function. A special case of the xector is equivalent to a conventional array, where the xector maps a set of natural numbers to a set of arbitrary atoms, thus the xector can be thought of as a more general version of an array.

For example

{ sky → blue grass → green tomato → red } is a xector which maps various objects onto their colours. The objects can be thought of as the indices for the xector. The more familiar array formulation would be

{ 1 → blue 2 → green 3 → red } or just [blue green red]

which is an array of the colours with indices 1, 2 and 3. There are various operations for accessing the individual elements of the xector, such as XREF which returns the mapping for a given index, and XSET which sets the mapping for given atom.

Operations on xectors may be performed using the α notation, which creates a xector of functions. When the xector of functions is applied to one or more xectors, all the functions are performed in parallel.

For example

(α+ [2 3 4] [4 5 6]) ⇒ [6 8 10]

adds the corresponding elements in the two array xectors together in parallel. Corresponding elements must have the same index, any indices which are not found in both input xectors are ignored.

Xectors and α notation give essentially the same functionality found in languages such as DAP FORTRAN-PLUS, that is data parallel operations on arrays of data. However CmLISP includes one more feature which makes it unique, namely beta reduction.

Beta (ß) reduction gives the programmer access to the Connection Machine's routing system. In its most general form ß reduction takes a combining operation and two xectors (generally array style xectors). The reduction process involves taking the values specified in the first xector and constructing a new xector which contains those values, stored at indices specified by the second xector. If two values map to the same index then the combining operation is used to calculate the value stored at that location.

For example

```
(ß+ [2 3 4] [X Y Y]) ⇒ [X → 2  Y → 7]
```

here the ß reduction is using addition as the combining operation. It takes each value in the first array and maps it into the result xector using the corresponding index in the second. So the value 2 is stored at index X. Both the other values are to be stored at index Y, so the combining operation is used to add then together leaving the sum in the result.

CmLISP is an extremely powerful and expressive language which allows full access to the Connection Machine's architecture. Unfortunately it would be very difficult to implement on anything other than a Connection Machine, since it relies on the routing system which is almost unique to the Connection Machine amongst existing SIMD architectures.

4.4 Conclusions

Users who have a specific requirement to use existing sequential software on a parallel machine, may find parallelising compiler technology an appropriate choice. For all other users it is clear that the most effective way of programming a parallel machine is using an explicit parallel programming language. Pure functional languages, while they provide an interesting alternative, suffer from a general lack of acceptance which means it is unlikely they will ever be a serious challenge to conventional languages.

Of the explicit parallel programming approaches it is clear that there is much merit in using the highest level programming model possible. In particular the notion of separate processes with local data which communicate by message passing seems to offer a number of advantages, in terms of ease of programming and debugging, over shared memory models.

It is also very important that the programming model chosen maps well onto the architecture being used. This is essential if efficient parallel programs are to be produced because of the great differences in approach taken by different parallel architectures. Implementing a shared memory programming language on a distributed memory machine, for example, would produce a very inefficient result.

Chapter 5

A Dual Paradigm Programming Model

5.1 Introduction

All the languages and techniques mentioned in the previous chapter were oriented towards either data or control parallel architectures, reflecting the architectures on which there were intended to be used. The Warwick Pyramid Machine is a multi-paradigm architecture which combines both data and control parallelism, and therefore none of the existing languages is suitable, since none address this multi-paradigm model.

One solution to this dilemma is to make use of two languages, one which supports data parallel constructs, and one which supports control parallel constructs. This is the solution chosen by a number of existing projects, such as the IUA which uses Forth for SIMD programming and LISP for its MIMD programming, PASM where it is proposed that MIMD programming be performed in Ada or CSP while SIMD programming be performed in assembly language.

Initially the Pyramid Machine project was to adopt this solution, using Transputer C and a low level data parallel language, Cluster Programming Language, which was designed, but never implemented.

However while this approach works it is not an elegant solution to the problem, and causes a number of difficulties. A principal criticism is that such an approach does not provide a means of expressing algorithms which make use of both control and data parallelism in a single uniform way, making programs more obscure and requiring programmers to become familiar with two programming systems rather than one. Also it does not provide a solution to expressing interactions between the two parts of the machine. To be acceptable to potential users and applications programmers it was considered that a single language should be developed that could be used to program all parts of the machine.

The chosen solution is to make use of a single programming model which is expressive enough to encompass both control and data parallelism. This model is based on object oriented programming which provides methodology for structuring programs based on the data that they manipulate. This model, which has been popular for some time in sequential programming systems, can be used to express all the parallelism required in our application without almost no alteration. This has allowed an existing object oriented programming language to be used almost unchanged.

A dual paradigm programming language based on object oriented programming called Pyramid C++ has been developed which, as its name suggests, is based on the popular C++ programming language, an object oriented extension to C.

In this and the following chapter the design and implementation of this language will be described in detail. However before dealing with the specifics of Pyramid C++ itself, object oriented programming as applied to sequential programming will discussed. The extensions necessary to support both control parallel programming and data parallel programming will then be discussed, with reference to related work on parallel versions of the object oriented model.

3.2 Object Oriented Programming

Object oriented programming is a design methodology intended to address the problems associated with large software systems. These problems come about because of the explosive increase in complexity of modern software projects. While the performance of hardware systems have progressed at an ever increasing rate, software systems have struggled to keep up. The problems encountered in generating new software at a sufficient rate to keep up with changing demands, and then maintaining the ever increasing body of existing software, are of such magnitude they have given rise to predictions of a so called software crisis [Meyer 88].

5.2.1 Structured Programming

Over many years software engineers developed improved methods of software construction based on structured programming, modular design and abstract data types. These design techniques have done much to improve the situation as far as improving the reliability and maintainability of software, and has to a certain extent helped in the creation of new software, by making the software easier to understand.

Structured programming encompasses a number of techniques which are aimed at making programs easier to write and maintain. Central to these ideas is modular design. This involves dividing a program into modules such that there is as little interaction as possible between modules. To achieve this programs are divided up into modules based on their requirements to share information. This introduces the idea of coupling between different parts of a program. Coupling is a measure of the amount of common data shared by two parts of a program. If two parts of a program often require access to the same piece of data then they are said to be strongly coupled. Conversely if they seldom require access to the same data they are loosely coupled. Modular design attempts to produce strong coupling within a module, known as cohesion, and weak coupling between modules.

The principal motivation for this design approach is to allow programs to be altered without introducing unwanted side effects. Modular design aids this by isolating each module from the rest of the program, such that changes made to one module should not affect other parts of the program. This however relies on the overall structure of the program remaining the same, so that the chosen module structure does not have to be altered. It is therefore very important to ensure that the initial choice of module decomposition is likely to remain stable over the lifetime of the program.

There is strong evidence to suggest that the structure of programs designed around the data that they manipulate are more stable than those designed around the functions that they perform [Meyer 88]. Taking as an example an image processing package, or a word processor, the aspect which characterises them is the data on which they operate, i.e. images or text. The exact functions that may be provided by such packages will probably vary greatly, but the structure of the data that they manipulate will be much more consistent.

5.2.2 Abstract Data Types

A successful modular structure should be based on the data that a program manipulates, which should also provide the necessary strong internal and weak external coupling. To this end the concept of abstract data types was introduced. An abstract data type is a programmer defined data structure, together with a set of routines which manipulate it. A typical example of this might be a data base, where the structure of the data and the operations to search and manipulate it would be grouped together in a single module.

Languages such as Ada and Modula 2 have specific support for the concept of abstract data types. Both provide facilities to define modules which contain data structures and the routines that act on those structures. These modules are divided into two parts, a definition part, and an implementation part.

The definition part will typically contain the definitions for the data structures manipulated by the module, and the operations provided by the module to manipulate the data, while the implementation part contains the code that actually implements the operations. In this way it is possible to design other modules which interact with a particular module by referring only to the definition part of the module. If the system is designed and built in this way it should be possible to change any module's implementation, without affecting the functioning of other modules provided that the new implementation conforms to the existing definition.

5.3.3 Reusable Software Construction

Despite the progress made by the use of structured programming ideas, researchers at Xerox and elsewhere were concerned that these ideas needed to be extended still further to avoid the software crisis. In particular it was noticed that the same or very similar code was written over and over again for each new software project. For example routines to store and search data bases, or sort data, appear in many programs. Hardware engineers can call upon libraries of integrated circuits which perform common functions, and so seldom have to reimplement such functions from scratch. What seemed to be required was what has been referred to as a software IC [Cox 86].

The conventional approach to reusable software is the use of libraries which provide commonly used functions, which can be called from the user program. However it was found that in many cases these functions were not used, instead programmers would rewrite them from scratch. The reason for this was found to be that library routines were too specific, and lacked the generality to work in a case other than the one for which they were intended. For example a sort routine, might sort integers, but the same sort routine would be incapable of sorting employee records. The proposed solution to this problem was object oriented programming.

5.2.3.1 Message Passing

Object oriented programming extends the concept of abstract data types to include the concept of message passing. In object oriented programming modules are replaced by objects. An object contains both data and the operations that act on that data. The data within an object is private to that object, so that the only access other objects have to the data is via the operations defined by the object.

Objects communicate by message passing. A message is a request for the object to perform one of its operations. The crucial aspect of message passing is that the object will interpret the message according to the type of object it is. For example a print message will be implemented differently by an integer object than by an employee object. This behaviour is known as polymorphism or dynamic binding.

This behaviour is essential to software reusability. Earlier the example of a sort routine was given. Using normal software construction techniques the sort routine will need to be rewritten to handle a different data type. Using object oriented programming the sort routine will sort any collection of objects provided they implement messages which compare two objects. This comparison may be implemented differently for every type of object, but the appropriate implementation will be chosen dynamically at run time so that the code behaves as expected.

The first object oriented language was Simula [Dahl 66], a language intended for simulation, which used interacting modules to represent objects in the real world, hence the term object oriented. While many of the facilities we now associate with object oriented languages first appeared in Simula, it was Smalltalk [Goldberg 83] that is generally regarded as the parent of object oriented languages. Smalltalk is a pure object oriented language in the sense that all data in the language is represented as objects.

5.2.3.2 Classes

Objects in Smalltalk, and other object oriented languages, are grouped according to their class. A class is a complete specification for all objects of that class. A class definition contains the specification of the data that the objects will contain, the definitions of the operations that are available on the objects, and the actual implementations of those operations.

An example class definition for an employee class in C++ is given below.

```
class employee{           // Name of class
     date date_of_birth;  // specification of data
     string name;
public:
     get_age();           // Definition of operations
     get_name();
};

employee::get_age()       // Implementation of operation
{
     return (today - date_of_birth)
}
```

As can be seen the employee object contains two data items, a name and a date of birth. Each employee object understands two messages, one to get the name of the employee, the other to get the age (a real definition would obviously include many others). Each message is associated with a function or method, such that when the object receives a message one of its methods is called.

An object of the employee class can be created by declaring a variable of that type.

```
     employee example;
```

This creates an object example of class employee. A message can then be sent to the object

```
    example.get_age();
```

This sends the message `get_age` to the object `example`. Notice that the object does not actually store the age of the employee, only the date of birth, it calculates the age by subtracting todays date. It would be possible to change the implementation so that it really did store the age, without affecting any other part of the program, provided the object still behaved the same way when sent the same messages.

5.2.3.3 Inheritance

Most object oriented languages allow classes to be defined as extensions to existing classes. This is known as inheritance. For example a class of *circle* objects might be defined as a sub-class of a *shape* class. This implies that all *circle* objects are extensions of *shape* objects. Thus all messages understood by *shape* objects are also understood by *circle* objects, although the actual implementation of the operations associated with each message might be different. For example if the operation *display* is defined for *shape*s then it will be defined for *circles* as well, although its implementation will most probably be completely different.

In addition to those methods inherited from the *shape* class the *circle* class may also define new operations which are only valid for circles, such as setting the radius of the circle.

Inheritance has two principal uses. Firstly, if a new class is required that is similar but not identical to an existing class, it allows programmers to extend the existing class rather than having to reimplement it from scratch. Secondly it allows routines to be written which make use of common behaviour of classes which share a common super-class.

For example in the previous example, if a routine were written which processed *shapes*, it would also be able to process *circles* , and any other object which was derived from the *shape* class. Both these uses greatly enhance the reusability of software written in this way.

In Smalltalk all objects are derived, directly or indirectly, from a single class called *Object*. A section of the class hierarchy for Smalltalk is shown in figure 5.1. Notice how the classes form a tree where all children of a particular node are derived from that class. Thus all children of the class node *Magnitude* will understand the messages implemented by the *Magnitude* class. This class implements a comparison method to compare two objects of class *Magnitude*, so that a sort routine could be written. Once this was done any object of one of the classes derived from *Magnitude*, such as *Date* and *Number* could also be sorted.

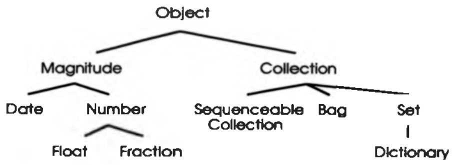


Figure 5.1: Section of Class tree for Smalltalk.

Not all object oriented languages work this way, C++ for example allows classes to be defined which are not derived from any other class, and thus does not have a predefined class structure in the way Smalltalk does, but it still allow inheritance for user defined classes.

There are now a number of object oriented languages, many of them extensions of existing languages, such as Objective C [Cox 86] and C++ [Stroustrup 86] as well as pure object oriented languages such as Smalltalk [Goldberg 83] and Eiffel [Meyer 88]. These languages are rapidly gaining in popularity, and many modern compilers for non object oriented languages, such as Think C and Turbo Pascal are providing object oriented features. This popularity is a powerful argument in favour of the use of object oriented techniques regardless of its suitability for parallel programming.

5.3 Control Parallel Object Oriented Programming

Conventional sequential object oriented languages have found much favour recently for building large systems due to their excellent modularity and resulting understandability, maintainability and reusability. One of the principal aims of the object oriented approach is to divide up a problem into small independent units, which contain both data and the operations which can be performed on that data.

To take advantage of a distributed memory MIMD processor architecture, a problem must be divided up so that each part can be executed on a different processor. Since in general most MIMD processor machines have a much higher bandwidth to local data than non-local data, it is important to keep as much data as possible local to each processor. This implies that each part of the program should be as independent as possible, and should ideally require only limited communication with other parts of the program.

As can be seen the two problems share a common goal, namely dividing a problem into independent units. Thus it may be appropriate to consider object oriented programming as a candidate for use as a control parallel programming model. It is particularly encouraging that the method has gained popularity, not because of its applicability to parallel processing, but because of its ability to construct large reliable systems. It is often suggested that parallel programming is difficult, principally because dividing programs into separate parts is considered to be awkward, yet it appears that programmers have evolved techniques for doing this quite independently of the needs of parallel programming.

In an object oriented language programs are divided up into collections of independent interacting objects. In a control parallel language programs are divided into multiple independent processes, each of which executes on a separate processor. An attractive way of merging these two concepts is to assign a separate process to each object, and then assign each object to a separate processor.

5.3.1 Actors

In a conventional sequential object oriented language a single thread of control passes between communicating objects. When one object 'sends a message' to another object, control is passed to the target object. When the operation associated with the message has completed, control is returned to the sender. In the proposed parallel object oriented model a separate thread of control is associated with each object. When an object sends a message to another object, the receiving object accepts the message and then independently executes the operation associated with that message. This model is known as the autonomous object or *actor* model.

It is interesting to note that sequential object oriented languages such as Smalltalk, use the term message passing to refer to the inter object communications, even though in reality it is implemented by what is essentially a procedure call mechanism. This terminology encourages the view that the objects are autonomous and independent even though in reality they are entirely passive entities. This gives further backing to the notion that programmers actually prefer to think of objects as independent even when not dealing with parallel systems.

A number of languages have been proposed which adopt the autonomous object or *actor* based approach. Amongst the languages are POOL [America 86] a conventional imperative language developed at Phillips for use with their DOOM (Distributed Object Oriented Machine) architecture, Concurrent Smalltalk [Yokote 86a] a concurrent extension to the Smalltalk 80, Act3 [Agha 86] a pure functional object oriented language, ACT++ [Kafura 90] an extension to C++ to support actors, SINA [Berge 89] an experimental imperative language and ABCL/1 a LISP like object oriented language.

The term *actor* was originally coined by Hewitt [Hewitt 77] to refer to a very specific functional model as used in Act3. However the term has since come to encompass a more general autonomous object system as used in most of the languages listed above. In this thesis the term actor will be used in its more general sense.

In the actor model, as in the conventional object oriented model, each actor consists of some local data and a number of methods which operate on the data. In addition to this there is a routine called the object controller which reads incoming messages from a system maintained queue, stores incoming messages until they can be processed, and determines which method should be called. It also controls which messages can be accepted by the actor at any given time. Associated with each actor, is a process or thread which executes the object controller routine and the methods. This is shown diagrammatically in figure 5.2.

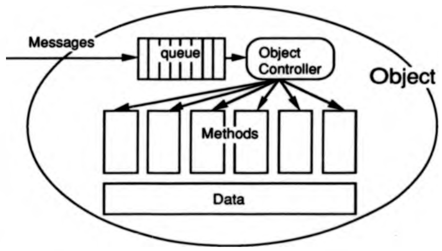


Figure 5.2: An autonomous Object or Actor

In this model each actor processes one message at a time. When there are no messages to process the actor lies idle waiting for one to arrive, and if a message arrives while another is being processed it is simply queued awaiting the currently executing method to complete.

This approach has the advantage that since only a single thread of control is active within the object at any given time, and because the object oriented model guarantees that only the object's methods can directly access its local data, there is no requirement for synchronisation to ensure correct access to the data. This makes the approach simple to understand and thus languages which use it are relatively easy to program in.

This is similar to the monitor approach to synchronisation, except that in this case the organisation of the data into private blocks which are only accessible by a small set of routines derives from the object oriented approach, rather than being a separate concept.

5.3.1.1 Controlling Message Acceptance

The job of the object controller in the actor model is to control the acceptance of messages by the object. The default behaviour is simply to accept the next message at the front of the queue, but it is also possible to select certain messages to be accepted and others not to be. Messages which are not accepted remain on the queue until the controller is told to accept them. These waiting messages do not block the message queue, so that other messages which arrive can still be received by the actor.

These ideas are illustrated in the classic bounded buffer problem. Here an object receives requests to read from or write to a finite sized buffer. The implementation is quite straightforward except for the two cases of a full and empty buffer. If a read request is received when the buffer is empty (or a write when it is full), the object cannot process the message. It also cannot simply wait for the buffer to become non-empty, because while it is active no other messages can be received by the actor.

One solution is for the object to abort and return an error, but then the sender will presumably simply have to try again, creating a polling situation, which is generally undesirable since it consumes unnecessary processor time. What is required is a mechanism for suspending the process which performed the read or write operation and then restarting it at a later stage when it can be performed.

This is achieved by use of the conditional acceptance in the following way.

```
      object buffer
      data
                buffer
      methods
                read() :returns integer;
                write(integer a);
      end;

      initialise()
      {
a)              controller.accept(read);
b)              controller.block(write);
      }


      read()
      {
c)              if (this is last entry in buffer ) {
d)                        controller.block (read);
                }
e)              controller.accept(write);
                return(buffer entry);
      }


      write(a)
      {
```

```
f)              put a in buffer;
                if (buffer full){
g)                      controller.block(write);
                }
h)              controller.accept(read);
        }
```

As can be seen the *read* methods check to see if the buffer is empty (c) and blocks the acceptance of further *read* messages (d) (by sending a message to the object controller). It then accepts *write* messages (e), on the assumption that the buffer cannot now be full since at least one entry has been read out (of course the controller may already be accepting write messages). The *write* method performs the reverse operation. It puts the value passed to it in the buffer (f), and then checks to see if the buffer is now full. If so it disables the acceptance of further write operations (g). It then enables read operations to account for situations where the reads were disabled due to the buffer being empty. The *initialise* method causes the object to accept *reads* (a) but not *writes* (b) , since the buffer is initially empty.

The above example is similar to the approach used in SINA were control of the acceptable messages is distributed amongst the methods as required. In POOL however a special routine known as the object body is used to group this control into one place. Act3 incorporates the acceptance handling in its replacement behaviour definitions, which has a similar effect of centralising the acceptance control. It is not clear that this approach provides any advantage over the approach used by SINA, but it does cause a significant problem when inheritance is introduced, since it becomes unclear which acceptance control routine or definition should be used in the derived class [Kafura 90]. Neither POOL or Act3 provide inheritance, which reduces their capabilities from an object oriented point of view.

5.3.2 Introducing Multiple Threads

With the model described so far each actor has its own thread of control that remains idle until it receives a message, at which point it becomes active while it processes the message, and then returns to the idle state. While the receiver is active processing the message the sender of the message is blocked waiting for the receiver to return a value.

If the program initially has one active thread then, given these semantics, only one thread will ever be active at any one time in the program. This is effectively the same as having only one thread, as in a conventional sequential program. Clearly if parallelism is to be achieved more than one thread must be active at one time. There are essentially two ways to achieve this

- allow explicit creation of multiple or additional active threads
- use modified message passing semantics to create parallel threads

Taking the first approach there are a number of ways in which new threads could be created. Probably the most straightforward is to assign a startup routine to each actor, so that when a new actor is created this routine is executed as an independent thread. This is the approach taken by POOL.

An alternative is to allow an actor to perform an operation similar to a fork, whereby the object creates two copies of itself each with its own thread which can then execute in parallel. The two copies do not share their data, so there is still only one thread of control per actor and no synchronisation is required. This approach is taken by SINA and Act3, and is called detach in SINA, and a replacement in Act3 and ACT++.

There are slight differences in the two implementations of this technique. In SINA the detach operation produces an exact copy, whereas in Act3 the replacement actor may be one of a set of other actors, (although in practice it is often the same one). Also whereas in SINA the original thread processes the next message to arrive, while the child continues to process the current message, in Act3 the child processes the next message while the original thread continues to process the current message.

The principal alternative to explicit thread creation, namely altering the message passing semantics, can be achieved in two different ways

- Allow the sender to continue without waiting for a reply
- Allow the receiver to continue after having sent a reply

The first option is called asynchronous message passing. Both POOL, Concurrent Smalltalk and Concurrent C++ provide this type of message passing by use of explicit syntactic constructs, while all message passing in the Actor based systems Act3 and ACT++ is asynchronous.

Asynchronous message passing works well where there is no value returned by the receiver. However if a value is required the sender would still have to wait for the invoked method to complete.To overcome this limitation the concept of a deferred result is introduced.

A process which wishes to send an asynchronous message but requires a return value, creates a return mailbox, and specifies it as the return address for the message. Once the message is sent, the sender may proceed in parallel with the receiver. When the receiver is ready to produce a value it writes it into the return mailbox. Then, at some later stage when the sender has finished its other processing and requires the value it can read it from the mailbox. If the sender reads the value before it has been written by the receiver, the sender is blocked until a value is written. Both Concurrent Smalltalk and ACT++ provide a deferred result mechanism using a return mailbox construct known as a CBox.

The second of these two methods, whereby the receiving object having returned a value may continue execution is supported by Concurrent Smalltalk and ACT++. Note however that it may not process any further messages until it has completed executing.

5.3.3 Discussion

The actor or autonomous object model has a number of appealing features. The object oriented system provides a guarantee of encapsulation so that code can only access data local to its object, and can only communicate with other objects using explicit message passing. This provides a convenient and secure unit for distribution over a collection of distributed memory processors, with the assurance that the non local memory accesses will never be required.

Also, because of the restriction to processing one message at a time combined with the guarantee of private data, all code can be guaranteed to be single threaded, and requires no additional synchronisation primitives. This provides a greatly simplified programming model, which produces programs which are easier to understand and hence likely to be less prone to programming errors.

Finally because it is based on a conventional object oriented programming model it should mean that programmers find it straightforward to learn. It is particularly interesting to notice how little change is required to allow object oriented programming techniques to be applied to parallel programming. It would appear that the two systems are actually very closely related, and so form an ideal partnership.

For completeness it should be noted that a number of parallel object oriented languages have been proposed that do not use the active object model. These languages provide separate mechanisms for processes, rather than combining them with the idea of objects. Because these languages do not use the single thread per object model they require an additional synchronisation mechanism. Examples of such languages are Concurrent C++ [Gehani 88] which uses Ada style rendezvous communications, Smalltalk 80 [Goldberg 83] which uses fork and semaphores and PRESTO [Bershad 88] and Emerald [Jul 88] which use spawn and monitors.

While these languages may have some merit they lack the simplicity and elegance of the active object approach, and fail to capitalise on the natural symbiosis that is achieved by the actor model. They also require the programmer to become familiar with low level synchronisation primitives, which are inherently more difficult to use than the implicit sequentialisation provided by the actors model.

5.4 Data Parallel Object Oriented Programming

In most existing data parallel languages such as DAP FORTRAN-PLUS and CmLISP special parallel data types are provided which allow access to the underlying architecture of the machines. These data types contain collections of data items typically arranged in a square array, or in some cases a more general connected graph. Each data item in the collection is mapped onto one processor of the processor array, such that when an operation is performed on the collection as a whole, it can be performed in parallel on all the elements.

This model maps very well onto the majority of SIMD architectures and is simple and straightforward to understand and use. It also provides the programmer with constructs which accurately reflect the capabilities of the machine, encouraging the development of algorithms appropriate to the architecture, a point which is particularly important for most SIMD machines.

5.4.1 Internally Parallel Objects

Objects in an object oriented languages behave in exactly the same way as these collection data types. Each object will in general contain a number of data items, and when a message is sent to an object it can operate on all those data items in a single operation. In all existing object oriented languages these operations are performed sequentially, but it is possible to envisage a language where they would be performed in parallel. The object oriented model guarantees that the implementation of the operations provided by objects are hidden from other objects, so the complexity of a parallel implementation could be completely hidden from the application programmer.

The object oriented approach is not limited to providing only parallel array objects, but could also provide a number of different parallel classes to coexist within the same language. For example it would be possible to provide both a set of classes equivalent to the DAP style square array classes and another similar to the Connection Machine's xectors within the same language.

Existing data parallel languages based on conventional languages must include extensions to these languages to allow primitive operations on composite data types. In a conventional version of FORTRAN or C the operation of adding two arrays together is meaningless and would produce an error from the compiler. In such languages operations such as addition and subtraction are built into the languages and cannot be redefined, or applied to data types other than those intended by the language definition.

By comparison addition in an object oriented language is interpreted as sending a message to an object which requests it to perform the addition operation. The exact implementation of the addition operation is determined by the object to which the addition message is sent. Thus alternative object classes such as parallel arrays or even parallel graphs can be provided, and assuming they implement a method corresponding to an addition message, they can be added in the same way as any other object.

It is possible to provide functions which perform low level arithmetic operations on parallel data types in languages such as C or Fortran, but these cannot be invoked using the conventional arithmetic syntax but must instead use an alternative, more verbose and usually more obscure syntax which can make complex expressions rather impenetrable.

For example

```
R = ((A+B)*(C/D)+(E-F)*G)
```

might become something like

```
p_assign(&R,p_add(p_mult(p_add(A,B),p_div(C,D)),
p_mult(p_sub(E,F),G)))
```

Such a system has the additional disadvantage that it is inconsistent when compared to non parallel expressions, so the fact that a parallel implementation is in use is constantly visible to the programmer. In the object oriented case this fact is essentially invisible to the programmer, who is free to program in a familiar style. This is particularly important for end user programmers who it is hoped should not have to be aware of the low level details of the implementation.

5.4.1.1 Use of Existing Object Oriented Languages

It is important to stress that unlike languages such as DAP-FORTRAN it is not necessary to extend the syntax or indeed the basic semantics of an object oriented language to support these operations it is simply necessary to provide the appropriate implementations for the parallel classes.

One possible implementation approach for such classes is for the compiler to support a set of built in primitive parallel classes, which map directly onto the underlying machine. From these other, more complex, classes could then be built up. This however requires that a modified compiler be produced for the chosen language.

An alternative approach is for the compiler to provide direct access to the machine instructions of the chosen architecture, a facility already present in a number of languages. The primitive data parallel classes can then be implemented using these low level operations to access the machine's parallel instructions. This approach has the advantage that in many cases no alteration is required to the host language. Also it should be stressed that the object oriented model ensures that the implementation of the parallel classes is hidden from the programmer, so both implementations will look identical to the programmer.

Because the parallel classes are definable from within the existing framework of the object oriented model, it is straightforward to provide alternative implementations for them depending on the architecture on which the system is implemented. This opens up the possibility of providing a portable data parallel language, where the data parallel constructs provided by the language could be mapped at run time to those of the machine on which the program is run.

Take for example a system which supported both fixed topology arrays, and variable topology xectors. On a DAP or CLIP the array classes could be implemented directly by the architectural features of the machine, while the xector classes could either be implemented sequentially or in parallel by providing a mechanism for simulating the arbitrary topologies. On a Connection Machine all the classes could be implemented directly in parallel by the machine.

Another possibility is to provide sequential implementations for the parallel classes, which allows software written for a parallel machine to be executed on a sequential one, or perhaps more usefully it allows software to be developed on a sequential machine for subsequent execution on a parallel one.

5.5 A Unified Control and Data Parallel Programming Model

In the previous sections some techniques for using object oriented programming to express control parallelism were described. Also it was shown how the object oriented model could be extended to express data parallelism. These techniques taken in isolation provide a viable approach to programming an architecture such as the Warwick Pyramid Machine which consists of both MIMD and SIMD parts.

However the real power of the object oriented approach is realised when these techniques are taken together. This provides a framework for a powerful unified approach to parallel programming, which allows problems to be expressed in their most natural form. This power derives from the use of the same concept to express both control and data parallelism, namely objects.

Objects are used both to model independent processors and to model parallel arrays. Due to the unified representation, parallel arrays are actually arrays of objects. Those objects can themselves be autonomous processes, or indeed other parallel arrays. Any combination is possible because to the programming system all these different concepts are simply objects.

This flexibility allows the model to capture not only both MIMD and SIMD aspects of the WPM but equally importantly the Multi-SIMD capability provided by the architecture.

The WPM consists of an array of clusters, where each cluster is an autonomous SIMD array tightly coupled to one processor of an MIMD array. These clusters can function either independently in a full Multi-SIMD arrangement to or can synchronise with neighbouring clusters to provide larger SIMD arrays, ultimately achieving full SIMD operation when all clusters are synchronised.

5.5.1 Autonomous Parallel Arrays

In the object oriented model Multi-SIMD behaviour may be captured using autonomous parallel array objects. These are objects which combine both of the attributes described in the previous sections, namely an independent thread of control and operations which operate in parallel over their data. In essence these objects are abstract representations of the hardware clusters, and their operations correspond to those which are implemented directly by the clusters.

These autonomous parallel array objects may then be linked together by making them the elements of a global parallel array object. Depending on the application it may be desirable to use a single global array object, which provides full SIMD behaviour, or use multiple autonomous arrays to allow partial Multi-SIMD and partial SIMD behaviour. The arrangement used is completely under the control of the programmer, and may be altered throughout the execution of the program to match the changing needs of the algorithms.

This model is not restricted to the WPM however, it should be equally applicable to other multi-paradigm architectures such as the IUA and Disputer. Perhaps more interestingly it should also be applicable to conventional single paradigm SIMD and MIMD machines, which although not capable of implementing all its facilities equally efficiently, can generally provide all the necessary facilities in some way. This provides the possibility of writing portable parallel programs, which should prove to be a very useful facility.

5.6 Conclusions

The dual paradigm programming model described above provides the first single solution to the problem of programming a dual paradigm parallel machine. The few other attempts to provide a solution to this problem involve the use of two separate languages, one which supports data parallel constructs, the other which supports control parallel constructs. This is not a satisfactory solution since it does not allow algorithms which use both control and data parallelism to be expressed within a single framework.

The use of object oriented techniques has allowed a single coherent dual paradigm model to be developed which, as will be shown in the next chapter, can be mapped onto existing object oriented languages. This approach is the first to provide a mechanism for expressing algorithms which combine control and data parallelism. By doing so it also allows the programmer to access to the Multi-SIMD capabilities of the WPM, which provides an efficient mechanism for the solution of many typical image analysis and image generation problems.

The object oriented approach is particular advantageous because it has already been accepted by the software engineering community as a powerful mechanism for allowing highly complex systems to be designed and maintained. This has the dual advantage that many software engineers are already familiar with the concepts involved, and also that it suggests that it is an appropriate choice for implementing the complex parallel systems found in applications such as real time image analysis.

Chapter 6

Pyramid C++

6.1 Introduction

In the previous chapter a model for dual paradigm parallel programming based on object oriented techniques was described. A programming language, called Pyramid C++ has been implemented which is based on these ideas. This language is targeted specifically at the Warwick Pyramid Machine, directly supporting all the concepts implemented by this architecture, however it is hoped that many of the concepts introduced in this language are applicable both to other object oriented languages and also to other architectures.

Pyramid C++ as its name suggests is based on C++, an object oriented extension to the C language. One of the principal design objectives has been to ensure that the new language is as similar as possible to the conventional language on which it is based. The principal motivation for this design decision was to allow programmers already familiar with C++ to move to the new language with as little effort as possible. It was noted that the majority of commercial parallel systems are programmed in extensions to existing sequential languages rather than totally new parallel languages. It was therefore felt that maintaining as much compatibility as possible with C++ would improve the likelihood of the language being adopted.

There was another important aspect to this approach however, namely to demonstrate that the object oriented paradigm as it stands is sufficient to support all the parallel programming constructs required to express both control and data parallelism, with little extension. The fact that this has been achieved with a high degree of success suggests that this original assertion has a high degree of validity .

In this chapter the Pyramid C++ language will be described in some detail, with emphasis on the extensions that have been made, principally to the semantics of the language to support dual paradigm parallelism. Following this the implementation of both the language translator and the run time environment, which plays a particularly important role in this system will be described. First however it seems appropriate to briefly describe the C++ language on which Pyramid C++ is based.

6.2 Sequential C++

C++ is an object oriented language designed by Bjarne Stroustrup [Stroustrup 86] at Bell Labs. It is based on the language C which was also designed at Bell Labs by Dennis Ritchie. C is a block structured procedural language, which is aimed at low level systems programming applications. It was originally written as part of the UNIX project, and UNIX itself, along with most UNIX applications programs, is written almost entirely in C.

C combines many of the features associated with structured programming languages such as Pascal including user defined data types, recursion and strong typing, with lower level facilities such as bit manipulation and access to machine addresses, a combination which makes it ideally suitable to complex systems programming projects such as operating systems.

C++ introduces a number of extensions to C which allow it to support object oriented programming. These include all the facilities normally associated with an object oriented language such as classes, inheritance, polymorphism and operator overloading. However unlike pure object oriented languages such as Smalltalk where all data is represented as objects, C++ is very much a conventional language with object oriented extensions.

The rationale behind extending C rather than developing a new totally object oriented language is that it allows the large number of programmers already familiar with C to learn the new language relatively easily. This is a very similar rationale to that behind extending C++ rather that developing a new parallel language. It is interesting to note that C++ has rapidly become the dominant object oriented language, which seems to suggest that these arguments have some merit.

Classes in C++ are templates, defining the pattern for all subsequent instantiations of objects of that class. This makes them essentially the same as user defined types and indeed the same syntax is used to declare them. This contrasts with Smalltalk where classes are regarded as objects which create instances of themselves.

A class definition in C++ contains a list of variables which are to be used to store the object's data and a list of functions which are the operations which will act on the data. Only the functions defined in the class (called member functions) may access the data of the object, so the data is only accessible to other objects via the operations defined for the object. This ensures the encapsulation which is so important for the use of object oriented languages for programming distributed arrays.

A class may be derived from another class using the inheritance mechanism. This implies that the new class, called the subclass, contains all the data and all the operations of the existing class, called the super class, plus any new data and new operations defined by the subclass. Data in the superclass may either be made accessible to the subclass or protected from it at the programmers discretion.

The subclass may redefine any of the functions defined in the superclass, provided those functions are declared as virtual functions in the superclass. Virtual functions provide the mechanism by which C++ implements polymorphism allowing a number of alternative function implementations to be associated with a single virtual function. Where a class has a number of subclasses, each of which defines different versions of the same virtual function, the compiler will ensure that, at run time, when the virtual function is called on a member of one the subclasses, the function implementation defined by that subclass will be called.

For non virtual functions the compiler can determine the address of the member function to call at compile time (early binding). For virtual functions however the exact address cannot be determined until run time, since the class of the object that the function is to be applied to cannot be known until run time. This is known as late binding. Virtual functions are implemented using a per class function table which contains the addresses of the virtual functions defined for that class. Each object contains a pointer to the virtual function table for its class. Thus when a virtual function is called, the address of the function to be called can be determined by indexing into the virtual function table. This achieves dynamic binding at a cost of only a single level of indirection.

It should be noted that while C++ supports object oriented programming, it also still supports all the existing C constructs, and while this upward compatibility has undoubtedly been instrumental in its popularity, it does have a negative aspect. Programs written in C++ vary considerably in style, from almost procedural to almost completely object oriented. This can make C++ programs rather hard to read and understand. All the parallel extensions made in Pyramid C++ are based on the object oriented model, and it is hoped this will encourage a more consistent object oriented approach.

6.3 Language Description

Pyramid C++ is an exact superset of C++, any C++ program is a valid Pyramid C++ program. An important goal of the design of Pyramid C++ has been to minimise the changes to the syntax of the existing C++ language. The intention was that all the support for parallelism would be provided by the existing object oriented facilities of the language, so no additional constructs would be necessary. In practice minor extensions have been necessary, but these mostly concern the semantics of the object oriented constructs rather than the syntax.

6.3.1 Support for Control Parallelism

The support for control parallelism is based on the active object or actor model discussed in the previous chapter. In keeping with the aim of maintaining compatibility with sequential C++ this support has been closely integrated into the existing C++ object oriented mechanism. This contrasts with the approach taken by both ACT++ and Concurrent C++ where the parallel constructs are separate and incompatible with the existing C++ mechanisms.

6.3.1.1 Actors

An actor is an autonomous object with its own thread of control. Actors communicate by sending messages which instruct the recipient to perform one of its methods. An actor becomes active whenever a message is received, and performs the method specified by the message. When there is no message to process, the actor becomes idle and waits for another to arrive. If a message arrives while another is being processed it is queued until the current method has completed. Queued messages are serviced in a first in first out order.

The strictly one at a time message processing permits only a single thread within each actor at any one time, and thus provides automatic sequentialisation of accesses to the actor's data. This, combined with a guarantee of data privacy provided by the object oriented model, means that no synchronisation mechanisms are required to ensure correct access to local data, which greatly simplifies the task of the programmer.

Actors are defined using the same syntax as other objects in C++, except that before the keyword `class` in the class definition, the keyword `actor` is used. An example of an actor definition is given below.

```
actor class example{
     int  data_item1;      // definitions of data items
     char data_item2;
public:
     int  op1(int arg);    // definitions of operations
     void operation2();    // known as methods or member
}                          // functions
```

Like a conventional object an actor has a number of data items, which are private to it, and a number of public member functions or methods, which perform operations on that data. This guaranteed privacy of data is the key to allowing the system to automatically distribute actors over an array of processors.

An actor may only directly access its own data, so if it wishes to access the data of another actor it must do so via the operations defined on that actor. Because of this the system has only to ensure that the local data defined for an actor resides on the same processor to guarantee that no non local memory accesses will occur. This is crucial for distributed memory systems which do not support non local memory accesses.

In fact in conventional C++, objects of the same class are allowed to access each others private data. This allows certain operations to be performed more efficiently, but is not consistent with the generally accepted object oriented model, and is not allowed in languages such as Smalltalk. This facility has been necessarily removed for actors, which are never allowed to access other actors private data even if they are of the same class to ensure that non local memory accesses do not occur.

Actor Creation

Actors may be created in two ways, namely by declaring a variable of an actor class, or by explicitly calling the `new` operator on an actor class. In either case the run time system automatically places the actor on a processor which it determines, and returns the address of the actor. This address, which includes the number of the processor on which the actor resides, is either returned directly, in the explicit case, or bound by the system to the declared variable as appropriate. This address is used by the run time system to route any subsequent messages to the appropriate processor.

Thus actors may communicate with each other using these addresses, in the form of references to other actors, without regard to their physical position. In this way programs are always independent of any particular topology or any particular size of array, which makes them highly portable. This contrasts with languages such as Occam where the topology is fixed, and programs must be modified if the topology or number of processors is altered.

The code to declare an actor of the `example` class given below.

```
     example eg_actor;     //declare an actor of class example
```

Alternatively an actor can be created explicitly

```
     example *eg_actor = new example;
```

In this case the operator `new` is applied to the actor class which returns the address of the newly created actor. This is then assigned to the `eg_actor` variable which is declared as a pointer to an `example` actor, signified by the * before the variable name.

All classes in Pyramid C++, including actor classes, may specify a special method responsible for initialisation, called a *constructor*. This method is called automatically by the runtime system whenever an object of the class is created, either explicitly or by declaration. In the case of an actor the constructor will be executed by the newly created actor's thread, and can therefore proceed independently of the thread in which the actor was created. This provides the first mechanism by which multiple threads may be created in Pyramid C++.

Message Passing

All objects in C++ communicate by calling one of the set of defined member functions of another object. This process is often referred to as message passing, even though for normal passive objects it is actually implemented as a simple function call. Thus when a 'message' is sent to a passive object control passes to the appropriate method, the operation is performed, and control passes back to the caller, possibly returning a result.

For actors however calling a member function is implemented using message passing which is capable of passing messages between any two objects, regardless of whether they are on the same processor or different processors.

The message passing system used is based on semi-synchronous message passing. When a message is sent to an actor the sender is suspended while the message is delivered. As soon as the message arrives at the receiver, the sender may continue execution, assuming no return result is required. The object in receipt of the message then proceeds independently and in parallel with the sender. This semi-synchronous approach is taken to resolve the conflict between synchronous and asynchronous message passing.

With pure synchronous message passing the sender must wait until the method associated with the message has been completed, and thus no use is made of the potential parallelism that could result if no return value is required. The asynchronous alternative, where the sender is allowed to proceed as soon as the message is sent, gets round this problem, but introduces an implementation problem.

For pure asynchronous message passing to be implemented, messages must be able to be sent even if the intended recipient is busy. This requires that messages in transit must be buffered. Since in a real implementation buffers will be finite, it is possible that these buffers may fill up. If this happens, objects attempting to send messages will be blocked until a sufficient number of messages are accepted by other objects to allow more messages to be buffered. This introduces an unexpected dependency between the actor attempting to send a message and the actors that the messages currently filling the buffers are intended for.

These dependencies will not have been apparent to the programmer who is presented with a pure asynchronous programming model. Also these dependencies will generally be non deterministic in nature, depending on the momentary state of the message passing system. Such parasitic dependencies can easily lead to deadlock and other problems, and present a real implementation challenge for asynchronous systems.

The semi-synchronous model guarantees a finite use of buffers, since only one outgoing message can be pending from each actor, but still allows the operations to be processed independently thus exploiting any potential parallelism. This provides the second mechanism by which independent threads may be created.

The syntax for sending a message to an actor is identical to the conventional C++ syntax used to call a member function in a passive object.

```
    eg_actor.op1(arg);   // send a message to the actor
                         // to invoke operation 1
```

The code to send a message to the actor `example` defined previously is shown above.

Returning Results

In its most basic form when the semi-synchronous message passing scheme is applied to a method which returns a result, the sender will be suspended until the return value is received. This system therefore only leads to the creation of parallel threads when there is no value returned, so that the sender is not required to wait until the message is processed. To overcome this limitation two mechanisms are provided to decouple the sender and receiver and so introduce the possibility of parallelism. These mechanisms are based on a deferred value return mechanism and non terminating value returning.

If the called method returns a result, which the sender does not require immediately, it is possible for the sender to defer receipt of the return value until a later stage using the deferral mechanism. This is similar to the CBox mechanism used in Concurrent Smalltalk. When a message is sent to a deferred method, the sender immediately receives a `Pledge` as a return value. At some later stage the sender can evaluate the `Pledge`, at which point the value from the called method will be returned. If the called method has not yet returned a value the sender will be suspended until one is received.

This example shows a deferred result being used

```
int result;                           // Final result
Pledge deferred_result;               // Promise of result
                                      // to come

deferred_result=actor.method();       // get promise
/* other code */                      // do something else
result << deferred_result;            // get result
```

In addition to this when a called method returns a result, it does not necessarily have to terminate, but instead may continue execution independently of the sender and terminate at some later stage. This is achieved by the use of the `reply` construct which sends a reply to a specified actor, generally the sender, which does not cause the method to terminate in the way that the conventional `return` construct does.

For example

```
actor::method()
{
     /* do some processing */
     sender.reply(result);// Send return value
     /* do some more processing */
}
```

The combination of non terminating reply and deferred methods, allows complete control over inter actor communication. They allow sending messages to be completely decoupled from receiving replies. Thus the programmer is free to choose whether one or two way communication is required. If two way communication is needed, the programmer can decide both at which point in the called method the reply should be sent, and also at which point in the caller the result should be received.

Passing Parameters in Messages

Certain necessary restrictions are associated with message passing, which involve the types of parameters that can be passed to an actor method. If the parameters are primitive data types, user defined data types or passive objects, they must be passed by value, or in other words pointers may not be passed. This restriction is necessary since the destination actor may reside on a different processor from the sender. If this is the case any pointers passed as parameters will be meaningless on the other processor, which does not have access to the memory referred to by the pointer.

The exception to this rule is for actors, which must always be passed by reference if used as a parameter in a message. This restriction is introduced because passing actors by value is considered too complex and expensive. To implement value actor passing, would require copying all data associated with the actor to the destination processor, and then creating a new thread for that actor on the new processor. If this facility were provided it would restrict actors from being able to use pointers internally, since these pointers would not be able to be passed between processors. Most object oriented languages pass all objects by reference, so this behaviour is quite common in this environment.

To allow actors to be passed by reference, pointers to actors are implemented differently from ordinary pointers, in that they contain a unique network wide address, which points to a specific actor on a specific processor. In this way these pointers are still valid when passed from one processor to another, and thus may be used as parameters in messages.

Controlling Message Acceptance

Associated with each actor is an object controller. This controller is responsible for dispatching the operations requested by incoming messages. An actor can alter the behaviour of the controller by use of special calls. These calls allow the actor to block some messages while accepting others. There are two calls available to the actor, one blocks an operation the other accepts an operation.

To block the acceptance of a particular message the method would call the block method in the controller-

```
    controller.block(method_name);
```

Once blocked the message can be re-enabled using the accept call

```
    controller.accept(method_name);
```

When an operation is blocked, any incoming messages that request that operation are queued until it is unblocked. In the meantime messages requesting other operations are still accepted. The use of this facility is illustrated in the example given in the previous chapter to implement a bounded buffer.

6.3.1.2 Coding Example

To illustrate the above points an example program is given below. This program implements a sieve of Erastothenes, an algorithm used to generate prime numbers. The program creates a linked list of processes, each of which stores a prime number. Candidate prime numbers are passed along the list, and at each stage are tested for divisibility with the prime number stored at that stage. If the number reaches the final stage it must be prime, and so it is outputted and stored ready to test subsequent numbers.

```
actor class sieve{              //process class definition
     int prime;                      //variable to store prime
                                //number
     sieve *next;               //pointer to next stage in
                                //list
public:
     sieve();                   //constructor, for
                                //initialisation
     test(int candidate);       //operation to test candidate
                                //prime
};

sieve::sieve()                  //constructor
{
     prime = 0;
}

sieve::test(int c)              //implementation of test
operation
{
     if(prime == 0){            //this is the last stage so p
                                //must be prime
          print("%d",p);        //print it
          prime = p;      //store it
          next = new sieve;     //create the next stage
     }

     else if((p % prime)!=0){  //test p for divisibility
```

```
                                 //against the stored prime
->         next->test(p) ;       //if not divisible pass onto
                                 //next stage
     }
}

main()                           //main routine, execution
                                 //starts here
{
     int i;
     sieve *head;

     head = new sieve;           //create first stage in list
     for(i=0; i<100; i++){       //iterate over 100 candidate
                                 //primes
->         head->test(i);        //pass each one to head of
                                 //list
     }
}
```

An important point to notice about the above example is that the program appears very much like a sequential C++ program. Indeed simply by removing the keyword actor on the first line it would compile and run correctly using a conventional C++ compiler, although obviously it would make no use of parallelism. This emphasises the small degree of the changes to the syntax of the language necessary to support control parallelism.

The operation of the program is presented diagrammatically below.

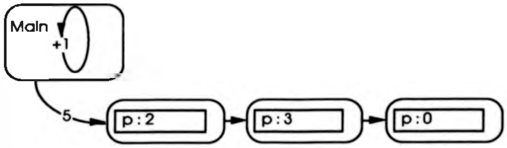


Figure 6.1: Sieve in operation

The main routine continually increments a counter which holds the candidate primes and passes each on to the first sieve in the list. In this example the next candidate prime is five. The candidate is sent to each sieve stage in turn, and at each stage is tested for divisibility against the prime number stored at that stage. If the number is divisible by the prime it is not itself prime and is rejected. If it is not divisible, then it is passed to the next stage.

In this example five is not divisible by two or three, and so continues to the final stage. The final stage is identified by storing a zero. If a number reaches the final stage it must be prime, so it is printed, and stored to be used to test subsequent candidates. At this point the last stage creates a new stage which is initialised to zero.

As mentioned above this algorithm will work equally well as a sequential or a parallel program. In the parallel version however all the sieve stages can operate simultaneously. Thus instead of a single potential prime number passing along the sieve, a continual stream of potential primes moves along. In this way many potential primes can be tested in parallel, and therefore the throughput of the program will be increased several fold.

The parallelism in the above example is introduced because of the modified message passing semantics. Since when a message is passed to the sieve object (on the lines marked by ->), no result is required, the sender is free to continue execution immediately after the message has been sent. Thus as soon as the candidate prime has been passed to the next stage, the current stage is ready to process another prime. In this way the whole sieve can operate in parallel.

An important point to notice is that the number of active threads in this example increases throughout the life of the program. This facility to dynamically create threads contrasts with languages such as Occam where the number of threads is fixed at compile time.

6.3.2 Support for Data Parallel Programming

In addition to providing control parallel constructs, Pyramid C++ includes data parallel support based on the object oriented techniques described in the previous chapter. This support is provided by in built parallel collection classes which provide a number of operations which are performed in parallel over all the data items in the collection. In keeping with the chosen application areas of image analysis and generation, these collection classes are organised as two dimensional arrays, generically called Images. This arrangement also corresponds with the SIMD array hardware provided by the Warwick Pyramid Machine.

6.3.2.1 Parallel Images

Images may be created in the same way as other objects, either statically by declaration, or using the C++ `new` operator. In either case the data for the image is distributed over the SIMD processor array, with one data item per processor. When an operation is performed on an Image, it is executed on all processors in the array in parallel. Thus if two images are added together all the corresponding pixels in the two images are added together in parallel to yield a third image.

Images may be declared to be any number of bit planes from one to thirty two. In particular one, eight, sixteen and thirty two bit deep images are supported as special classes called BitImage, ByteImage, WordImage, and IntImage respectively. The generic Image class allows numbers of bit planes between these values to be specified. Images are first class objects, that is they may be used in all situations where any other object may be used. Thus Images may be members of other objects, they can be passed as parameters to methods, and used in arrays.

The Image class and its subclasses define all the basic arithmetic and logical operations provided on conventional integers. In addition they define a number of operations which allow data to be moved within the array. To fit in with the nearest neighbour communication topology employed by the SIMD array, the movement operations provided allow the image to be shifted in one of four directions, referred to as north, south, east and west.

A simplified definition of the Image class is given below.

```
class Image{
public:
     Image();
     Image& operator+(Image&);
     Image& operator-(Image&);
     Image& operator*(Image&);
     Image& operator/(Image&);
     Image& operator<<(Image&);
     Image& operator>>(Image&);
     Image& S();
     Image& N();
     Image& E();
     Image& W();
}
```

The full Image class definition is much larger than this and contains many more operations, but this illustrates the fundamental ones.

To see how this can be applied an example program is given below which evaluates a Sobel edge detector. This implements two simple convolutions by the 3x3 masks shown

```
1    0   -1           1     2     1
2    0   -2           0     0     0
1    0   -1          -1    -2    -1
```

The results of these two convolutions give the edge strengths in the vertical and horizontal directions, which are then squared and added to provide the square of the total edge magnitude. A square root is normally not performed due to the computational complexity of this operation.

```
      main()
      {
a)      ByteImage im(512,512), sobel(512,512);
        ByteImage vert(512,512), horiz(512,512);

b)      horiz = im.N().W() + im.N()*2 + im.N().E();
c)      horiz-= im.S().W() + im.S()*2 + im.S().E();
d)      vert+= im.S().W() + im.W()*2 + im.N().W();
e)      vert-= im.N().E() + im.E()*2 + im.S().E();
f)      sobel = vert*vert + horiz*horiz;
g)      return(sqrt(sobel));
      }
```

The routine starts by declaring four eight bit images each 512 by 512 pixels (a). The variables are declared in the same way as ordinary objects, but the data that they contain is actually stored in the memory of the SIMD array, not the Transputer running the Pyramid C++. Despite this they can be manipulated just like ordinary variables, their implementation being completely hidden from the programmer.

The routine creates two intermediate results, one using the horizontal mask (`horiz`), and one using the vertical mask (`vert`). The intermediate results are calculated in two parts, first adding in the pixels corresponding to the positive mask values (b&d) and then subtracting the pixels corresponding to the negative mask values (c&e). These two values are then squared and added together (f). This produces the square of the vector magnitude, which is then square rooted and returned (g).

The expressions of the form `im.N()` are used to express nearest neighbour communications, by performing image shifts. The `im.N()` operation returns the image `im` but with each pixel replaced by the pixel to the north of it (north is taken as the direction of decreasing y, and east as the direction of increasing x). This definition was chosen because the other programmers who have used the system prefer to think of the operations as they affect each pixel. Thus the operation `im + im.N()` adds every pixel to its north neighbour. This model is similar to that used in the Apply language.

6.3.2.2 Conditional Expressions

Conditional expressions allow a subset of pixels within an image to be affected by a particular operation. This is achieved using a conditional assignment operation which is similar in functionality to the DAP FORTRAN matrix selection facility. The operation takes an image to be assigned to, and a boolean mask, and only those pixels in the image which correspond to a non-zero pixel in the mask image will be affected by the assignment. The mask image will often be produced as the result of a logical operation of another image.

For example an expression which replaces every pixel greater than a certain threshold value with the pixel to its north, would be written:

```
    image.where(image>threshold)=image.N();
```

The `where` construct is implemented as a member function of the base `Image` class, and does not represent an extension to the C++ language, it is provided using the existing object oriented facilities.

6.3.3 Support for Multi-SIMD Programming

The Image classes described above provide pure SIMD operations over the whole bottom level of the Pyramid Machine. The Pyramid machine also allows each cluster to act independently of the others, so called Multi-SIMD operation, which is supported by use of the `Cluster` actor class.

A cluster is an actor which represents a physical cluster. Each cluster is associated with one patch of the SIMD array. Operations on images are actually performed by the clusters, so that, for example, when two images are added together, the Image class broadcasts a request to all the clusters to add the patches in their part of the image together. A `Patch` is the class which corresponds to one patch of the image stored in a single physical cluster. The `Patch` class has exactly the same set of operations defined for it as the `Image` class.

All the operations defined for images map directly onto operations on the SIMD array, so that pure SIMD behaviour results. However the programmer can define new operations for the clusters which contain data dependent conditional code thus allowing the clusters to operate independently.

An example of this is the hierarchical Hough transform algorithm described in [Francis 90] . This has been implemented in Pyramid C++ and makes extensive use of its Multi-SIMD facilities. The hierarchical Hough transform performs locally independent Hough transforms on each patch independently, and then extracts the lines found in each patch to perform edge linking with neighbouring patches. These linked lines are then passed to a model based matching system.

To implement this it is necessary to be able to define a local Hough transform routine which can operate independently on each cluster. In Pyramid C++ this can be achieved by providing a Hough operation for the `Patch` class. This operation is implemented in terms of operations on patches, whose operations are implemented in parallel on the SIMD array, so the programmer does not need to be concerned with the low level access to array parallelism.

Having done this the programmer defines a global `Image` operation which performs the Hough transform over the whole image by making calls to the newly defined cluster operation. In this way the new operation appears to the application programmer just like any other SIMD image operation, although it is actually operating in a Multi-SIMD fashion.

6.3.4 Discussion

Pyramid C++ provides both control and data parallel operation in a single object oriented framework. Control parallelism is provided by autonomous actors which are distributed across multiple processors and communicate using message passing. Data parallelism is provided to allow the programmer to perform global SIMD operations using the `Image` class, and to use Multi-SIMD facilities using the `Cluster` and `Patch` classes.

Multi-SIMD operations are implemented by combining the control parallel actor facilities with the data parallel array classes. In this way the two facilities are combined to form a single coherent programming model for the Pyramid Machine.

6.4 Implementation

The implementation of the Pyramid C++ system is divided into three parts, the preprocessor, the parallel class library and the run time system. The preprocessor is responsible for translating the control parallel Pyramid C++ constructs into conventional C++. The C++ code generated by the preprocessor makes extensive use of the run time system to implement the control parallel primitives. This run time system is responsible for passing messages from Transputer to Transputer, and from the Transputer to the cluster controller. Finally the parallel class library contains the definitions for data parallel constructs, and their implementations. These implementations are divided into two sections, one of which runs on the Transputer, the other on the cluster controller. Each of these areas will be described in detail below.

6.4.1 Preprocessor

As described in a previous section, the control parallel constructs provided by

Pyramid C++ are provided by actor classes. The inclusion of these classes involve a number of semantic and some syntactic changes to the C++ language. These changes are implemented using a preprocessor which translates the various Pyramid C++ constructs into their equivalent in C++.

There are essentially two facilities required to implement the actor classes, multiple threads and message passing. Whenever an actor is created a new thread of control (or process) must be created, possibly on a different processor. Also whenever a call to a member function of an actor is made, it must be implemented as message passing. Creation of new threads and message passing are actually implemented by the run time system, but the preprocessor must generate code which calls the run time system at the appropriate points.

Stub Classes

To do this the preprocessor creates two so called 'stub' classes. The first of these is aliased to appear in the place of the actor class. It is responsible for the creation of new actors, and the sending of messages. The second of the stub classes is responsible for implementing the object controller which receives incoming messages and dispatches them to the appropriate member function. This scheme is shown diagrammatically below.

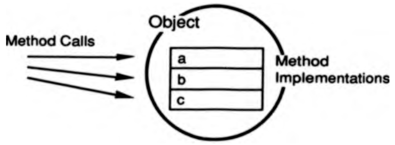
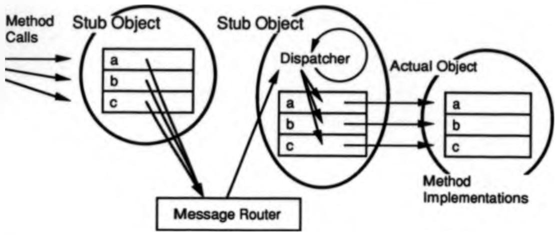


Figure 6.2a: Conventional Object



6.2b: Actor Implementation using Stubs

In the case of a conventional C++ passive object shown in the first diagram, method (or member function) calls are passed directly to the method implementations in the object.

If the object has been declared as an actor, all calls to its member functions are passed through the stub classes as shown above. This has the effect of replacing the conventional procedure calling mechanism with a message passing mechanism.

The first stub class has one method for each of the methods in the actor. These methods are responsible for converting the method calls into messages. To do this they take the parameters passed to the method and construct them into a packet. Then a unique method identifier is appended to the packet, which is then sent using the message router provided by the run time system to the second stub class.

The second stub object consists of two parts, the dispatcher, and a set of methods. The dispatcher is executed by the thread associated with the actor. It performs an infinite loop, repeatedly waiting for incoming messages, and then dispatching them to the appropriate method. As with the first stub class there is one method for each of the methods in the original actor. The methods in this stub perform the opposite function than those in the first.

They take the incoming packets and extract the appropriate set of parameters from them. They then call the appropriate method in the original object.

The originally defined actor object is implemented as a conventional passive C++ object, its autonomous behaviour being provided completely by the stub classes created by the preprocessor.

This can be illustrated in the following simple example. The definition of the actor is as follows (the method implementations will be ignored since they are not relevant to this explanation).

```
actor class example{
     int variable;  // actor's local data
public:
     example();           // constructor for actor
     int method1( int param1, char param2 ); // two methods
     char method2();
}
```

When passed through the preprocessor this will be converted into

```
class invis_example{
     int variable;
public:
     invis_example();
     int method1( int param1, char param2 );
     char method2();
}
```

Notice that the actor class has been replaced with a conventional C++ class. Also notice that the name has been changed to `invis_example`. This is done to hide the new class definition from the rest of the program. All references to it will be diverted to the newly created stub classes.

The first stub class has the form shown below

```
class example{
      Port port;
public:
      example();
      int method1(int param1, char param2);
      int method2();
}
```

Notice that the method names correspond to those in the original actor definition, but the data does not. This newly defined data will be used by the stub methods. Note that changing the data stored in the object is invisible to the rest of the program, because of the object oriented model.

The method called `example` is a special method called the constructor. It is called automatically by the C++ compiler whenever a new object is created. The constructor performs any initialisation of the object that may be required.

In this case the constructor is responsible for creating the actor. To do this it makes use of a number of routines in the run time system. The new constructor has the form shown below

```
example::example()
{
a)   Packet packet;       // variable declarations
     MemberPtr mp;

b)   mp=proc_example::p_example;
c)   packet<<mp;
d)   port=port_mgr.create(sizeof(proc_example),packet));
}
```

The constructor first declares two variables (a), a packet which will be used to send a message to the other stub object, and a member pointer, which is a special C++ pointer which stores the address of a member function. It then assigns to the member pointer the address of the pseudo-constructor for the other stub object (b)(see below) . This pointer will be used by the run time system to create the second stub object. The pointer is then appended to the packet (c)(using the << operator).

The last line of code (d) actually creates the remote stub, by calling the `port_mgr.create` routine in the run time system. This routine chooses an appropriate processor, and creates a new object on it. It does this by allocating storage of the size given to it (`sizeof(proc_example)`), and then calling the constructor associated with the stub class using the pointer passed to it in the packet. This constructor then initialises the storage.

The `port_mgr.create` routine returns a `Port`, which is a special object used by the run time system's message router. Ports are described more fully in the section on the run time system, but essentially they act as system wide addresses for actors. The port returned by the create routine is stored in the `port` variable declared in the stub class declaration. This value will be used by the other stub methods to send messages to the remote stub.

In addition to the constructor method, there are also stub methods for each of the original actor methods. These create packets containing their parameters and send them to the remote stub.

```
int
example::method1( int p1, char p2)
{
a)    Packet packet;
b)    MemberPtr mp;
c)    int retval;
d)    ReplyPort reply_port;

e)    packet << p2;
      packet << p1;
f)    packet << reply_port;
g)    mp = proc_example::p_method1;
```

```
     packet << mp;
h)   port.send(packet);
j)   reply_port.receive(packet);
k)   packet >> retval;
m)   return(retval);
}
```

The stub method declares a packet for the message (a), a member pointer to point to the remote method to be called (b), an integer to hold the value to be returned by the remote method (c) and a reply port(d). Reply ports are more fully described in the section on the run time system, but like ports they act as network wide addresses for messages. The reply port here is used to indicate the address to which the result should be sent, i.e. back to this routine.

The stub first appends the parameters to the packet (e). It uses the << operator defined for the `Packet` class, which is provided as part of the run time system. It then appends the reply port (f), which can be thought of as a return address for the result. Then it appends the address of the remote stub method `proc_example::p_method1` which will be called at the destination (g).

The packet is then sent to the destination given by the port which was returned by the create routine (h). Having done this it waits for a return value by performing a receive operation on the reply port (j). When the result packet is received the return value is extracted (k). This value is then returned to the caller (m), which will not be aware that the message passing has taken place.

The corresponding remote stub class is defined as shown below

```
class proc_example{
     Packet packet;
     MemberPtr mp;
     Port port;
     invis_example *object;
public:
     void p_example(Port p);
     void p_method1(Packet& packet);
     void p_method2();
}
```

Notice that this class also has one method for each method in the original class, but that these take Packets as arguments not the actual parameters. These routines are responsible for unpacking the packets.

The remote stub object is created automatically by the run time system. When created the pseudo-constructor `p_example` is called with its port as an argument. This port is the address at which incoming messages will be received.

```
proc_example::p_example(Port p)
{
a)   object = new invis_example();
b)   for(EVER){
c)        p >> packet;
d)        packet >> mp;
e)        (*mp)(packet);
     }
}
```

The pseudo-constructor first creates the object actually defined by the programmer as the actor as a conventional passive object (a). It then executes an infinite loop (b) which reads packets from its port (c), extracts the address of the appropriate method (d) and then calls that method with the packet as its argument (e).

Notice that the name used in the sending stub corresponds to the receiving stubs equivalent method. Similarly the constructor address passed in the first sending stub's constructor corresponds to the receiving stub's pseudo-constructor.

The receiving methods have the form shown below

```
proc_example::p_method1(Packet& packet)
{
a)   ReplyPort reply_port;
b)   int retval;
c)   int p1;
     char p2;

d)   packet >> reply_port;
e)   packet >> p1;
     packet >> p2;
f)   retval=object->method1(p1,p2);
g)   packet.reset();
h)   packet << retval;
j)   reply_port.send(packet);
}
```

The receiver first declares a number of variables, to hold the reply port (a), the return value (b) and the parameters (c). It then extracts the reply port (d) and the two arguments (e) from the packet. Then it calls the method in the original actor object, with the extracted parameters (f), assigning the returned value into the local `retval` variable. It then clears the `packet` ready for reuse (g) and appends to it the return value (h). It then sends the packet containing the return value to the reply port (j).

This description gives an overview of the system's functionality, however it should be stated that this is in fact a rather simplified explanation. Certain details have been left out which were considered unnecessary, and would only complicate understanding the functioning of the system.

6.4.2 Parallel Class Library

The data parallel constructs in Pyramid C++ are provided by a pre-written class library. This library implements a set of classes which provide a set of two dimensional array structures suitable for storing images. These array structures have the property that operations performed on them are carried out in parallel over all the elements in the array. To achieve this the elements of the array are mapped onto processing elements in the SIMD array which forms the lowest level of the Pyramid Machine.

The class library allows array objects whose data is stored within the memory of the SIMD array to be manipulated from within C++ as if they were normal objects with their data stored locally. To achieve this the implementation is divided into two parts. The first part is a set of classes implemented in C++ on the Transputer, which provide an interface to a set of primitive routines which run on the SIMD array. These routines which form the second part are implemented in CLASS (CLuster ASSembler) and perform the actual parallel operations on the array data.

The C++ image classes can be thought of as providing a layer of abstraction between the programmer and the SIMD array. The intention is that the array objects appear to the programmer to be as similar as possible to conventional scalar objects such as integers. Thus they should be able to be created and destroyed, passed as values, and operated on in just the same way as integers.

To achieve this the C++ classes must perform two main functions. Firstly they must provide a mechanism to associate each image object with an appropriate amount of memory in the SIMD array where the image data will be stored. Secondly they must define a set of operations, which when invoked, perform the appropriate operations on the data stored in the SIMD array.

6.4.2.1 Array Memory Management

With conventional objects memory allocation is performed directly by the compiler or run time system, but the compiler has no knowledge of the memory which resides within the SIMD array, and therefore the provided memory allocation operations used by the compiler are of no use to the array classes. To overcome this problem an equivalent set of memory allocation routines have been implemented for use by the array classes.

These routines must allocate an appropriate amount of array memory whenever an image object is created, and must conversely deallocate the memory whenever an image object is destroyed. C++ objects can be created in two ways, either implicitly by being declared, or explicitly using the new operator.

Similarly they can be destroyed implicitly by going out of scope, or explicitly by use of the delete operator. Fortunately C++ provides two special functions associated with each class, a constructor which is called whenever an object of that class is created, and a destructor which is called whenever an object of that class is destroyed. These routines provide the required interface to the memory allocation routines.

All the parallel image classes are derived from the base class Image. The constructor for this class, is responsible for allocating memory in the SIMD array to store the image data. It uses a bit mapped memory allocation scheme, which can allocate anything from a single bit plane to a thirty two bit deep image. The memory allocator returns an address, which is the address within the SIMD array memory where the image data is to be stored. This address is then stored in the C++ image object, so that when operations are subsequently performed on the object, the appropriate data can be acted upon. In a similar way, the destructor for the Image class is responsible for deallocating the memory associated with each image object.

In addition to allocating and deallocating memory, the Image class must also provide a mechanism for copying images. The C++ compiler often performs copies that are invisible to the programmer. For example when values are passed to functions, a copy operation is performed for all value parameters. Similarly values returned from functions must be copied. This is in addition to the more obvious case of assignment. Thus in order to make image objects behave just like scalars these copy operations must be performed. Again C++ provides a convenient handle for this, called the copy constructor.

A copy constructor is a special constructor which creates a new object which contains the same data as another object. If a copy constructor exists for a class, then the C++ compiler will use it whenever it needs to copy one object to another. The Image class provides a copy constructor which both allocates a new area of array memory, and copies the values from a specified existing object into that memory. Additionally the Image class provides an assignment operator which copies the data between two existing objects.

6.4.2.2 Image Operations

The second job of the C++ classes is to provide a set of operations, which are actually performed by the SIMD array. In keeping with the aim of making the image objects appear as much as possible like conventional scalars, the classes define all the usual arithmetic and logical operations that are defined for integers. All the defined operations make use of the operator overloading facilities provided by C++ to allow the usual infix operator notation to be used, so for example two images may be added by the expression `image1 + image2`.

The methods associated with each operation do not actually perform the operations themselves, instead they request the SIMD array to perform the operations on their behalf. These requests take the form of messages which are passed using the dual ported memory to the cluster controller. Each message specifies the operation to be performed, the addresses of the array memory on which to perform it, and the number of bit planes over which they should operate.

The information necessary to construct these messages is stored within the C++ image object. Each object contains the address of the SIMD array memory where its data is stored and the number of bit planes the data occupies. Each of the predefined methods also knows the address of the microcode routine which performs the operation associated with each of them.

To illustrate these ideas, a simplified class definition is given below

```
class Image{
     int address;   // address of data in SIMD array
     int bits;      // number of bit planes
public:
     Image(int b);        // constructor
     ~Image();      // destructor

     Image& operator+(Image& I);   // addition operator
}
```

This example class is shown with the single operator addition, although other operators work in a similar way. The constructor for the class is shown below

```
Image::Image(int b)
{
a)   bits=b;
b)   address=array_memory.allocate(bits);
}
```

The constructor is passed the number of bit planes in the image to be created in the parameter `b`, which it stores in the variable `bits` (a). It then allocates the appropriate amount of space in the SIMD array memory by calling the run time system routine `array_memory.allocate`, and stores its address (b).

```
Image::~Image()
{
a)   array_memory->free(address,bits);
}
```

The corresponding destructor calls the system routine which deallocates the array memory associated with the image (a).

```
Image&
Image::operator+(Image& I)
{
a)    Image result(bits);
b)    cl_controller->send(ADD,address,I.address,
                          result.address,bits);
c)    return(result);
}
```

Arithmetic operators in object oriented languages are interpreted as messages. Thus the expression 2+2 is viewed as meaning send the message +2 to the object 2. Similarly the addition operator defined above adds the object on which the operator is invoked, to the object passed to it in the message, and returns the result. This is why only one parameter is passed to the add operator, the other parameter is implicitly taken to be the object itself.

The method first creates a new temporary Image which will store the result of the addition (a). It then sends a message to the cluster controller, which instructs it to perform the addition (b). The message contains the address of the microcode add routine (the constant ADD), the address in array memory where the image data is stored (`address`), the address in array memory where the parameter's data is stored (`I.address`), the address where the result data (`result.address`) is to be stored and the number of bits on which the operation is to be performed, taken to be the number of bits in the object (`bits`).

The variable `cl_controller` in the above code is a special object which performs communication with the cluster controller. The communication takes the form of messages such as the one in the example above, as well as a small number of control messages. These messages are passed to the cluster controller via the dual ported memory by the run time system.

On the cluster controller is a dispatcher which reads the messages from the dual ported memory and executes the piece of code specified in the message. All the code on the cluster controller is written in CLASS.

6.4.4.3 Cluster Assembler: CLASS

CLASS (CLuster ASSembler) is the microcode macro assembler used to program the cluster controller and attached SIMD patch. It provides a one to one mapping from assembler instructions to microcode words. Each assembly language operation contains separate opcodes for each of the functional units (described in chapter 3), the sequencer, the ALU, the bus and the SIMD array.

The programmer must take into account all the interdependency between these functional units, and schedule instructions so that results are always available before use. For example results from the edge register are delayed by three cycles. For more details of the low level cluster controller programming see section 3.3.5 on the Cluster Controller Programming Model.

These kinds of complexities make it quite difficult to program in CLASS, and this combined with the limited amount of wide micro-code memory has influenced the design of the overall programming system. In the Pyramid C++ environment it is hoped that the applications programmer will not have to write any code in CLASS. The intention is that the operations defined on the Image classes within C++ will provide all the necessary functionality. Of course if performance is paramount then the option to add extra operations programmed in CLASS does still exist, but it is hoped this will not be needed.

6.4.3 Run Time System

The automatically generated actor stub classes require a message passing system to provide inter-object communication. This message passing is provided by the run time system. This system allows messages to be automatically routed from any process on any Transputer to any process on any other Transputer. Such a system is essential to the implementation of Pyramid C++, since the language allows objects to communicate arbitrarily with other objects in dynamically changing patterns. This allows programs to be topology independent, but does necessitate that the system support dynamic message routing.

6.4.3.1 Router

The message routing system must satisfy a number of constraints in order to be compatible with the semantics defined for Pyramid C++. The most important of these is that it should not introduce deadlocks into the system. This requires that the router itself cannot deadlock, and also that the router does not introduce parasitic dependencies and so cause a deadlock in a program which would not otherwise deadlock.

Deadlock Free Routing

Deadlock can occur in any concurrent system where a process can block waiting for a resource to be released by another process. In such a system it is possible for the situation to arise where two processes are blocked waiting for each other to release a resource the other requires. This situation is known as deadlock or deadly embrace.

There are two generally accepted ways of dealing with deadlock. The first is to detect when it has occurred, and intervene to stop it. The other is to prevent it from occurring in the first place. For deadlock to occur a cycle of dependencies must exist. This cycle could involve anything from a single process (waiting for itself) to an arbitrarily large chain of processes each waiting for the next with the last waiting for the first.

The prevention of deadlock therefore involves the prevention of such cycles forming. In systems where the topology of the dependencies can be determined in advance it can be possible to arrange the system statically such that it contains no cycles. This is done by constructing a dependency graph, whose nodes are processes, and whose arcs are dependencies to other processes. Thus a node representing a process will have one arc to each process that it can wait for. If the graph can be made acyclic then the system is guaranteed not to deadlock since no cyclic dependencies can occur.

In systems where this is not possible, dependencies may often be eliminated by the use of buffers. Taking the standard producer/consumer problem, which provides a good model for the message passing systems used here, the producer must always wait for the consumer to become ready to receive data before continuing. If a buffer is inserted between the two then, provided the buffer never becomes full, the producer will never have to wait for the consumer. In this way the dependency between the producer and the consumer can be eliminated (a similar result holds for the dependency from the consumer to the producer, but here the requirement is for the buffer not to become empty). Buffering has the double advantage of both avoiding deadlock and, in the cases where the buffers fill up, allowing easy detection of failure to prevent deadlock.

The router implemented by the run time system is divided into two layers, the first handles inter processor routing, while the second provides routing resolution down to the process level, and provides the port abstraction used by the automatically generated stub classes.

The inter-processor message passing is performed by a store and forward packet based router. This router is specifically designed to work with the four connected topology of the Transputer array. Because the topology of the system is fixed, it has been possible to guarantee that the router is deadlock free. Figure 6.3 shows a section of the dependency graph corresponding to four Transputers.

Each Transputer has four bidirectional links, one connected to each of its four neighbours. Each Transputer has eight processes, two associated with each link, one for the input half, the other for the output. The nodes in the diagram represent these processes.

The router has been implemented so that packets are always routed north/south before east/west, such that a packet is never transferred in an east/west direction unless it is already at the correct north/south position. Because of this the processes which receive messages from either the east or west never transmit them to the north or south, and so there is no dependency between them and the north/south processes

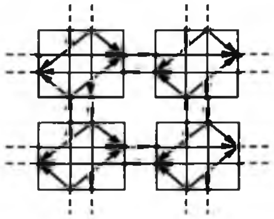


Figure 6.3: Section of dependency graph for router

This can be seen in figure 6.3. Each node represents a process, and each directional arc represents a dependency. Notice that the nodes associated with the incoming east and west links do not have arcs to either the north or south processes, whereas the nodes associated with the north and south links have arcs to both east and west as well as the other north or south link.

The omission of these critical dependencies makes this graph acyclic, i.e. starting at any node it is not possible follow the arcs and return to the same node. This acyclic nature guarantees the router to be deadlock free.

This however assumes that the inter-processor router is a closed system, when in fact it must communicate with the program which is passing the messages. The acyclic nature of the router's dependency graph assumes that messages that leave the system simply disappear. In reality these messages are delivered to the running program, which will also be sending messages back into the system. This immediately produces potential cycles, and these cannot be determined statically, since they will depend on the particular program. Therefore a second method is required to ensure that these potential cycles do not occur.

The second layer of the message router is responsible for this task. It makes use of buffers to decouple the inter-processor router from the user program so that no dependencies exist provided the buffers do not fill up. If the buffers should fill up, the router simply discards any incoming packets, and requests that the sender retransmit them. Whatever happens the inter-processor router is never blocked waiting for a user process.

Parasitic Dependencies

It is not enough however for the routing system itself not to deadlock, it must also not introduce unwanted dependencies into an otherwise deadlock free program. One obvious example of this is the use of a finite buffer to implement asynchronous message passing. In this case it is possible that all the buffer space in the system will be exhausted, at which point a process wishing to send a message will be blocked until buffer space becomes available. This will occur when a process accepts a message. However this introduces a random dependency between the process wishing to send a message and the process about to accept a message. This dependency will not be apparent to the programmer, and could easily cause an unforeseen dependency cycle, and possible deadlock.

This particular problem is solved by the use of semi-synchronous message passing semantics, where the sender is suspended until the message is accepted by the receiver. This places a finite limit on the total number of outstanding messages, and can therefore be implemented on a real system with finite memory.

Another common cause of these parasitic dependencies is the use of FIFO message delivery. If a processor accepts a stream of messages from a network in FIFO order, then if the message at the head of the queue cannot be delivered then neither can any message further back up the queue. Therefore the processes sending those messages become dependent on the process which is to receive the message at the head of the queue. To get around this problem the message router does not use FIFO buffering. Instead it has a buffer pool into which incoming messages are placed. Messages from this pool can be delivered in any order to their destinations, so avoiding any parasitic dependencies.

The implementation of the buffer pool poses an interesting problem, since it requires the ability to wait for one of a number of processes to become ready to receive. This process, known as select on output, is not supported by the Transputer. One possible solution is to pole each process in turn until one becomes ready, but this is inefficient since it consumes processor cycles that could be used elsewhere. The method chosen by this system is to assign a separate guardian process to each message. The job of this process is simply to wait until its message can be sent, and then deliver it. In this way the router as a whole can block simultaneously on an arbitrary number of processes. This implementation relies on the very low cost of generating processes on the Transputer, and may not be suitable for other systems

Ports

The final aspect of the routing system is that it provides an abstraction known as a port. A port is a many-to-one communications mechanism. There is at least one port associated with each process in the system. Associated with each port is a port address or handle. This handle specifies both the processor on which the port resides and the address of the port on that processor. The port handle is used to specify the destination of messages to the message passing system.

Many processes may possess the address of a single port, and may all send messages to it simultaneously. It is therefore necessary for the port to sequentialise the receipt of messages. If the message is delivered between processors, then sequential delivery is guaranteed by the message routing system. However if the destination port is on the same processor as the sending process, an alternative method is required. In this system ports are implemented using Transputer channels with semaphore protection.

A channel is a unidirectional message passing construct provided by the Transputer. It handles the transfer of the data and the scheduling to sender and receiver, however it is strictly a one to one mechanism. To allow multiple processes to send messages down a single channel, each port includes a semaphore. Before a process can send a message along the channel it must acquire the semaphore. In this way only one process can use the channel at once, which provides the necessary sequentialisation.

6.5 Conclusions

Pyramid C++ provides a coherent programming system for the Warwick Pyramid Machine. It supports both control parallel and data parallel programming concepts in a single unified model based on object oriented programming.

The language as implemented makes very few of the extensions required to support parallelism visible to programmers, allowing programmers already familiar with C++ to become proficient in the new language very quickly. This was possible because it was not necessary to introduce any radically new concepts in to the language, instead it was possible to make use of the ideas of encapsulation already present in object oriented programming to express the required parallelism. This can be taken as a measure of the power of the object oriented programming model.

6.5.1 Pyramid C++ in Context

Pyramid C++ was designed as part of the Warwick Pyramid Machine project, and its emphasis on support of both control and data parallel styles of programming reflect this fact. There are relatively few other architectures that support both MIMD and SIMD paradigms, and still fewer are sufficiently mature to have developed any particular style of programming. A number of the machines are programmed in two different languages, one to express control parallel parts of an application the other the data parallel part, others have no high level support for one of the two styles.

For example the IUA uses a custom dialect of Forth for programming the SIMD CAAPP and a dialect of LISP for its MIMD processors. PASM is proposed to use Ada for control parallel programming but has no high level language support for data parallel programming. Similarly the Disputer uses Occam for control parallel programming, but has no high level support for data parallel programming.

None of these systems provide a single coherent programming model for applications programmers. Proposals have been made to provide some support based around one of the control parallel languages such as Ada or Occam, by extending those languages to include array primitives. Pyramid C++ provides all the functionality that such an approach would give, but does so in a coherent way based on the single object oriented model, rather than the simple juxtaposition of two separate models.

In the absence of any directly comparable languages it seems worthwhile comparing the support provided by Pyramid C++ for parallel programming with a selection of both data and control parallel languages.

Control Parallel Support

The control parallel constructs are clearly related to those provided by the other actor based languages such as POOL, Concurrent Smalltalk and ACT++.

A crucial design decision was the choice of C++ as the base language for the Pyramid C++ system and the use of the existing C++ object oriented features to implement them.

Languages such as POOL and SINA are new and untried languages that have no existing programmer base. This seems an unnecessary hurdle for them to cross, since as the design of Pyramid C++ shows it is possible to provide the facilities that these languages provide without the recourse to totally new languages. A significant amount of the design and implementation time, not to mention the learning time for such languages is taken up on trivial details, such as the syntax for control structures, which surely are not valid topics of research when perfectly adequate languages already exist.

Systems such as Concurrent Smalltalk and Distributed Smalltalk which extend the Smalltalk language, while avoiding the problems of a totally new language, inherit the problems of their base language. Smalltalk, while a fine language for rapid prototype development has never proved popular for real applications due to its inefficiency (it is a partially interpreted language) and would have been a wholly unsuitable choice for the real time image analysis problems the WPM is aimed at.

ACT++ and Concurrent C++ have the advantage of being based on C++, an efficient compiled and above all popular object oriented language, but peculiarly fail to capitalise on their base language's facilities. Both provide separate and orthogonal facilities for concurrency, one based on actors the other rendezvous and tasking, which are not related to C++'s object oriented facilities. This is particularly curious in the case of ACT++ which provides an essentially object oriented model of concurrency, yet using its own separate facilities.

Pyramid C++ avoids all these problems by allowing existing C++ programmers to use the new language with little extra learning, so that their effort can be concentrated on the real task of learning how to design parallel algorithms. Also note that none of the other languages has any support for data parallel programming, and so are not directly comparable to Pyramid C++.

Data Parallel Support

The data parallel support is provided by special parallel image classes. These classes provide functionality very similar to those found in DAP-FORTRAN, but unlike this language they do not require any extensions to the base language. All the required facilities can be provided within the framework of the object oriented facilities of C++.

This approach has the advantage that further data parallel classes can be added which provide extended functionality, without the necessity of redefining the language, and all the problems that that entails. Obvious examples of this include the xector construct used in CmLISP, which could be provided by adding additional parallel xector classes within the existing class framework.

These data parallel classes can create abstractions for the applications programmer just as straightforward to use as those provided by languages such as DAP FORTRAN where specific language extensions have been used.

For example addition can be provided with the usual infix plus operator, and variables can be created by simply declaring them. In languages such as IPC these functions must be performed using calls to library routines, so making the underlying implementation of the data parallel facilities more visible to the programmer.

The same note applies here that these data parallel languages do not support control parallelism, and so do not provide a suitable language choice for the WPM irrespective of their data parallel capabilities.

6.5.1 Portability Issues

In the process of developing the hardware and software for the Pyramid Machine it was found to be very useful to be able to run application programs on a conventional MIMD machine, in this case an array of Transputers. This was achieved by providing a sequential implementation of the parallel image classes that would normally be implemented on the SIMD array. These provide exactly the same operations as their parallel counterparts, but are implemented conventionally in C++ on the Transputer.

The object oriented model used by the Pyramid C++ system guarantees that programs will perform in the same way with either the sequential or the parallel class implementations provided both conform to the same class definitions.

This process has now been extended to a completely sequential implementation of the Pyramid C++ system which uses light weight processes to support control parallel constructs and the same sequential implementations of the image classes to support the data parallel constructs. This sequential system runs on a conventional SUN workstation and allows programs to be written which make use of all the facilities of the Pyramid Machine, even when the prototype machine is not available.

This ability to provide alternative implementations, which make use of whatever parallelism is available on the host architecture provides interesting possibilities for the generation of portable parallel software. Already this system provides portability between sequential, MIMD and Multi-SIMD machines. It should also be possible to provide an implementation on a pure SIMD machine, which would implement the data parallel constructs in parallel, and use multi-tasking to implement the control parallel constructs.

This opens the way for a single programming language which could be used to write software for all these types of machines, and which would allow such software to run on any of the others. Further work would be necessary to ensure that efficiency is maintained between architectures, but this seems a promising start in that direction.

Chapter 7

Application Study

7.1 Introduction

In this chapter two applications will be examined with reference to how they can be mapped onto the Warwick Pyramid Machine using the object oriented techniques described in previous chapters. One of these applications is from the field of image understanding, the other image generation. Both are examples of applications which exhibit the heterogeneous structure characteristic of image based applications. In both cases processing can broadly be divided into two parts, one which deals with image data and involves fairly simple and global processing, and the other which deals with high level model based data, and involves more complex local processing.

These applications are not contrived specific cases, but represent realistic problems which occur in real applications. The fact that these typical applications have a structure which is compatible with the heterogeneous pyramid architecture of the Pyramid Machine, supports the argument that the Pyramid Machine architecture is well matched to a fairly general class of image based applications.

7.2 Image Understanding: Object Alignment

The image understanding application chosen concerns determining the rotational alignment of a simple geometric object from a single side on image. This application is fairly typical of a variety of real time image understanding problems, and can be generalised for use in real applications.

The geometric objects considered are cuboids, and cuboids with two sloping faces. These shapes were chosen as they are closely analogous to a large class of real man-made structures such as vehicles and buildings. The techniques used for this application were developed in the context of a real time image understanding application described in [Atherton 90b]. This demonstrates that these techniques can be generalised to perform alignment determination on real objects such as vehicles.

To build a system which can achieve such a task it is necessary to determine the set of *characteristic features* of the object under rotation, that is those visual features that change depending on the rotational alignment of the object. This set of features will depend on the type of object being processed, in this case a simple geometric shape, and the type of rotations that the object experiences, in this case perpendicular to the image plane.

Many image based systems use the shape of the object as the characteristic feature. This approach works particularly well for rotation in the x-y plane as might be expected in an aerial view.

7.2.1 Shape Based Techniques

Many two dimensional orientation problems can be solved by considering the shape of the object being processed [Gonzalez 77]. Most of these systems determine the outline of the object, using some segmentation process, and then use this outline to obtain the orientation of the object. A number of techniques exist to do this, including radial projection, axis projection, moments and template matching.

Moments - If the object is fairly asymmetric then the major and minor axes as determined from the moments will give the orientation directly.

Radial projection - First the centroid of the object is found, then a graph is generated of the distances of the outline from the centroid versus angle of projection. This graph can be matched against stored versions, and the relative phase of the graph with the matched version will give orientation.

Axis projection - First the major and minor axes are found, then the object is convolved with a line perpendicular to each of the axes in turn. The results are plotted on two graphs. These graphs can then be matched with stored versions. The orientation is given by the major and minor axes.

Template matching - This involves storing a template of the object at every orientation and simply matching the image against all the stored templates. This technique, while simple, involves large storage and is not scale invariant.

7.2.1.1 Limitations of Shape Based Techniques

Shape based techniques are generally good at detecting rotation in the x-y plane since the outline of an object will usually change under this kind of rotation. However if the rotation is in the x-z plane the shape of the outline may not change at all under rotation, see below.

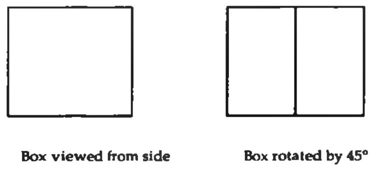


Figure 7.1: Rotation in x-z plane

This simple example is closely analogous to the problem under consideration, where we have a side on view of an object. Clearly in these cases shape based approaches are not appropriate, and a more sophisticated approach is required. One such approach is model based understanding.

7.2.2 Model Based Techniques

A general purpose vision system takes an image and generates a model of the scene which is represented by that image. This extremely difficult problem can be greatly simplified if, as in many practical applications, a model of the scene can be generated in advance. If this is the case, then the vision problem becomes one of matching the image to a known model [Huttenloch 87] [deFig 87][Hourard 87b].

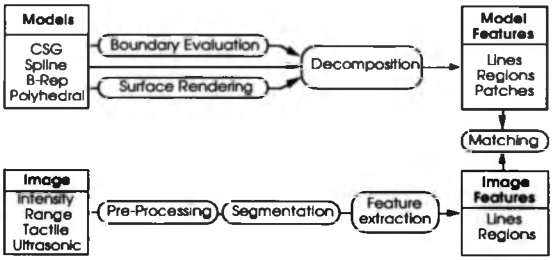


Figure 7.2: Model Based Processing

The process, shown in Figure 7.2, consists of two parts [Bhanu 87]. One takes a model of the scene/object and performs a series of transformations on it to produce a set of features. The other proceeds from the input image and processes it using conventional image processing techniques to produce a set of compatible features. The features from the two processes are then matched. Once a match has been obtained, the transformation used to convert the stored model into the matching set of features gives the necessary information about object orientation and position.

To implement such a system many design decisions are required, such as the representation of the stored model, the features that are to be extracted from it and the image, and the matching technique that is to be used.

7.2.2.1 Model Representation

The choice of model representation will depend on the features that are to be generated from it. In this application those features must include the characteristic features of the object under rotation. Some examples of commonly used model representations are given in figure 7.3 and are described below.

Constructive Solid Geometry (C.S.G.) - This technique involves representing the model as a collection of geometric primitives, such as cylinders and cubes. Each primitive can be scaled, rotated and translated, and some systems allow primitives to be subtracted. C.S.G. has the advantage of requiring very little storage, since each primitive can be represented by just a few numbers. However these models typically require more computation to produce a set of image features than the other methods described here.

Boundary Representations (B-Rep) - [Watt 89] Here only the visible surfaces of the object are represented rather than its substance. The two most common representations used for B-Rep are patches and planar polygons. Patches are curved surfaces defined by a mathematical function, usually a quadratic, e.g. a sphere or cylinder, or a bi-cubic equation, which allows almost arbitrary curves. More sophisticated representations include Non-Uniform Rational B-Splines (NURBS), which can represent an even richer set of curves. If less accuracy is required then curved surfaces are often approximated to by sets of planar polygons. This is probably the most common form of representation, and certainly the simplest.

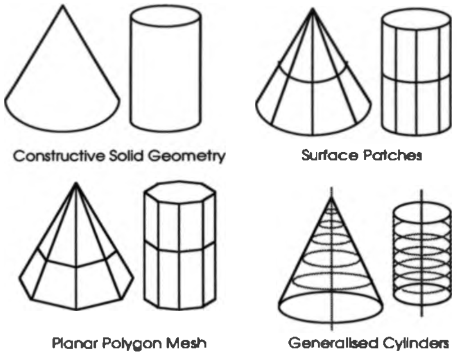


Figure 7.3 - Model Representations

B-Rep in general requires more storage than C.S.G. but requires less processing. Planar polygons in particular while less compact than curved patches require very simple processing.

Generalised Cylinders - or extrusions represent objects as the volume swept by a two dimensional shape moved along an axis [Bhanu 87]. The size of the shape may be varied as a function of its distance along the axis. This technique can be useful for representing many types of objects such as tubes and girders which are manufactured using extrusion processes.

7.2.2.2 Model Matching

Once the features have been generated from the model they must be matched to the corresponding features extracted from the image. This process typically uses a Hypothesis-Evaluation-Backtracking approach which involves generating a hypothesis for the transformation from model to image, based on certain cues in the image [Horaud 87a]. This hypothesis is then evaluated to generate a certainty of its correctness. If the evaluation fails then the process backtracks and tries a different hypothesis. Ideally the evaluation process will provide some information to help the system determine a better hypothesis.

The initial hypothesis is typically based on specific cues in the image, such as corners or lines or closed areas. These cues are determined in advance to be the most important features, whose position determines the position of the other features in the object. They are also chosen to be features that are relatively easy to extract from the image.

Once the initial hypothesis is generated other expected features can be tested to determine a certainty measure. This is either done by projecting other image features into model space and matching then against the model features in model space, or conversely taking other model features and projecting them into image space and matching in image space [Brisdon 88]. Matching in image space has the advantage that features that may not be strong enough to be extracted from the image can still be evaluated, to give a certainty value, even if that certainty is low. If model space matching is used, any weak features will be ignored.

The matching process should produce a measure of how close the current hypothesis is to a match, and also some indication of how to improve it. The system then either accepts the current hypothesis or backtracks and makes an improved estimate.

7.2.3 Implementation

The problem under consideration falls into the category of problems not solvable by shape based techniques, since it involves a side on view of the object. Therefore the chosen implementation uses a model based approach to the problem. However a key requirement of the system is to operate at very high speed, since it must coexist with other system components in a real time application. Therefore emphasis has been placed on speed over generality, and the model based techniques used are very specific to this application.

Apart from this aspect, the implementation falls into the conventional pattern of low-level processing followed by segmentation and feature extraction, followed by the extracted features being matched against an equivalent set of model features, with the model transformations giving the rotation alignment of the image object.

The first part of the system design centres around the choice of the representation for the model, the set of characteristic features, and the matching algorithm. As has been mentioned it is not possible to isolate each decision from the others since they are highly interrelated, but the most critical item is the choice of characteristic features, which largely determine the other two.

7.2.3.1 Choice of Characteristic Features

As shown in figure 7.1 it is not possible to determine rotational alignment of an object from the outline alone, so the system must take image features internal to the object into account. In the geometric shapes chosen the most important of these is the vertical edge formed by the nearest corner of the cuboid, and its position relative to the extreme edges of the object.

Taking the simple cuboid which has a known aspect ratio (figure 7.4(a)) the rotational alignment can be determined simply from the position of the vertical edge formed by the nearest corner, with respect to the outline of the whole cuboid. Thus these positions are suitable features to extract from the image, and they can be used to calculate the rotational alignment by matching them with an equivalent set of features extracted from a suitable model.

This measurement will however be ambiguous about 90° because the system has no way to differentiate between the long and short sides of the cuboid.

As the cuboid rotates, the apparent length of the two sides changes, and at certain rotations the longer edge of the cuboid will appear shorter than the short one. With no other information it is not possible to determine which edge is which, but this is often sufficient in certain applications.

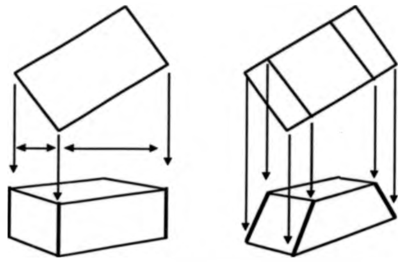


Figure 7.4: Affect of Rotation on Side on Image

If the problem is extended to objects which do not have parallel sides, such as that shown in figure 7.4(b), more precise alignment estimation is possible. The alignment for these shapes can again be determined from the relative positions of the object's edges, but in this case the slopes of those edges can also be used. The internal edge always slopes in the same direction as the visible sloping face, which gives the necessary information to discriminate the two visible faces.

Using this fact the angle may be determined within 180 degrees, although it is still ambiguous with respect to a 180 degree rotation, or in other words it is not possible to tell the front from the back. Nonetheless this is still a useful result in the chosen application, and therefore these provide a suitable set of characteristic features for these objects.

7.2.3.2 Image Processing and Feature Extraction

The overall function of the feature extraction process is to determine the presence or absence of the internal vertical edge which corresponds to the nearest corner of the object. It must also determine the slope and position of the edge relative to the object's outline. This process proceeds in a number of stages.

Filtering and Edge Detection

The first stage in the process will generally be low pass filtering to reduce any excess noise which might disrupt the performance of the edge detector. This filtering would take the form of a global image convolution with an appropriate mask. However this step will only be required where noisy images are expected. In the images used in this study no filtering was necessary, but with some imagery it has proved useful.

The preprocessing stage is followed by a Sobel edge detector, which consists of two 3x3 convolutions, which give the vertical and horizontal edge strengths separately. These two values can be combined to give the strength and direction of the edges in the image.

Both these operations are global iconic operations which are implemented on the SIMD layer of the Pyramid Machines by a series of global convolutions.

Detection and Segmentation

Following the edge detection, the system must detect the presence and position of the object in the image. This is achieved using a Spoke filter, which is a simplification of the circle Hough transform [Atherton 90b]. The basic operation of the Spoke filter is to project spokes out perpendicularly from all edge points, such that where many edge points lie in a roughly circular pattern, many of these spokes will intersect. Thus if the resultant spoke image is thresholded, the rough centroid of any roughly circular region will be given. In practice squares are 'roughly circular', so this provides a suitable technique for this application.

Segmentation, involves determining the exact outline of the object, which is achieved using a spoke segmenter. This projects spokes outwards from the centroid determined by the Spoke filter, until they cross an edge. These edge intersections are then used to give the overall size of the object. They can also be projected back onto the image, by applying a convex hull operation, to yield an object mask. This can be used to restrict subsequent processing to the interior of the object.

Feature Extraction

Having determined the position and extent of the object, it is then possible to extract the position of the internal vertical edge.

Simple Cuboid - in this case feature extraction can be achieved by using an edge density histogram. This involves passing a vertical mask over the edge detected image of the object, and accumulating the values of the pixels that fall within the mask.

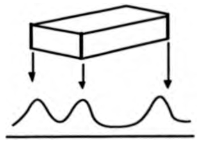


Figure 7.5: Edge Histogram

This gives a histogram that contains peaks at the positions of both the ends of the object and the internal edge. This histogram can then be processed using an algorithm such as Superspike [Beveridge 88], to determine the centres and number of peaks in the histogram. This is then used to directly calculate the rotation of the object without use of a model matching phase. This approach is however very sensitive to noise, and only works in ideal conditions. In the system that handles the more complex case of the cuboids with sloping faces a full model based approach is used.

Cuboid with Sloping Faces - the techniques required for the cuboid with sloping ends are slightly more complex, since both the position and slope of the internal edge are required. This is achieved using an extension of the edge density histogram technique, whereby three such histograms are extracted from the image. Each histogram represents the strength of edges at one of -45°, 0° or 45°, as shown in figure 7.6.

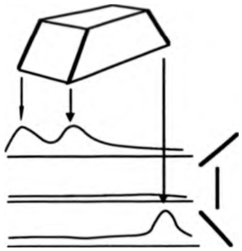


Figure 7.6: Three Oriented Edge Density Histograms

This is done in three stages, one for each histogram. The histograms are calculated in the same way as before, but this time the masks used are slanted, by either -45°, 0° or 45° as appropriate. This is further enhanced by testing the orientation value of each pixel, given by the Sobel operator. Only those pixels whose orientation lies within a set tolerance of the orientations being looked for i.e. 0°,45° or -45°, are accumulated into the relevant histogram. This approach gives greatly improved discrimination between edge slopes, by concentrating the energy of edges at a given slope into a small region of the histogram.

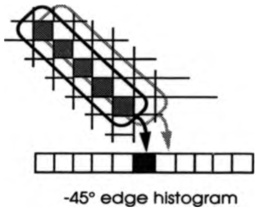


Figure 7.7: Extracting Orientated Histograms

The implementation of this technique on the SIMD array involves collapsing the object pixels down onto the base line of the object. The base line is the lowest row of pixels within the object, which is determined by the segmenter.

The histograms are built up by repeatedly summing the row of pixels immediately above the baseline (for the 0° histogram, or diagonally above for the 45° and -45° histograms) into the histogram. At the same time the object pixels are all shifted down one row of pixels (and along one for the 45° and -45° histograms), so the subsequent summation will use the next row of pixels.

This is illustrated in the simplified Pyramid C++ routine shown below

```
histograms(
        Image &result1,&result2,&result3,
        int baseline, int height)
```

```
{
     ComplexImage temp;
     BitImage mask, angle_mask;

a)   mask=(y_coord==baseline);
b)   temp=*this;
c)   for(int i=0;i<height;i++){
d)        temp=temp.N();
e)        angle_mask=(temp.angle> -22)&&(temp.angle< 22);
f)        result1.where(mask&angle_mask)+=temp.mag;
     }
     temp=*this;
g)   for(int i=0;i<height;i++){
          temp=temp.N().W();
          angle_mask=(temp.angle> -67)&&(temp.angle< -22);
          result2.where(mask&angle_mask)+=temp.mag;
     }
     temp=*this;
h)   for(int i=0;i<height;i++){
          temp=temp.N().E();
          angle_mask=(temp.angle> 22)&&(temp.angle< 67);
          result3.where(mask&angle_mask)+=temp.mag;
     }
}
```

This routine creates three histograms in images `result1` to 3. It is passed the y coordinate of the bottom of the object and the height of the object (produced by the segmenter) . The routine first creates a mask which is set to true for all the pixels on the baseline (a). This uses a special global image called `y_coord`, whose pixel values are their own y coordinates. Then it creates a temporary copy of the current image (b) and loops over the height of the object (c) producing the histograms.

This involves three steps, first the temporary image is shifted down (d). Then a new mask is created which selects all those pixels whose angles are within 22 degrees of vertical (e). The last step is to sum those pixels which are selected by the ANDing of both masks into the current histogram(f);

This process is repeated twice more (g,h) for -45° and 45°, the only difference being the angles that the mask selects and the direction the image is shifted in.

The histograms created by this process contain all the necessary information to determine the object's rotational alignment, i.e. the positions and slopes of all the edges. These histograms form the symbolic representation of the object which can be passed up to the symbolic processor for matching.

7.2.3.4 Model Representation

In a conventional model based system a model would be stored in one of the forms described in section 7.2.2.1 and would be processed to extract the desired features at run time. This allows the system to process any object which can be represented in the chosen form.

However in this application performance has been taken as being the overriding constraint, and so it was crucial to develop a technique tuned specifically to this application, which could make use of specific information about the application to produce a fast implementation. To this end a less general but faster technique has been adopted which is based around a procedural model.

A procedural model can be thought of as a program which directly generates a set of features, without the use of a stored model. Fractal generated mountains are a typical example of this technique, where a simple equation can directly produce a complex visual output with no need to actually store a model of a mountain.

In this case the procedural model consists of a set of equations which describe the behaviour of the edges formed by the corners of a cuboid with sloping faces, given a specific angular rotation. The procedural model produces the position of the three visible edges of the object, and their angle from the vertical.

These results are then passed to a synthetic histogram generator, which produces a set of histograms with peaks at the positions specified. The edge energy is divided appropriately between the three oriented histograms according to the angle from vertical. These values approximate to those which would be expected from an image of edges at the angles specified. These histograms can be generated at any size to match the size of the histograms extracted from the image.

7.2.3.5 Feature Matching

A suitable matching process is to perform a direct correlation between the histograms extracted from the image and the synthetic histograms. This involves pairwise multiplying the values in the real and synthetic histograms and then summing all the results. The most closely correlated histogram should correspond to the best matched orientation.

Before this is done both histograms are scaled so that the total area of both histograms is equal to zero, by subtracting the average histogram value from each histogram entry. This ensures that the total weighting of all histograms is equal, otherwise histograms with high background values, such as those produced from a noisy image, tend to produce good matches, even if the peaks of the histograms are not a good positional match. By choosing zero as the constant value, small values are scaled to negative values. This means that the absence of a peak where there should be one, or the presence of one where there should not be one counts negatively. Without this technique histograms with many spurious peaks can produce good matches.

The matching process is implemented on the symbolic processors. Since the rotational alignment is only required to fairly crude precision (5 degrees) and that the model is ambiguous about 180°, there are only thirty six possible angles. It is therefore quite feasible to match the extracted histograms against all possible angles in parallel.

To do this a separate actor is generated for each synthetic/real histogram pairing. Thus three actors are generated for each candidate angle. These actors are automatically distributed over the available processors under control of a main control actor. This actor is responsible for extracting the symbolic histogram data from the image, and distributing it to the matching actors.

The histogram data extracted from the image amounts to at most 300 bytes or so. This compares with 16000 bytes for the iconic image representation. This data reduction is typical in image understanding problems, and is reflected in the reduction in the number of processors in the symbolic layers of the Pyramid Machine architecture.

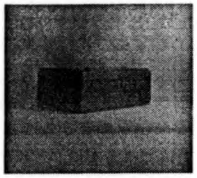
Once extracted the histogram is broadcast to all the matching actors. Each actor then generates its synthetic histogram at a scale to match the size of the histogram extracted from the image. The extracted histograms may have anything from around one hundred entries to as little as ten. The two histograms are then correlated, and results of all the correlations are sent to a coordinating actor which determines the closest match. This match should correspond to the rotational alignment of the object.

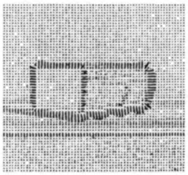
A measure of the certainty of the match can be produced by comparing the best match with the next closest. If the two values are very different then the match has a high certainty, whereas if the difference is very small it has a lower certainty. This measure can be used to give a percentage rating for certainty, along with a second choice and second certainty measure.

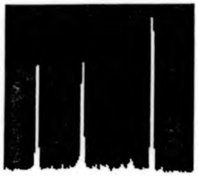
7.2.4 Results

The algorithms described above have been implemented, and tested on a variety of images. Shown here are some of the results of these tests.

Figure 7.8 shows results for the simple block case. Panel (a) shows the original image, panel (b) shows the corresponding edge orientation image produced from the Sobel operator. In this image each pixel is represented as a vector, which has length proportional to the edge magnitude, and direction perpendicular to the edge direction. Panel (c) shows the histogram extracted from the orientation image. Panel (d) shows the alignment estimation displayed diagrammatically as a plan view of the object.


a) Original Image


b) Orientation Image


c) Extracted Histogram

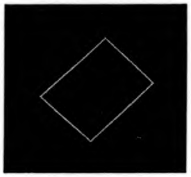
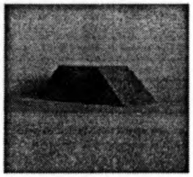
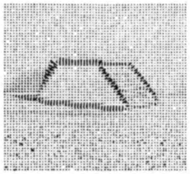
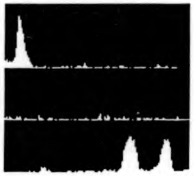
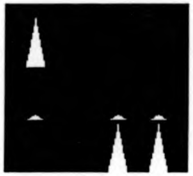

d) Estimated Alignment

Figure 7.8 - Simple Cuboid Alignment Estimation Results


a) Original Image


b) Orientation Image


c) Extracted Histogram


d) Matched Synthetic Histogram

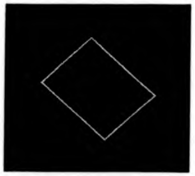

d) Orientation Estimate

Figure 7.9 - Alignment Results for Cuboid with Sloping Faces

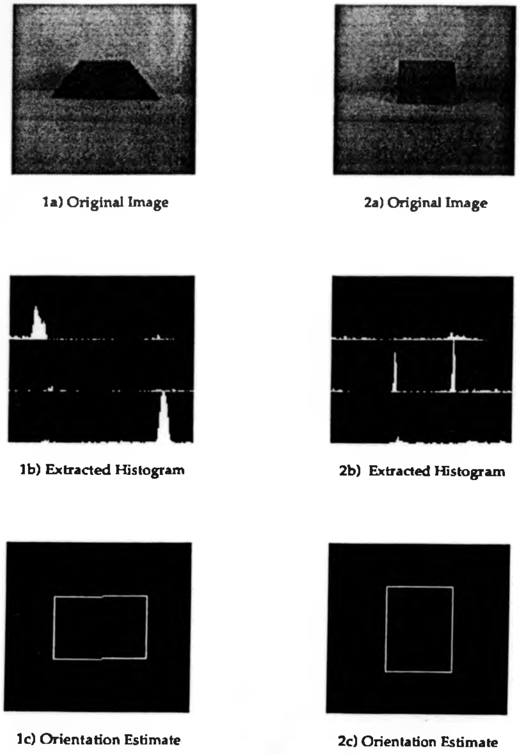


Figure 7.10 - Alignment Results for Cuboid with Sloping Faces

Figure 7.9 shows the more complex case of a block with sloping ends. Panel (a) shows the original image, panel (b) shows the oriented edge detection for the image. Panel (c) shows the three histograms extracted from the image, and panel (d) shows the best match synthetic histogram. Finally panel (e) shows the estimated orientation diagrammatically as a plan view.

Figure 7.10 shows similar results for two other orientations of input image. Here panels (1a&2a) show the original image, panels (1b&2b) show the three histograms extracted from the image, and panels (1c&2c) show the estimated orientation diagrammatically as a plan view.

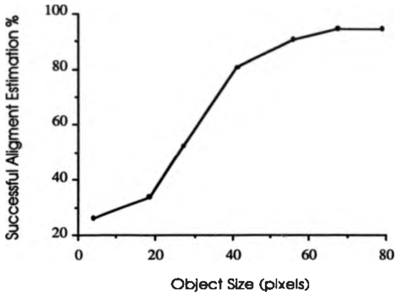


Figure 7.11- Alignment Performance against Object Width

Figure 7.11 shows the results obtained from running the alignment estimator on twenty sequences of seven images where the object was a different size in each image of a sequence. Success is defined as producing the correct answer to within the resolution of the system which in this case is 22°. As can be seen the results are reasonably good down to 40 pixels or so. After this the results degrade sharply as the internal features become indistinguishable. This is to expected as the algorithms rely heavily on the internal features of the object in the image. Thus these algorithms should only be used on images where the object is sufficiently large that internal features are distinguishable.

Correct Alignment Estimation (%)	Signal to Noise Ratio (dB)
0	30
5	30
10	30
15	60
20	60
25	80
30	80
35	90

Table 7.12 - Alignment Performance against Noise

Table 7.12 shows the results obtained from running the alignment estimation on ten of images with artificially added Gaussian noise. As can be seen the performance drops off smoothly as the noise increases, although it still achieves a 60% success rate at 15dB. In most cases the signal to noise ratio will be far better than this. If noise is a problem for a particular application then further image preprocessing could be used to enhance the alignment performance.

Timings

The overall timings for this process can be broken down into a number of key tasks, preprocessing, detection, segmentation, feature extraction and matching. The times for these operations on a typical image, estimated based on simulations, are given below.

Task	Algorithm	Time(μs)
Preprocessing	Sobel	44
Detection	Spoke filter	82
Segmentation	Radii Segmenter	810
Feature Extraction	Edge Histogramming	1810
Histogram Generation	Procedural Evaluation	74
Matching	Correlation	106
TOTAL		**2926 or 2.9 ms**

As can be seen from this table processing at video rates of one frame per 20ms should be easily achievable. In practice there may be some additional overhead for these figures when the system is run on a fully configured system, however it may be possible to offset these by more aggressive optimisation of the algorithms used, since these figures are based on straightforward implementations of these algorithms.

7.3 Image Generation: Polygon Rendering

Image generation can be thought of as the reverse process of image understanding, instead of starting with an image and producing a model, one starts with a model and produces an image. As with the image understanding example given above, the choice of model representation is an important factor. Image generation systems use very similar model representations to those described previously for model based image understanding, in particular the constructive solid geometry and boundary representations are particularly popular. The example used here is an application based on the most popular and simplest of these representations, the planar polygonal mesh.

There are a number of rendering algorithms, that is methods for transforming a stored model into an image, in common use. Notable amongst these are ray tracing and radiosity which can produce startlingly realistic images, but are very computationally intensive and even using the large computational power of a massively parallel computer cannot be computed at speeds even approaching real time. On conventional architectures these methods often have computing time measured in hours rather than seconds.

Because of this extreme computational requirement, most image generation systems use other less time consuming processing techniques. Most systems are based on the so called viewing pipeline described below. The viewing pipeline is not a single algorithm, but a collection of algorithms which if performed in sequence will produce an image of a stored model. For a good treatment of this topic see [Watt 89] or [Foley 82]. Hardware rendering systems such as Clark's geometry engine [Clark 82] translate the viewing pipeline into a physical hardware pipeline to perform real time rendering.

7.3.1 The Viewing Pipeline

The viewing pipeline can be divided into two halves, one dealing with high level model based data, the other with low level image data. This fits exactly into the expected image based application structure, and as such maps neatly onto the Pyramid Machine, with the early parts of the pipeline being implemented on the symbolic processors, and the stages being implemented on the iconic processors.

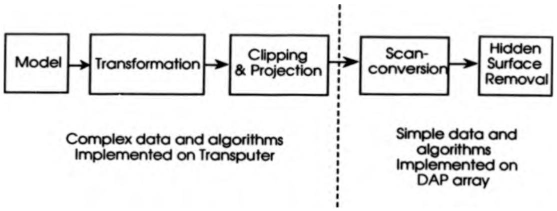

Model
Transformation
Clipping & Projection
Scan-conversion
Hidden Surface Removal
Complex data and algorithms Implemented on Transputer
Simple data and algorithms Implemented on DAP array

Figure 7.13: The Viewing Pipeline

7.3.1.1 Transformation

The first stage in the viewing pipeline is the transformation process which converts the model coordinates from model space into viewing space, as shown in figure 7.14.

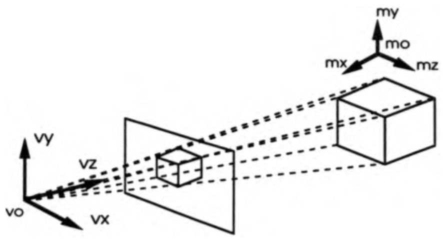


Figure 7.14: Projecting a Model into Image Space

The model's coordinate system has the origin *mo*, and the axes *mx,my* and *mz*. The viewer's coordinate's space has the origin *vo* and axes *vx,vy* and *vz*. The viewer's axes and origin are expressed as vectors in the model's coordinate system. Each vector consists of x y and z components, so for example the viewer's origin vector *vo* consists of vo_x, vo_y and vo_z and so on.

The transformation of a point *p* from model space to viewer's space is given by.

$$\begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \begin{bmatrix} vx_x & vx_y & vx_z \\ vy_x & vy_y & vy_z \\ vz_x & vz_y & vz_z \end{bmatrix} \begin{bmatrix} p_x - vo_x \\ p_y - vo_y \\ p_z - vo_z \end{bmatrix}$$

The viewer's origin is first subtracted from the point, giving a vector from the viewer's origin to the point in model space. This vector is then premultiplied by the matrix formed by the axes of the viewer's coordinate system, expressed in model space. This gives the vector in the viewer's coordinate space.

7.3.1.2 Clipping and Projection

The next stage in the pipeline is the clipping and projection process, which is responsible for taking the transformed model and projecting the visible parts of it onto an imaginary projection screen between the viewer and the model, known as the image plane, as shown in figure 7.15.

The first part of this process is to determine which parts of the model are visible to the viewer, and which are hidden, and then remove those parts which are hidden, a process known as clipping. This is done by intersecting each part of the model in turn with the clipping volume. Parts of the model which fall outside this volume are deemed to be invisible and are discarded. This process may involve dividing some polygons into a number of pieces if some parts of the polygon fall inside the clipping volume and other parts fall outside.

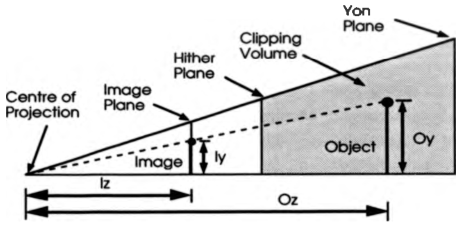


Figure 7.15: Perspective Projection onto the Image Plane

The clipping volume itself is formed by six planes. Four of these planes correspond to the top, bottom, left and right extents of the image. These planes pass through the centre of projection *vo* and four pairs of adjacent corners of the image plane. The other two planes are parallel with the image plane and determine the maximum and minimum visible depths within the viewer space. These planes are known as the yon and hither planes, for the farthest and nearest extents respectively.

The visible parts of the model must now be projected onto the image plane. The viewer's coordinate system is defined such that the x and y axes are parallel with the x and y axes of the image plane, and the z axis passes through the middle of the image plane. One possible projection would be to simply ignore the z coordinate and take the x and y coordinates as the two dimensional point in the image plane. This is known as parallel projection, but it does not produce a very satisfactory result. Most systems use the perspective projection system shown in figure 7.15.

Given a point in the viewer's coordinate system at Oy,Oz (or Ox,Oz) the projection onto the image plane Iy,Iz is given by the intersection of a line drawn from the point Oy,Oz to the origin (known as the centre of projection) with the image plane. Since the distance from the origin to the image plane Iz is known this can be calculated using similar triangles.

$$\frac{Oy}{Oz} = \frac{Iy}{Iz} \quad \text{so } Iy = \frac{Oy}{Oz} * Iz$$

7.3.1.3 Shading and Scan Conversion

The two dimensional polygons, now in image coordinates, must be drawn onto the output image, a process known as scan conversion. There are a number of sequential algorithms for scan conversion which will not be covered here. However the essential function of these algorithms is to determine which image pixels fall within the boundaries of each polygon, and to assign an appropriate value to those pixels. This value will normally represent the colour or intensity of the polygon. This must be determined from the shading and illumination model.

The intensity or brightness of each polygon can be determined from four factors, the surface properties of the polygon, its incidence to the illuminating light sources, the properties of these light sources and the incidence of the viewer to the polygon. The properties of the surface, such as its 'shininess' and colour, are stored in the model database and are generally static. The incidence of the polygon to the light source can be calculated given the position in view space of the light sources. The properties associated with the light sources, such as intensity, size and colour are also stored in the model database and are static. The incidence of the viewer to the light source can be calculated from the polygon's position in viewer's space, given that the viewer is always at the origin of the view space.

Given these factors the intensity of a polygon may be determined by

$$I = I_a k_a + I_l[k_d (L \bullet N) + k_s (R \bullet V)^n]$$

Where

I_a	= intensity of ambient light
k_a	= surface coefficient for ambient light
I_l	= intensity of incident light
k_d	= surface coefficient for diffuse reflection
L	= light source vector
N	= surface normal
k_s	= surface coefficient for specular reflection
R	= reflected light vector
V	= vector to viewpoint
n	= surface 'shininess' coefficient

The simplest shading system assigns the same intensity value to all the pixels within the polygon, and is known as constant shading. In many cases however the original polygonal mesh is actually an approximation to a curved surface. If this is the case then constant shading will give what should be a smoothly curving surface a faceted appearance.

To overcome this many systems use what is known as smooth shading, based on either Gourard [Gourard 71] or Phong [Phong 75] models. These systems assign different intensity values to every pixel in the polygon to give a smooth appearance. This shading is only an approximation however, since there is no way for the system to know how the original surface really curved, it only has access to the polygonal approximation.

Gourard shading calculates the intermediate values by linearly interpolating the intensity values calculated at each vertex across the polygon. Phong shading on the other hand calculates intermediate surface normals by linearly interpolating the vertex normals across the polygon, and then recalculates the intensity of each pixel from the intensity equation given above. The result of this is that Gourard shaded images have a permanent matt appearance while Phong shaded images can accurately represent surface properties such as shininess.

7.3.1.4 Hidden Surface Elimination

The final stage of the rendering process is hidden surface elimination. This is responsible for determining which parts of the model are hidden by other parts of the model which are in front of them. There are a number of algorithms for doing this, but they can mostly be broken down into two main classes, model based and image based. The model based systems attempt to order the polygons in the model so that those furthest away are drawn first, so that polygons in front of them will be drawn on top.

This is essentially a sorting process, but it is complicated considerably by the fact that there is in many cases no ordering that will achieve the desired result. This is because polygons can span a range of depths so that a polygon may be in front of another which is in turn in front of another which is partially in front of the first. In other words the relation 'in front of' is not transitive so A in front of B and B in front of C does not imply A in front of C. To overcome this the polygons must be divided into pieces such that the relation does hold, and the pieces can be sorted correctly. It is this division process which causes much of the complexity for model based algorithms.

Image based systems attempt to determine the polygon that is visible from each pixel in the image. The most common of these is known as the z-buffer algorithm. This involves storing the current depth of each pixel and only plotting values which are closer than the current depth. The depth of a pixel is simply the depth of the appropriate point on the polygon currently visible at that pixel. When a new polygon is plotted the depth of each pixel to be plotted is compared to the depth stored at each pixel, if the new value is smaller (closer) than the current depth, the pixel is replaced with the new value and its depth is updated, if not the pixel is left untouched.

7.3.2 Implementation

The implementation maps the polygon renderer described above onto the Pyramid Machine. As mentioned earlier the high level model based parts are mapped onto the Transputer array while the low level image based parts are mapped onto the iconic processors. A principal design aim has been to allow the system to scale effectively, so that use of a larger machine will result in improved performance.

One approach to mapping the rendering pipeline onto an MIMD array [Theoharis 87] is to use one processor for each stage in the pipe. This approach has two disadvantages, firstly it does not scale readily, one of the primary aims, since the number of pipeline stages is fixed to the number of stages in the rendering process. Secondly it involves the polygon data being transmitted between each stage in the pipeline, which in an MIMD machine where communications involve a significant overhead causes a severe bottleneck. In addition to this the pattern of communication implies that the processors be arranged as a pipeline, which does not correspond to the WPM architecture, making it an unsuitable choice.

7.3.2.1 Processor Farm

The approach taken here is to use a static 'processor farm'. This involves distributing the polygon database amongst the available Transputers in advance, such that each processor has an equal number of polygons. Since the processing required for each polygon is fairly constant this gives an even load balancing across the array.

For each new frame the viewing parameters and lighting details are broadcast to all the processors. All the processors then perform the transformation into view space, any necessary clipping and the projection onto the image plane for their set of polygons. Each polygon is then sent to the symbolic processor which is associated with the region of the screen in which that polygon falls. To do this it is necessary to clip each polygon to the cluster boundaries. This involves polygons which span more than one cluster being broken into several pieces figure 7.16.

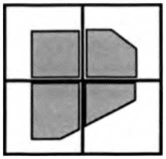


Figure 7.16: Clipping Polygons to cluster Boundaries

This arrangement allows the system to utilise an arbitrary number of symbolic processors, up to one processor for every polygon. It also gives an essentially linear speed up as the number of symbolic processors increases.

Each symbolic processor then passes the polygons sent to it to its associated cluster controller which performs the scan conversion and hidden surface elimination independently of all other clusters. This is only possible because of the Multi-SIMD architecture of the WPM and provides an excellent match to this application. On a pure SIMD architecture it would only be possible to render one polygon at a time, with the WPM each cluster can render polygons in parallel.

Pyramid C++ Implementation

A small section of the Pyramid C++ implementation code is given below which should serve to illustrate how the concepts described above can be mapped on to the Pyramid Architecture using Pyramid C++. This example code is shows the structure of the solution. Some code is omitted for the sake of clarity.

```
1)  actor class Scan_converter{
2)    void scan_convert(Patch); // omitted
    }

3)  actor class Renderer{
4)    void put_polygon(Polygon); // omitted
5)    void render(View,Light);
    }

6)  Renderer::render(View view, Light light)
    {
7)     for(int i=0;i<number_of_polygons;i++){
8)         poly=polygon[i].transform(view).project(view);
9)         poly=poly.shade(light);
10)        list=poly.clip(view);
11)        for(int j=0;j<list.elements();j++){
12)            scan_converter=list(j).scan_converter;
13)            scan_converter.scan_convert(list(j).patch);
           }
       }
    }

14) main()
    {
15) Polygon polygon[number_of_polygons];
16) Renderer renderer[number_of_renderers];
17) Scan_converter scan_converter[num_of_scan_converters];
```

```
18)      read_polygons();
19)      for(int i=0; i<number_of_polygons; i++){
20)         renderer[j].put_polygon(polygon[i]);
21)         j=(j+1)%number_of_renderers;
22)      }
23)      for(EVER){
24)         get(view,light);
25)         for(i=0;i<number_of_renderers;i++){
26)              renderer[i].render(view,light);
            }
         }
      }
```

This code segment defines two actor classes, the first is the `Scan_converter` actor class (1) which is responsible for actually drawing the polygons on the screen. This class has (in this simplified example) one method which scan converts one polygon, which is passed to it (2). The other actor class is the `Renderer` class (3), which is responsible for all the earlier stages in the viewing pipeline. This class has two methods, the first takes a polygon that is passed to it and remembers it for later use (4), the other renders all the polygons the render has stored for a given viewpoint and light source (5).

The code for the `render` method is shown (6) above. It iterates over all the polygons it has stored (7) and for each one performs transformation and projection for the given viewpoint (8). Then it shades the polygon (9) for the given light source. Next it clips the polygon to the cluster boundaries using a routine not shown here, which returns a list of patches (10), one patch for each cluster that the polygon overlaps.

The routine then iterates over all the patches that (11), and extracts the scan converter from the list (12) which was calculated by the clipping routine. Finally the patch is sent to the given scan_converter for scan conversion (13).

The main body of the code (14) performs the overall system control. It declares three arrays which store the model (15) and the renderers (16) and the scan convertors (17). The main routine first read in the model (18) and then iterates over all the polygons in the model (19).

It sends each polygon to a renderer (20) and then moves on to the next renderer (21) being sure to wrap round after the last renderer is reached. In this way the polygons are distributed evenly between the renderers.

After this the routine goes in to an infinite loop (23) which reads a view point and a light source (24) and then sends this to all the renderers (26) which will then render their polygons. This process will repeat for subsequent view points and light sources.

This code represents a fairly natural implementation of this application, and there is little additional complexity as a result of it being a parallel implementation. Indeed this program will work perfectly satisfactorily as a sequential C++ program, simply by deleting the actor keywords. This demonstrates how easily and naturally the object oriented model can be used to express this kind of control parallelism.

7.3.2.2 Scan Conversion

There are two existing methods for scan conversion on an SIMD array, one developed for the Pixel Planes architecture by Fuchs [Fuchs 89] the other developed for the DisArray architecture by Theoharis [Theoharis 87]. Both of these map well onto the SIMD layer of the Pyramid Machine.

Fuchs' Algorithm

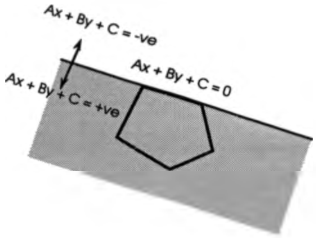


Figure 7.17: Fuchs' algorithm - one step

The Fuchs algorithm proceeds in a number of stages, taking each edge of the polygon to be rendered in turn. 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 negative result. All the pixels with a positive result are flagged as candidates, while all the pixels with negative values and thus not inside the polygon are eliminated.

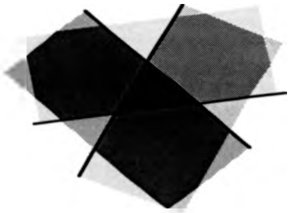


Figure 7.18: Polygon Formed by Intersection of Regions

When this process is repeated for all the edges, those pixels which are marked as candidates for all edges correspond to the pixels within the polygon. These are then marked for subsequent colouring (figure 7.18).

Pre-Stored Patches

The alternative method proposed by Theoharis is based on the use of precalculated patches. Theoharis noticed that the number of possible paths an edge can take through a single 16x16 patch is 1040. This is sufficiently small that it is feasible to pre-compute a complete set of masks which includes every possible polygon edge. Each mask consists of an edge which divides the mask into two parts, one part all ones the other all zeros. These masks are conceptually similar to the half spaces created by Fuchs algorithm. In the same way a number of these masks can be combined such that the polygon formed by the intersection of them is formed.

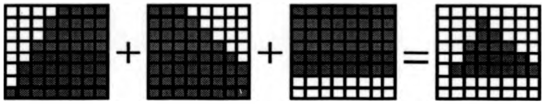


Figure 7.19: Scan Converting with Pre-computed Masks

A number of techniques are used to reduce the number of masks which need to be stored. For example it is not necessary to store both positive and negative versions of each mask, since inversion is extremely quick on the SIMD array. Also masks can be shifted vertically or horizontally very quickly to form other masks.

Hidden Surface Elimination

Hidden surface elimination is performed using the Z-buffer system. Each processor stores the current depth of the pixel at that position. When the interior pixels of each polygon have been determined using one of the methods described above, those processors associated with interior pixels compares the depth of the new polygon with the existing depth. Only if the new depth is less than the old do they take the value of the new polygon.

To determine the depth of each pixel within the polygon the processors evaluate the equation $Ax + By + C$ as used in the Fuchs algorithm. However in this case the coefficients A,B and C describe the plane $Ax + By + C = z$, where z is the depth of the polygon at the point x,y.

A similar technique can be used to implement Gourard shading. Here the coefficients are chosen to represent the plane $Ax + By + C = I$, where I is the intensity of x,y. It is even possible to extend this technique to Phong shading, although this involves computing the illumination equation of every pixel in the array.

7.3.2.3 Results

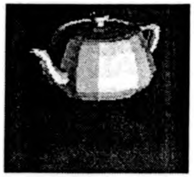
Some typical images produced by this system are shown in figure 7.20. All these images can be produced at TV frame rates by a fully configured pyramid machine. The limiting factor in this system is the high level processing performed by the Transputer array.

a) Space Shuttle

b) Wine Glass

c) Kline Bottle

d) Volkswagen Beetle


e) Teapot

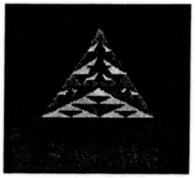

d) TetraPyramid

Figure 7.20 - Example Output of Polygonal Mesh Renderer

The times to render the example images using pre-computed patches, estimated using the Transputer based simulator, are as follows

	Polygons	Scan Conversion Time (ms)	Total Time (ms)
Tetra Pyramid	36	0.06	0.1
Space Shuttle	389	0.6	0.9
Wine Glass	360	0.6	0.9
Kline Bottle	192	0.3	0.4
VW Beetle	1018	1.7	2.5
Teapot	1184	1.9	2.9

A single cluster can scan convert approximately 20,000 triangles per second using Fuchs' algorithm or roughly 40,000 triangles using pre-computed patches (at the expense of over 1000 bitplanes of array memory). A full machine containing 64 clusters could scan convert up to 1.2 or 2.4 million triangles per second. This assumes that all clusters are fully active all the time which is optimistic, however in practice the system still achieves approximately 600,000 triangles per second. Based on these results the SIMD array can always scan convert at a rate greater than the Transputers can produce transformed polygons, making the Transputers the limiting factor.

Based on these results the system can be seen to render approximately 400,000 triangles per second. This allows all the example images to be rendered at TV frame rates by this system.

7.4 Conclusions

The two example applications described in this chapter provide representative examples from the two principal application areas at which the WPM is targeted. Both these applications map straightforwardly onto the architecture and do, as hoped, exhibit a clear division into iconic and symbolic processing. In both cases it has been possible to make full use of the dual paradigm nature of the WPM to provide an implementation which would have proved difficult to achieve as effectively on an architecture which did not support the dual paradigm model.

Unfortunately it has not been possible to include direct comparisons between the WPM and other architectures on these tasks, because no figures are available for the performance of these algorithms on other architectures. Since the algorithms described here are not standard benchmarking algorithms for which independent results might be available, the only way to produce comparative results would be to run these algorithms on other architectures. This has not been possible, partly because of lack of access to other architectures, and partly due to lack of a portable programming environment across comparable architectures, which would necessitate a complete rewrite of all software for another architecture.

This lack of comparative performance figures for parallel architectures is a generic problem, and it is one which urgently needs further work. The principal problem is a lack of a portable way of programming parallel architectures of varying designs. This prevents simple benchmarking programs like those used on sequential architectures being produced for parallel machines.

One possible solution to the portability problem is to publish sets of algorithm independent tasks to be performed on an architecture such as the DARPA image processing benchmark [Weems 88]. However this approach still has problems, as the freedom to use any algorithms, given to avoid unfairly favouring a particular architecture, means that it is not clear whether the timings produced are the result of the architecture or good or indeed bad algorithm design.

A realistic comparison of the performance of the WPM and other architectures would involve a full benchmarking study being carried out. This is a major piece of work in its own right, and is beyond the scope of this thesis.

Chapter 8

Conclusions

8.1 Discussion

This thesis has been concerned with the design of a complete parallel system aimed at real time image based applications. This has included not only the design of appropriate hardware but also both system and application software. The key motivation behind this work has been the desire to produce a system which can overcome the limitations of present parallel systems which have concentrated on either fine grain SIMD, or coarse grain MIMD configurations, by combining these two models into a single coherent system.

8.1.1 Hardware

It has been recognised for some time that applications such as image analysis and image generation contain vast amounts of intrinsic parallelism, and that they present a computational challenge beyond that which can be met by sequential methods. This has made them highly appropriate applications for solution by parallel means. The exact approach to this solution however is still very much open to debate.

The two most common approaches are based on massively parallel fine grain SIMD arrays and highly parallel coarse grain MIMD arrays. Neither of these solutions are completely satisfactory however as both exhibit performance that is highly dependent on the exact structure of the algorithms used.

The work described in this thesis is based on the observation that the structure of image based problems falls into a particular pattern, which corresponds to the transformation of the data set from an iconic representation to a symbolic form. The basis of the object oriented approach to computation is that the structure of problems tends to be determined by the nature of data on which operations are to be performed. If this perspective is applied to the image analysis application one can see that data is transformed from a highly regular iconic representation through a more loosely structured numeric representation to a potentially irregular model based structure. The image generation case is similar but reversed.

From this one would expect that the nature of computations performed on these different data types to also change as the data moves through these various stages. Notice that the exact application is not the driving factor in this analysis, but instead the structure of the data used. It is argued that *any* image based application will exhibit this basic structure, regardless of its exact function since it must support these basic data structures. These arguments have been used to design software systems which are stable over time, even under dramatic functional changes, and are the basis of object oriented design techniques.

In this case however these arguments have been applied to optimise design of the hardware architecture to a particular application. By analysing the data types encountered at different points in the chosen applications it has been possible to produce an architecture optimised for the general class of image based applications rather than a specific one.

As described in the preceding chapters this architecture is based on a combination of a fine grain SIMD array and a coarse grain MIMD array, which are coupled via an array of communications and control processors to allow high bandwidth inter array communication. This architecture maps exactly to the expected image application structure. SIMD arrays with orthogonal communication are an ideal match for regular image data, while coarse grain MIMD arrays with general routed communications are ideal for the more complex structure of model based data, and its more complex interrelationships. Thus by combining these two approaches it is hoped that the architecture achieves a high degree of compatibility to a wide range of image based applications, and so combines the performance advantages of highly optimised special purpose hardware with the flexibility of general purpose programmable systems.

8.1.2 System Software

Such a radical approach to parallel architecture design, requires an equally radical approach to programming. Since the vast majority of systems are based on single paradigm parallel hardware, the existing software systems follow the same pattern. This has therefore required the development of a novel programming system, which is also based on the object oriented design approach.

The key to the object oriented approach to programming is to provide data types which provide appropriate metaphors for the real world objects which are being manipulated. In this case these objects represent abstract entities such as images and models, which correspond to the various phases of processing involved in an image based application.

Associated with each type of data structure is a particular style of processing. In the case of image data this tends to involve globally uniform operations with local communications, whereas for model data it tends to be more local processing with more global communications. The objects used to represent the different data structures support the appropriate type of operations. Thus image objects support globally uniform array operations, whereas model data is represented using actors which support autonomous local operations and arbitrary global communications.

These two styles of processing are brought together in a single language, Pyramid C++, which uses its object oriented facilities to provide all the necessary parallel programming constructs, in a single uniform model. In this way the applications programmer is largely insulated from the precise details of the hardware, and is instead presented with a natural set of parallel programming tools which match the intended problem areas.

8.1.3 Application Software

Two specific example applications have been described which are fairly typical of the types of problem that can be tackled with this dual-paradigm approach. These demonstrate that realistic applications do indeed fit into the prescribed processing model and can be represented using the object oriented design system. They also give some preliminary guide to the kind of performance levels one can expect from this hardware arrangement, although it should be stressed that at this stage neither the hardware nor the software has been fully optimised.

It should also be pointed out that in addition to the applications given in this thesis other researchers have mapped other algorithms onto this architecture using the Pyramid C++ object oriented model, most notabley the hierarchical Hough based recognition system [Francis 90] which further substantiates the claim that this architecture matches the structure of image based applications.

8.2 Contribution of this Work

The work described in this thesis has produced a viable heterogeneous parallel architecture which provides an alternative to conventional approaches to real time image based systems. In particular this work offers significant advantages over existing approaches where real time performance combined with algorithmic flexibility are crucial, and where the application falls into a iconic/symbolic framework.

The development of a working prototype machine is particularly important since it demonstrates that such an approach is feasible using available components.

The work has also resulted in the development of a novel programming language, Pyramid C++, which provides a coherent programmers model of an architecture which combines two conventionally separate parallel computational models, namely MIMD and SIMD. Pyramid C++ is the first programming language to address this dual paradigm programming model, making it an excellent language for use on the WPM.

By making use of object oriented techniques Pyramid C++ successfully unifies the two principal parallel programming models into the single unified concept of autonomous internally parallel objects. These objects allow not only control and data parallelism to be described in the same language, but as importantly it allows the two techniques to be neatly combined to provide solutions to heterogeneous problems such as image understanding and image generation. These capabilities are unique to Pyramid C++, and are the result of the unified approach taken.

The implementation of a number of realistic applications using this applications development environment has demonstrated that this system provides a convenient platform for image based applications. It has also demonstrated how real applications can take advantage of multiple heterogeneous computational elements to optimally support the variety of computational styles found in image based applications.

8.3 Implications for Future Work

This work has been based on a specific application area, namely image based systems, but it seems likely that similar techniques could be applied to other application areas. In particular the combination of control and data parallel programming in a single model seems to provide a valuable programming tool for many systems.

Currently the programming system used depends not on the problem to be solved, but on the machine architecture on which it is to be implemented. This seems a poor way to choose a programming style. In reality many problems are naturally solvable using either a data parallel or control parallel approach. This is independent of the proposed hardware platform, but is inherent in the structure of the problem to be solved.

Many real applications contain a wide variety of subproblems, each of which may have its own natural structure, such as the low-level image processing task, and high-level model matching problems found in image analysis applications. Thus an ideal parallel language should support *both* parallel programming styles. It then remains to map these programming styles onto the underlying hardware.

In the case of the Warwick Pyramid Machine this mapping is done by simply implementing the data parallel constructs on the SIMD array and the control parallel constructs on the MIMD array. This works very well for the image based tasks studied, and may be satisfactory for other applications.

An interesting extension to this work would be to apply this dual paradigm programming style to other problems and study the distribution of control parallel and data parallel constructs in 'natural' implementations of these problems, that is implementations written without regard to the underlying hardware. Such a study would provide a number of interesting results, such as the relative importance of control and data parallel support, and how these interact.

Such a study could provide a sound basis to decide whether the Pyramid Machine as it stands is applicable to other problem areas, or whether modifications are required, or indeed whether an alternative approach is required. Whatever the outcome though, it seems very likely that both control and data parallel components will be required to efficiently implement a dual paradigm programming model, and in turn it seems that such a programming model is the only way to allow algorithms to be expressed in their most natural form, which should lead to their most efficient hardware solution.

Bibliography

[Abram 85] *"VLSI-Architectures for Computer Graphics"*, Microarchitecture of VLSI Computers, NATO ASI Series on Applied Science, No 96, 1985

[Agha 86] *"Concurent Programming using Actors"*, G. Agha, C. Hewitt, Object Oriented Concurrent Systems, MIT Press, 1986

[Ahuja 86] *"Linda and Friends"*, S. Ahuja, IEEE Computer, August 1986

[Aida 90] *"Compiling Concurrent Rewriting onto the Rewrite Rule Machine"*, H. Aida, J. Goguen, J. Meseguer, SRI International Technical Report SRI-CSL-90-03, February 1990

[AMD 87] *"Am29C331 CMOS 16-Bit Microprogram Sequencer"*, AMD, Data sheet, Dec 1987

[AMD 87] *"Am29116 CMOS 16-Bit ALU"*, AMD, Data sheet, Nov 1987

[AMT 88] *"DAP 500 Technical Overview,"*, Active Memory Technology, April 1988

[AMT 90] *"DAP Series Technical Overview, Introducing the DAP/C8 range"*, Active Memory Technology, Sales Support Note 7, April 1990

[America 86] *"POOL-T - A parallel object-oriented language"*, P. America, Object Oriented Concurrent Systems, MIT Press, 1986

[Annot 90] *"POOL and DOOM: The Object Oriented Approach"*, J. Annot, P. den Haan, Parallel Computers, Ed. P. Treleavan, Wiley, 1990

[Atherton 90a] *"A Scalable Multiple-SIMD Architecture For Real-Time Image Understanding"*, T.J. Atherton, G.R. Nudd, N.D. Francis, D.J. Kerbyson, R.A. Packwood, G.J. Vaudin, IEE Colloqium on 'The role of image processing in defence and military electronics', London, April 1990

[Atherton 90b] *"Detection and Segmentation of Blobs using the Warwick Multiple-SIMD Architecture"*, T.J. Atherton, G.R. Nudd, S.C. Clippingdale, N.D. Francis, D.J. Kerbyson, R.A. Packwood, Y.K. So, G.J. Vaudin, D.W. Walton, SPIE/SPSE Symposium on Electronic Imaging Science & Technology, Santa Clara, February 1990

[Barton 87]	*"Data Concurrency on the Meiko Computing Surface"*, E. Barton, Proceeding of BCS Parallel Processing for Display, London, May 1987
[Batcher 80]	*"Design of the Massively Parallel Processor"*, K. Batcher, IEEE Transactions on Computers, Sept 1980
[Batcher 82]	*"Bit-Serial Parallel Processing Systems"*, K. Batcher, IEEE Transactions on Computers, VOL C-32(5), May 1982
[Beal 90]	*"Floating Point Support for SIMD Array Processors"*, D. Beal, C. Lambrinoudakis, QMW College University of London Technical Report 511, November 1990
[Bennett 87]	*"The Design and Implementation of Distributed Smalltalk"*, J. Bennett, Proceedings OOPSLA, 1987
[Berge 89]	*"An Implementation of the Object Oriented Concurrent Programming Language SINA"*, A. Tripathi, E. Berge, Software Practice & Experience, VOL 19(3), 235-256, March 1989
[Bershad 88]	*"PRESTO: A System for Object Oriented Parallel Programming"*, B. Bershad et al, Software Practice & Experience, VOL 18(8), 713-732, August 1988
[Besl 85]	*"Three-Dimensional Recognition"*, P. Besl, C. Ramesh, Computing Surveys, VOL 17(1), March 1985
[Beveridge 88]	*"Segmenting Images using Localized Histograms and Region Matching"*, J.R. Beveridge, J. Griffith, COINS Tech Report 87-88, October 1987
[Bhanu 87]	*"CAD-Based 3D Object Representation for Robot Vision"*, B. Bhanu, C. Ho, IEEE Computer, August 1987
[Briot 89]	*"Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment"*, J. Briot, Proceedings ECOOP, 1989
[Brisdon 88]	*"Feature Aggregation in Iconic Model Matching"*, K. Brisdon et al, Proceedings Alvey Vision Conference 1988
[Brooks 85]	*"Performance Evaluation of the Butterfly Processor-memory Interconnection in a Vector Environment"*, E. Brooks, Proceeding of the International Conference on Parallel Processing, 1985

[Burn 90] *"Implementing Lazy Functional Languages on Parallel Architectures"*, G. Burn, Parallel Computers: Object Oriented, Functional, Logical, J. Wiley 1990

[Chesney 87] *"The Computing Surface as a High Perfromance Graphics Computing Server"*, H. Chesney, Proceedings Parallel Processing for Vision and Display, London, May 1987

[Clark 82] *"The Geometry Engine: A VLSI Geometry System for Graphics"*, Computer Graphics 16(3), July 1982

[Cox 86] *"Object Oriented Programming: An Evolutionary Approach"*, Addison Wesley, 1986

[Dahl 66] *"SIMULA - An ALGOL-based simulation language"*,O. Dahl, B. Myhrhaug, ACM VOL 9(9), September 1966

[Davis 88] *"System Architecture Concepts for the BLITZEN Massively Parallel Machines"*, E. Dvis, J. Rosenburg, Tech Report TR88-30, MCNC, December 1988

[Duff 81] *"Languages and Architectures for Image Processing"*, M. Duff, S. Levialdi, Academic Press, 1981

[Eggers 89] *"Evaluating the Performance of Four Snooping Cache Coherency Protocols"*, ACM Computer Architecture News, VOL 17(3), June 1989 .

[Feng 81] *"A Survey of Interconnection Networks"*, T. Feng, Computer, Dec 1981

[deFig 87] *"A Framework for Automation of 3D Machine Vision"*, R. deFigueiredo, TI Technical Journal, Winter 1987

[Flynn 66] *"Very High Speed Computer Systems"*, M. Flynn, Proceeding IEEE, VOL 54(12), Dec 1966

[Foley 82] *"Fundamentals of Interactive Computer Graphics"*, J. Foley, A. van Dam, Addison Wesley, 1982

[Foster 83] *"A Content Addressable Parallel Array Processor"*, C. Foster, C. Weems, S. Levitan, COINS Technical Report 83-32, September 1983

[Fountain 87] *"Processor Arrays: Architectures and Applications"*, T. Fountain, Academic Press 1987

[Fountain 88] *"The CLIP 7A Image Processor"*, T. Fountain, K. Mathews, M. Duff, IEEE Transactions on Pattern Analysis and Machine Intelligence, VOL 10(3), May 1988

[FPS 87] *"The FPS T-Series: A Parallel Vector Supercomputer"*, Floating Point Systems, November 1987

[Francis 90] *"Performance Evaluation of the Hierarchical Hough Transform on an Associative M-SIMD architecture"*, N. Francis, G.Nudd, T. Atherton, D. Kerbyson, R. Packwood, J. Vaudin, Proceeding ICPR, Atlantic City, June 1990

[Fuchs 85] *"Fast Spheres, Shadows, Textures, Transparancies, and Image Enhancements in Pixel Planes"*, H. Fuchs et al, Proceedings SIGRAPH, 1985

[Fuchs 89] *"Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor Enhanced Memories"*, H. Fuchs et al., ACM Computer Graphics, VOL 23(3), July 1989

[Gehani 88] *"Concurrent C++: Concurrent Progamming with Class(es)"*, N.H. Gehani, W.D. Roome, Software Practice & Experience, VOL 18(12),1157-1177, Dec 1988

[Gehringer 88] *"A Survey of Commercial Parallel Processors"*, E. Gehringer, J. Abullarade, M. Gulyn, Computer Architecture News, VOL 16(4), September 1988

[Gelernter 89] *"Linda in Context"*, N. Carriero, D. Gelernter, Communications of ACM, VOL 32(4), April 1989

[Glassner 85] *"Hardware Enhancements for Raster Graphics"*, A. Glassner, H. Fuchs, Fundamental Algorithms for Computer Graphics, Ed. R.A. Earnshaw, Springer-Verlag, 1985

[Goguen 89] *"The Rewrite Rule Machine"*, J. Goguen et al, Oxford University Computing Laboratory Technical Monograph PRG-76, August 1989

[Goldberg 83] *"Smalltalk 80: A Language and its Implementation"*, A, Goldberg, D. Robson, Addison Wesley 1983

[Gonzalez 77] *"Digital Image Processing"*, R. Gonzalez, P. Wintz, Addison Wesley, 1977

[Gottleib 83] *"The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer"*, A. Gottlieb, IEEE Transaction on Computers, Feb 1983

[Gourard 71] *"Computer Display of Curved Surfaces"*, H. Gourard, IEEE Transactions. C-20, June 1971,623-629

[Hamey 87] *"Low-level Vision on Warp and the Apply Programming Model"*, L. Hamey, J. Webb, I-Chen Wu, CMU-RI-TR-87-17, 1987

[HenPat 90] *"Computer Architecture a Quantitive Approach"*, J. Hennessey, D. Patterson, Morgan Kaufman 1990

[Hewitt 77] *"Viewing Control Structures as Patterns of Passing Messages"*, C. Hewitt, Artificial Intelligence, August 1977

[Hillis 85] *"The Connection Machine"*, D. Hillis, MIT Press, 1985.

[Hoare 78] *"Communicating Sequential Processes"*, C. Hoare, Communications of the ACM, VOL 21(8), August 1978

[Horaud 87a] *"New Methods for Matching 3-D Objects with Single Perspective Views"*, R. Horaud, IEEE Transactions on Pattern Analysis and Machine Intelligence, VOL 9(3), May 1987

[Hourard 87b] *"Model Based Strategy Planning for Recognizing Partially Occluded Parts"*, R. Horaud, T. Skordas, IEEE Computer, August 1987

[Hunt 89] *"AMT DAP - A processor array in a Workstation Environment"*, D. Hunt, Computer Science and Engineering, VOL4(2), April 1989

[Huttenloch 87] *"Object Recognition using Alignment"*, D. Huttenlocher, Proceedings of the First International Conference on Computer Vision, IEEE Computer Society Press, pp102-111, 1987

[Intel 87] *"The Intel iPSC/2 System Product-Information"*, Intel 1987

[Jesshope 89] *"Design of an SIMD Microprocessor Array"*, C. Jesshope, R. O'Gorman, J. Stewart, IEE Proceedings VOL 136(3), May 1989

[Jesshope 90] *"Communications Architectures for Fine-Grain Parallel Processing"*, C. Jesshope, Proceedings Paralla, Sept 1990

[Jul 88] *"Fine Grained Mobility in the Emerald System"*, E. Jul et al, ACM Transactions on Computer Systems, Vol 6(1), Feb 1988

[Kafura 89] *"Inheritance in Actor Based Concurrent Object-Oriented Languages"*, D. Kafura, K. Hae Lee, Proceeding ECOOP, 1989

[Kafura 90] *"ACT++: Building a Concurrent C++ with Actors"*, D. Kafura, K. Hae Lee, Journal of Object Oriented Programming, May 1990

[Kaiser 89] *"MELDing Multiple Granularities of Parallelism"*, G. Kaiser et al, Proceeding ECOOP, 1989

[Kerbyson 90] *"Study of the use of Count Hardware on the Warwick Pyramid Machine"*, Internal Research Report,1990

[Kuck 74] *"Measurements of Parallelism in Ordinary FORTRAN Programs"*, D. Kuck et al, Computer, VOL 7(1), Jan 1974

[Lea 86] *"VLSI and WSI Associative String Processors for Cost Effective Parallel Processing"*, R. Leas, The Computer Journal, VOL 29(6), 1986

[Lea 88] *"ASP: A Cost Effective Parallel Microcomputer"*, R.M. Lea, IEEE Micro, October 1988

[Matelan 85] *"The FLEX/32 Multicomputer"*, N. Matelan, Proceedings of the International Symposium on Computer Architecture, 1985

[Meyer 88] *"Object Oriented Software Construction"*, B. Meyer, Prentice Hall, 1988

[Mock 86] *"Processes Channels and Seamaphores (Version 2)"*, Logical Systems C Technical Documentation, 1986

[NCUBE 88] *"Overview of the NCUBE/10"*, NCUBE Corporation, 1988

[Nickel 87] "Digital IC Technology: 1987-2000",S.J.Nickel,Proc. VLSI & Computers, Hamburg 1987

[Nudd 88] *"A Heterogeneous Architecture for Parallel Image Processing"*, G.R. Nudd, R.M. Howarth, T.J. Atherton, N.D. Francis, G.J. Vaudin, D.W. Walton, UK Information Technology '88 conference, pp. 495-499, Swansea, July 1988

[Nudd 89] *"WPM : A Multiple-SIMD architecture for image processing"*, G.R. Nudd, T.J. Atherton, R.M. Howarth, S.C. Clippingdale, N.D. Francis, D.J. Kerbyson, R.A. Packwood, G.J. Vaudin, D.W. Walton, 3rd IEE conference on Image Processing and Its Applications, pp. 161-165, Warwick, July 1989

[Nudd 90a] *"Application of Image Processing to Land Vehicle State Vector Determination"*, G.R. Nudd, T.J. Atherton, S.C. Clippingdale, N.D. Francis, D.J. Kerbyson, G.J. Vaudin, IEE Colloquium on "The Role of Image Processing in Defence and Military Electronics", London 1990

[Nudd 90b] *"A Hierarchical Multiple-SIMD Architecture for Image Analysis"*, G.R. Nudd, T.J. Atherton, N.D. Francis, R.M. Howarth, D.J. Kerbyson, R.A. Packwood, G.J. Vaudin, ICPR, Atlantic City,(to be published in Journal of Machine Vision and Applications), June 1990

[Nudd 91a] *"A Hierarchical Multiple-SIMD Architecture for Image Analysis"*, G.R. Nudd, N.D. Francis, T.J. Atherton, D.J. Kerbyson, R.A. Packwood, G.J. Vaudin, ICPR,To be published in 'Machine Vision and applications', Springer Verlag 1991.

[Nudd 91b] *"A Massively Parallel Heterogeneous VLSI Architecture for MSIMD Processing"*, G.R.Nudd, D.J. Kerbyson, T.J. Atherton, N.D. Francis, R.A. Packwood, G.J. Vaudin, in 'Algorithms and Parallel VLSI Architectures', Elsevier North Holland, 1991.

[O'Gorman 89] *"Design and Application of the RPA II"*, R. O'Gorman, PhD Thesis University of Southampton, 1989

[Page 83] *"DisArray: A 16 x 16 RasterOp Processor"*, I. Page, Eurographics, 1983

[Page 87] *"The Disputer - A multi-paradigm Parallel Architecture"*, I. Page, Proceeding of BCS Parallel Processing for Display, London, May 1987

[Parkinson 90] *"Massively Parallel Computing with the DAP"*, D. Parkinson, J. Litt (Editors), MIT Press, 1990

[Perihelion 89] *"The Helios Operating System"*, Perihelion Software, Prentice Hall, 1989

[Perrot 79] *"A Language for Array and Vector Processors"*, R. Perrot, ACM Transactions on Programming Languages, Oct 1979

[Pfister 87] *"An Introduction to the IBM Research Parallel Processor Prototype (RP3)"*, G.F.Pfister et al., Experimental Computer Architecture, Ed. J. Dongarra, North-Holland.

[Phong 75] *"Illumination for Computer Generated Pictures"*, B.T.Phong, Communications of the ACM,VOL 18(6), June 1975.

[Porrill 86] *"TINA: The Sheffield AIVRU vision system"*, T. Porrill et al, AIRVU Lab Memo, University of Sheffield, 1986

[Potmesil 89] *"The Pixel Machine: A Parallel Image Computer"*, M. Potmesil, E. Hoffert, ACM Computer Graphics, VOL 23(3), July 1989

[Refenes 90] *"European Parallel Computing"*, A. Refenes, E. Odijk, Parallel Computers, Object Oriented, Functional, Logical, Ed. P. Treleaven, Wiley 1990

[Reynolds 81] *"IPC User Manual"*, D.Reynolds, G. Otto, Report No. 82/4, IP Group, University College London, November 1981

[Ruechardt 87] *"Microelectronics & VLSI: Status and Trends"*, H. Ruechardt, Proc. VLSI & Computers, Hamburg 1987

[Russell 78] *"The Cray-1 Computer System"*, R. Russell, Communications of the ACM, January 1978

[Sequent 86] *"Balance Technical Summary"*, Sequent Computer Systems, Nov 86

[Scott 90] *"Multi-Model Parallel Programming in Psyche"*, M. Scott et al, ACM Transactions on Computer Systems, 1990

[Shaw 84] *"SIMD and M-SIMD Varients of the NON-VON Supercomputer"*, D.Shaw, Proceedings of COMPCON'84, 1984

[Shepard 87] *"The Transputer and OCCAM"*, R. Shepard, Proceeding of BCS Parallel Processing for Display, London, May 1987

[Siegel 81] *"PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Rocognition"*, H. Siegel et al, IEEE Transactions on Computers, Dec 1981

[Smith 90] *"A C++ Environment for Distributed Application Execution"*, K. Smith, A. Chatterjee, MCC Technical Report no. ACT-ESP-275-90, Sept 1990

[Sproull 83] *"The 8x8 Display"*, R. Sproull, I. Sutherland, ACM Transactions on Graphics, Vol 2(1), 32-56, January 1983

[Stroustrup 86] *"The C++ Programming Language"*, B. Stroustrup, Addison Wesley 1986

[Theoharis 87] *"Algorithms for Parallel Polygon Rendering"*, T. Theoharis, Springer-Verlag, Ed. G.Goos & J. Hartmanis, 1987

[TMC 89] *"Connection Machine: Model CM-2 Technical Summary"*, Thinking Machines Corporation, 1989

[Uhr 81] *"A 2-Layered SIMD/MIMD Parallel Pyramidal 'Array-Net'"*, L. Uhr et al, IEEE PAMI, Feb 1981

[Vaudin 89a] *"A Heterogeneous Array for Computer Graphics"*, J. Vaudin, G. Nudd, N. Francis, T. Atherton, D. Kerbyson, R. Packwood, Computer Graphics '89, Workshop on Computer Graphics Technology. London, November 1989.

[Vaudin 89b] *"A Generalised Parallel Architecture for Image Based Algorithms"*, J. Vaudin, G.Nudd, T.Atherton, S.Clippingdale, N. Francis, R.Howarth, D. Kerbyson, R. Packwood, D.Walton, Eurographics 1989, Hamburg, September 1989

[Watt 89] *"Fundamentals of Three Dimensional Computer Graphics"*, A. Watt, Addison Wesley 1989

[Weems 88] *"Proceeding of the DARPA Image Understanding Benchmarking Workshop"*, Ed. C. Weems, UMass, Oct 1988

[Weems 89] *"The Image Understanding Architecture"*, C. Weems et al, International Journal of Computer Vision, (2), 1989

[Yantchev 89] *"Adaptive Low Latency, Deadlock Free Packet Routing for Networks of Processors"*, J. Yantchev, C. Jesshope, IEE Proceedings, VOL 136(3), May 1989

[Yokote 86a] *"The Design and Implementation of Concurrent Smalltalk"*, Y. Yokote, M. Tokoro, Proceedings OOPSLA, 1986

[Yokote 86b] *"Concurrent Programming in Concurrent Smalltalk"*, Y. Yokote, M. Tokoro, Object Oriented Concurrent Systems, MIT Press, 1986

[Yonezawa 86a] *"Object-Oriented Concurrent Programming in ABCL/1"*, A. Yonezawa, J. Briot, Proceeding OOPSLA, 1986

[Yonezawa 86b] *"Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1"*, A. Yonezawa et al, Object Oriented Concurrent Systems, MIT Press, 1986

Appendix A

Pyramid C++ Language Definition

A.1 Introduction

Pyramid C++ is a superset of the C++ programming language which incorporates a number of features that enable it to be used to express control and data parallelism. These extensions have been carefully designed so as to involve the minimum possible changes to the syntax of C++ itself. To this end many of the features of the programming system are provided by class libraries rather than extensions to the base language.

This description of Pyramid C++ is divided into two parts, the first of which deals with the extensions to C++ added to Pyramid C++, the second of which deals with the supporting class library. Taken together these constitute a definition for the Pyramid C++ programming system as a whole.

A.2 Language Extensions

Pyramid C++ is a superset of the C++ language defined in "*The C++ Programming Language*" [Stroustrup 86]. All valid C++ programs are also valid Pyramid C++ programs. In addition to those language constructs defined in C++ Pyramid C++ defines the following others.

A.2.1 Actors

An actor is an independent object with its own flow of control. Actors are instances of actor classes. An actor class is declared as follows

```
actor class <class_name>
```

The declaration is identical to that for a normal class, except for the inclusion of the keyword `actor`.

An actor (that is instances of an actor class) is an object which can execute its member functions autonomously and potentially in parallel with other processing. When a member function of an actor is called, the calling function is free to continue with its processing while the actor executes the called member function.

Actors and actor classes can be used in the same way as ordinary classes and objects except that:

- Actors may not access the member variable of another actor, even one of the same class
- Parameters to member functions of actors may only be passed by value, unless they are themselves actors in which case they must be passed by reference.

A.2.2 Message Passing Semantics

Actors consist of two components, a passive (normal C++) object and an object controller. When a member function of an actor is called a message is sent from the calling function to the actor's controller . The object controller is responsible for receiving the messages which request the execution of a member function and executing the appropriate member function in the passive object.

Messages are sent to actors using the C++ member function calling syntax

```
<object> . <member_function_name>(<parameters>)
```

or

```
<object_reference> -> <member_function_name>(<parameters>)
```

The message passing mechanism supports a number of different behaviours. Both the sender and receiver of the message can determine certain aspects of the behaviour required.

A.2.2.1 Message Sender

There are two principal alternative message passing behaviours for the message sender

Member Functions not Returning a Value - the sender always continues immediately after the message is sent.

Member Functions Returning a Value - the sender may either wait for the value to be returned (the default behaviour) or can continue execution and fetch the return value at a later stage using the deferral mechanism.

Deferred Value Returning

To defer the receipt of the returned value the caller must create an object of class `Pledge`, which will be used to access the return value at a late stage.

```
Pledge <pledge_object>;
```

This `Pledge` object can then be associated with the message transaction

```
<pledge_object>=<object> . <member_function>(<parameters>);
```

Subsequently the actual return value may be read into a variable using the << operator defined on the `Pledge` class

```
<return_value> << <pledge_object>
```

If at the time this statement is performed the return value has not been produced by the called member function, the statement will cause the caller to block and wait until the value is produced.

A.2.2.2 Message Receiver

The receiving actor can control its message passing behaviour in two ways. Each member function that returns a value can specify at what point in its execution the value is returned. In addition the object controller can determine which messages the actor will accept at any given time.

Non Terminating Value Returning

In order to maintain compatibility with C++ the conventional C++ `return` statement may be used. Alternatively a value can be returned by explicitly sending a message to the sender of the message being processed. In the latter case the object makes use of a special pseudo-object `sender`, which refers to the sender of the message which invoked the current member function. This is achieved as follows

```
sender.reply(<return_value>)
```

This statement does not terminate the member function's execution as a conventional return statement does, allowing the function to continue execution after a value has been returned.

Controlling Message Acceptance

Each member function of an actor may be enabled or disabled under control of the object controller. An enabled member function may be invoked by a message. A disabled member function cannot be invoked by a message. Any messages received requesting the execution of a disabled member function are queued. The default state is for all member functions of an actor to be enabled.

The object controller supports two operations, one to enable and one to disable a member function. These are performed by calling the `controller` pseudo-object defined for every actor. This object supports two operations `block` and `accept`. A member function can be disabled as follows

```
controller.block(<member_function>)
```

A member function can be enabled as follows

```
controller.accept(<member_function>)
```

When re-enabled any queued messages will be processed, in a non specified order.

A.3 Parallel Class Library

The data parallel programming support is provided by a parallel class library. This provides a number of classes which allow operations to be performed on collections of objects.

The class hierarchy for the parallel classes is shown below

```
Image
      BitImage
      ByteImage
      ShortImage
      IntImage
Patch
      BitPatch
```

```
          BytePatch
          ShortPatch
          IntPatch
     Cluster
```

Subclasses are shown indented under their base class.

A.3.1 Class Image

Class `Image` is the most important class in the library. It provides the interface to all the data parallel operations. The `Image` class allows any number of bits per pixel to be specified. Its subclasses `BitImage`, `ByteImage`, `ShortImage` and `IntImage` are provided simply for convenience, to allow common image depths to be easily specified. All data parallel functionality is provided by the Image base class. The base classes will therefore not be discussed further.

A.3.1.1 Public Definition

```
class Image{
public:
// housekeeping
        Image(int bits);
        ~Image();
        Image(Image&);
        Image& operator=(Image&);

// arithmetic operations
        Image& operator+(Image&);
        Image& operator-(Image&);
        Image& operator*(Image&);
        Image& operator/(Image&);
        Image& operator&(Image&);
        Image& operator|(Image&);
        Image& operator^(Image&);
        Image& operator&&(Image&);
        Image& operator||(Image&);
        Image& operator^^(Image&);
        Image& operator<(Image&);
        Image& operator>(Image&);
        Image& operator<=(Image&);
        Image& operator>=(Image&);
        Image& operator==(Image&);
        Image& operator+=(Image&);
        Image& operator-=(Image&);
        Image& operator*=(Image&);
        Image& operator/=(Image&);
        Image& operator&=(Image&);
        Image& operator|=(Image&);
        Image& operator^=(Image&);
        Image& operator&&=(Image&);
        Image& operator||=(Image&);
        Image& operator^^=(Image&);
        Image& operator<<(int);
        Image& operator>>(int);
// image shifting operations
        Image& N(Image&);
        Image& S(Image&);
        Image& E(Image&);
        Image& W(Image&);
}
```

A.3.1.2 Member Functions

Arithmetic Operations

All arithmetic and logical operations defined on images that have standard definitions on integers are interpreted as performing the standard operation to every pixel of an image. For example

```
Image& operator+(Image&);
```

adds each pixel of the target image with the corresponding pixel in the parameter image and returns an image containing the resultant pixels.

Image Shifting Operations

Four operations are provided to shift images within the image plane. These are:-

```
        Image& N(Image&);
        Image& S(Image&);
        Image& E(Image&);
        Image& W(Image&);
```

The operations are N, S, E and W, for north, south, east and west. North is interpreted as towards decreasing y-coordinate, and West is interpreted as towards decreasing x-coordinate.

The North operation replaces each pixel with the pixel to its north. The others perform the same operation for the appropriate direction.

A.3.2 Class Patch

The `Patch` class provides the same functionality as the `Image` class at the level of a single cluster. A `Patch` is a single 16x16 image which exists on a single cluster. Classes `BitPatch`, `BytePatch`, `ShortPatch` and `IntPatch` are provided for convenience, to allow common image depths to be specified.

A.3.2.1 Public Definition

```
class Patch{
public:
// housekeeping
        Patch(int bits);
        ~Patch();
        Patch(Patch&);
        Patch& operator=(Patch&);
// arithmetic operations
        Patch& operator+(Patch&);
        Patch& operator-(Patch&);
        Patch& operator*(Patch&);
        Patch& operator/(Patch&);
        Patch& operator&(Patch&);
        Patch& operator|(Patch&);
        Patch& operator^(Patch&);
        Patch& operator&&(Patch&);
        Patch& operator||(Patch&);
        Patch& operator^^(Patch&);
        Patch& operator<(Patch&);
        Patch& operator>(Patch&);
        Patch& operator<=(Patch&);
        Patch& operator>=(Patch&);
        Patch& operator==(Patch&);
        Patch& operator+=(Patch&);
        Patch& operator-=(Patch&);
        Patch& operator*=(Patch&);
        Patch& operator/=(Patch&);
        Patch& operator&=(Patch&);
        Patch& operator|=(Patch&);
        Patch& operator^=(Patch&);
        Patch& operator&&=(Patch&);
        Patch& operator||=(Patch&);
        Patch& operator^^=(Patch&);
        Patch& operator<<(int);
        Patch& operator>>(int);
// patch shifting operations
        Patch& N(Patch&);
        Patch& S(Patch&);
        Patch& E(Patch&);
        Patch& W(Patch&);
}
```

A.3.2.2 Member Functions

Arithmetic Operations

All arithmetic and logical operations defined on patches that have standard definitions defined on integers are interpreted as performing the standard operation to every pixel of an patch.

Patch Shifting Operations

Four operations are provided to shift patches within the image plane. These are:-

```
        Patch& N(Patch&);
        Patch& S(Patch&);
        Patch& E(Patch&);
        Patch& W(Patch&);
```

The operations are N, S, E and W, for north, south, east and west. North is interpreted as towards decreasing y-coordinate, and West is interpreted as towards decreasing x-coordinate.

The North operation replaces each pixel with the pixel to its north. The others perform the same operation for the appropriate direction.

A.3.3 Class Cluster

The `Cluster` class provides access to the Multi-SIMD capability of the Warwick Pyramid Machine. One instance of class `Cluster` exists for each physical cluster of the machine. The main functionality provided by the `Cluster` object is to invoke operations on `Patches`. Since the `Patch` class is not an actor class, `Patches` cannot support autonomous operations. The `Cluster` class is used to provide this autonomy, to simulate the behaviour of the real cluster.

A.3.3.1 Public Definition

```
actor class Cluster{
public:
        //constructor
        Cluster();
        //message forwarding
        Patch& call(MemberPtr member_function,Patch& p1,p2);
}
```

A.3.3.2 Member Functions

Message Forwarding

Message forwarding allows an operation to be performed on a `Patch` as if it were an actor object.

The `call` member function

```
        Patch& call(MemberPtr member_function,Patch& p1,p2);
```

takes a pointer to the member function to be called as a parameter. It also takes two patch references as parameters, the first being the patch on which the function is to be called, and the second being the parameter to the member function (which may be null if no parameter is required). It returns a reference to the result patch.

The `Cluster` class will not in general be used by the applications programmer, who will generally use only the `Image` and `Patch` classes.