Concurrent Viola Jones Classifiers on a Portable Beowulf Cluster

A thesis presented in partial
fulfilment of the requirements

for the degree

of Master of Engineering

in Mechatronics at
Massey University

Ravi Kiran Chemudugunta
2008

Abstract

Real-time Computer Vision is an interesting application for supercomputing, real-time applications (vision processing in particular) employ special purpose hardware such as DSPs to achieve high performance. This thesis explores parallel computers particularly commodity general purpose hardware. We also build a prototype to better understand the economics of supercomputing, specifically related to mobile computing - low power, rugged design by building a mobile computer. A new communication layer is built, where by the nature of the locality of the nodes allows one to optimise the protocols to reduce the latency comparably. Finally a study and in depth results of the algorithm, the Viola Jones Object detector in parallel are presented followed by reflection and future work based on the current results and platform.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Andre Barczak. Without his advise and motivation I wouldn't have been able to succeed. I thank him for listening to my sometimes strange ideas and steering me in the right direction throughout this work. I would also like to acknowledge my co-supervisor Dr. Johan Potgieter for input and advise on the project.

I extend my thanks to Dr. Napoleon Reyes for being encouraging and Dr. Chris Messom for his valuable insights and input. I would like to acknowledge everyone from the Computer Sciences Department, Guy Kloss, Rushad Irani, Anton Gerdelan and Daniel Playne for their encouragement and friendship through the years.

I would like to thank Thomas Felton for counselling me through my life, his support has been integral for me to be able to finish this work. Finally, I would like to thank my family, who supported me in everyway during this work, without them I would not be here today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The motivation for this work is to realise a concept of real-time video processing on a parallel computer. Image Processing promises intelligent systems that are able to communicate with humans better. However it is a very compute intensive process and requires specialised hardware to make implementation viable. Image Processing application exhibit inherent parallelism and so can be parallelised using multiple processors each doing some sub-task in a larger task thereby speeding up the operation. Using the analogy of a hole being dug by many workers - the more workers the faster it is completed, however one cannot have an infinite number of workers digging the same hole as in reality there are space restrictions around how infinite workers can be placed in a physical space and issues of how they collaborate with each other so they do not run into each other all the time. The same is true for parallel computing, the more computers there are on some task, the faster it can be performed, but like the workers it depends on how much interprocess communication exists between them, how often they have to synchronise and physical restrictions such as bandwidth and memory.

The objectives for this work are to build a system that demonstrates the use of multiple processors on an vision detection algorithm, and optimising where necessary algorithms and communication protocols to make the system as efficient as possible with the given resources. The vision is to have a reusable framework for medium sized mobile robotics such as humanoid size robot and autonomous vehicles to enable high speed image processing so to make them more aware of their surroundings (for e.g. localisation) and intelligent in they way they interact with humans (for e.g. gesture recognition).

The scope of the work includes using a particular form of vision detection algorithm, the Viola Jones method. Extensive research has been conducted on this topic and many resources are available from which to start from, these include work by Andre L. Barczak (supervisor of this thesis) for e.g. in [Barczak et al., 2005a] that provides a framework for hands detection (see chap. 2). Many types of architectures exist for computing, these include embedded microprocessors, desktop computing processors and special purpose hardware like FPGA and DSP processors. This work will focus on using commodity off the shelf hardware that is able to run linux out of the box and provide common interfaces such as USB for easy interfacing with easily available web camera hardware.

Chapter two reviews relevant literature and how it has shaped this work. The literature review is split into three categories, fundamental theory, software and hardware to de-lineate the different aspects of the project. First fundamental theory explains the workings of object

detection method used for this thesis and links some of the aspects of the algorithms to how it may function on parallel machines.

Past work is reviewed, including designs and proposals of architectures and systems for parallel image processing. This includes review of literature on parallel machines based on the Beowulf formula for achieving faster computing, their operation and construction.

A particular programming problem can be solved in many ways; either leveraging on already available application programming libraries or starting from scratch. The section on software infrastructure surveys available options such as Message Passing Interface messaging platform for communication.

The literature review concludes with a summary of all of the literature reviewed and what things were taken into consideration and how they effected this work.

Chapter three discusses the materials and methods used during for the development. The specifications of hardware chosen, why it was chosen and its general setup are outlined. The second part of the chapter details the software infrastructure used to develop the system. The distribution and its build system used to build the software and deploy the system is shown in detail. Testing tools allow one to improve the ability to check the functionality of some feature without having to actually deploy on the hardware. The various testing tools and how they were used in this work are discussed. Methods of benchmarking are shown and their results verified to show that all results are accurate are outlined towards the end of the chapter.

In chapter four a detailed description of the software, including the protocol and framework. At the lowest level the algorithms and packet structures are discussed. Synchronisation is an important aspect of any parallel programming, especially with communication protocols. Synchronisation over broadcast channels are explored, following by some conclusions on the approach. Finally the chapter concludes with a detailed description of the implementation of protocols designed around UDP, reusable framework and application programming interface that is exposed for the implementation of the system.

In chapter five results are presented. The chapter provides details of the setup of the system used for benchmarking the system. First isolated benchmarks of the protocols is presented conducted on a larger cluster to examine the effect of increased number of nodes. Following this, an examination of the hardware used for the embedded system is presented specifically related to rates at which it is able to compute algorithm related tasks.

Following isolated benchmarks, the results of the benchmarks on the completed system with all of the protocols and algorithms are shown with rigorous analysis and reflection of results.

Chapter six concludes the thesis by reviewing the results from chapter 5. A review of the work conducted and some perspectives on the various aspects are provided for anyone wishing to repeat a similar project. Finally the chapter concludes with future work and the various directions that can be taken from this point such as other object detection algorithms that can be performed on the system or modifications to the hardware to improve on the performance already gained.

Chapter 2

Literature Review

This thesis brings together many elements of research in the fields of machine vision, parallel computing, embedded software and hardware development. The literature review is divided up into sections to help put the various work reviewed into perspective.

2.1 Object Detection

Object Detection deals with identifying objects in images such as cars, people and animals. Ideal algorithms do not have have any constraints on the input images in order to work. For example it is quite easy to detect an object if a background frame with no object in it is available from which the new image can be subtracted to obtain the location and size of the object. This approach would not scale well if there were multiple types of objects in the scene and only a particular object was needed (without further processing).

Figure 2.1 shows an example of application, hands detection, for rotation in-variant gesture recognition.

There are fundamentally two ways of describing an object (according to [Yang et al., 2002] and [Bianchini et al., 2004], there are four methods; this thesis focuses on the feature invariant approach), using its statistical properties, such as an histogram or using morphological properties of the object such as geometric relationships of image intensities within the image and, component based, a combination of several of the above two mentioned. Generally, statistical descriptors allow for wide range of variation in objects while morphological operators allow for much less variability. One such method that is examined is the ViolaJones detector which uses a combination of statistical-like and morphological properties of an object to maximise its performance. They are similar to Joint Histograms which are a combination of histograms of colour channels at various areas in the image [Pass and Zabih, 1999]. The object descriptor will here by be referred to as the kernel.

Example of one such statistical detection system is detailed in [Kloss, 2008].

Further, objects presented can be of any size, orientation (angle) and position within the image. Ignoring for now the problem of orientation, the problem of variation in size and position of the object within the image can be solved by two ways. These are, moving and rescaling the kernel, and, keeping the kernel the same size and moving it across multiple

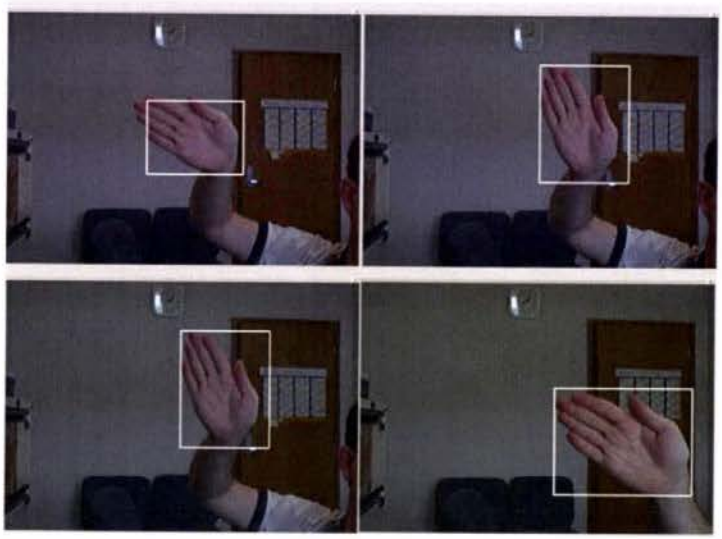

Figure 2.1: Samples of Hand Detection, source [Barczak and Chemudugunta, 2006]

versions of the image each at a different scale. The latter is sometimes referred to as multi-resolution analysis and is considered computationally expensive because scaling images takes a large percentage of time in comparison to the detection phase.

The Viola Jones detector [Viola and Jones, 2002] uses the moving/scaling kernel approach, where by the kernel is dragged across the image at different scale factors. The detector is able to run very rapidly, boasting 15 frames per second on a 700 MHz Desktop Computer @ a resolution of 320 x 224. The parameters that control a kernel's scale and position are referred to as scale factor and translation factor. The scale factor is the proportion of subsequent sizes of the scaled kernel, while the translation factor is the number of pixels the kernel is moved at each iteration. The kernel can be rescaled easily because a kernel is made up of rectangles (Haar Features) that describe some image area, and can themselves be described in relation to a proportion to the kernel size.

Figure 2.3 shows a simplified flow diagram of how the detector works. The ViolaJones works on pre-processed images which are called Integral Images [Crow, 1984]. An Integral Image is a special data structure designed to rapidly improve the calculation of the sums and differences of pixel intensities in images (Figure 2.8). A subwindow generator 2.2 is simply encapsulating the functionality of two nested loops, the output which is a series of subwindow 'addresses' containing position and scale information to dictate where the kernel must be applied in the image.

As a functional unit the detector can be seen as a binary function, if the given subwindow in the list matches the object specified by the Classifier then it outputs either true or false for that specific subwindow being examined.

Figure 2.4(a) shows some sample how a list of subwindows would map to the image, all the subwindows that can be generated haven't been shown for clarity purposes. Of the two marked subwindows SW0 and SW1 only SW1 would pass the detector test when a face classifier is used. Figure 2.4(a) shows a very small sample of subwindows, usually because we don't know where the image is all subwindows are searched to find the object, because

subwindows close to each other are very much like each other the detector returns true, the results of what this would look like is shown in figure 2.4(b).

The final stage of the detection pipeline (figure 2.3) is used to coalesce these neighbouring windows together to produce once result, the result of this operation is shown in figure 2.4(c) where multiple detections (figure 2.4(b)) are coalesced into one result around the object.

The OpenCV generator has a special form of the function 2.2 in which the translation is proportional to the scale.

An HCC is really a cascade of classifiers, we will discuss first how a monolithic classifier is constructed.

To understand the performance characteristics of the classifiers of the Viola Jones detector we must first understand how it is constructed. A single classifier is constructed by using a set of samples, split into training set (positive and negative) and test sets. The single classifier is trained such that it all samples in the positive set are positively marked while all samples in the negative set are attenuated. The SC is then run against the test set to obtain the performance characteristics such as detection rate D and false positive rate F. The detection rate D is the number of samples identified correctly while the false positive rate F is the number of samples identified incorrectly.

In practise however it is very hard to 'catch' all of the positive samples and 'attenuate' all of the negative samples in the training set because of the limitations of training method. For example one of the negative sample might look so much like the face that while attenuating it, we may loose the ability to classify some faces correctly. Generally this problem occurs because we cannot correctly split the N dimensional space into object/non-object areas.

2.1.1 Training

For a given input image a set of values can be obtained based on haar features. One can think of a haar feature as a very simple statistical descriptor of some image area, like an histogram. A feature value describes the difference of pixels described by the feature type (figure 2.7) and the image area underneath it. Like histograms, a image area with multiple configurations (different light intensities) can yield the same feature value.

Lets assume that the image size is 24x24 (a very small image) then the number of unique areas this image contains can be described by the number of pixels within the image, 24 * 24 = 576. However using Hf's the same image can be described by 45396 features [Viola and Jones, 2002], which is why Hf's are referred to as being an over-complete representation. The following process of computing the number of Hf's in an image area should not be confused with the scaling kernel approach discussed in the previous section.

Only a small number of important features are necessary to build a good classifier. Having 45,396 (for example) features is like having a space with that many dimensions. AdaBoost is used to select the most relevant features (dimensionality reduction) necessary to correctly classify the object. The selected features are called weak classifiers as by themselves they cannot correctly classify a subwindow (previous paragraph; discussion on statistical properties of Hf's) and so are combined to create a stronger classifier.

The literature points to the final monolithic face detector consisting of 200 weak classifiers to achieve good detection and false positive rates.

Figure 2.2: Generator Function

```
const int translation = 5;


const int M = kernel_size.width;
const int N = kernel_size.height;

const int W = img_size.width;
const int H = img_size.height;

double current_scale = 1;
int x = 0;
int y = 0;

int subwindows = 0;

while (
((M * current_scale) < W) &&
((N * current_scale) < H)
) {
x = 0;
while ((M * current_scale + x) < W) {
y = 0;
while ((N * current_scale + y) < H) {
subwindows++;
y+=translation;
}
x+=translation;
}
current_scale *= scale;
}
```

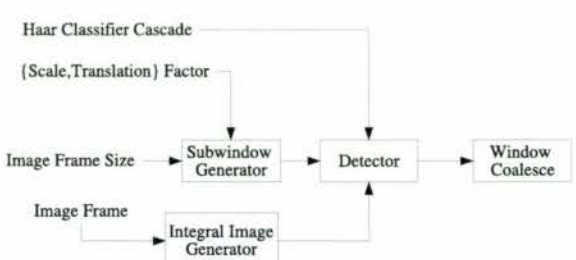

Figure 2.3: Flow Diagram

(a) Lena Sub Windows (b) Lena Many Hits (c) Lena One Hit

Figure 2.4: Lena

Figure 2.5: Composition of a Strong Haar Classifier
h: weak haar classifier
K: number of weak classifiers
w: weights associated with each weak classifier

$$strong\ classifier\ output = \sum_{k=0}^{K} h(k) \times w(k) \tag{2.1}$$

2.1.2 A Cascade of Detectors

[Viola and Jones, 2002] propose the use of multiple classifiers organised in a cascade. For example rather than having a monolithic classifier with 200 weak classifiers, having 10 - 20 weak classifier classifiers arranged in an cascade. Figure 2.9 shows the structure of the detector, the sub window passes through each of the stages and is only marked as the object if it has passed all of the stages. The advantage of using this architecture is that if trained correctly the cascade can be set up in such as way that all of the 'easy' samples can be rejected early and be spared of computing the rest of the classifiers and their weak classifiers in the cascade. Because the process is cumulative, if each of the individual stages is able to classify correctly to 99 % of the subwindows and comparatively large false positive rate, the cascading of these produces a very good classifiers.

The greater the number of weak learners the better the classifier, by limiting the number of weak learners in the early stages the number of subwindows which are not the object can be thrown out early (since each of the detectors have a good detection rate), the false positives or positives are forwarded to the next stage where yet more false positives are eliminated until all of the windows are eliminated or in the case it is the object then it passes the entire detector and is marked as an object.

This has some implications in the context of executing detection algorithm in parallel; because each of the classifiers takes a different amount of time depending on the input image (and the particular subwindow in question), one of the processes may finish early and be idle while the other processors are still working on their stages.

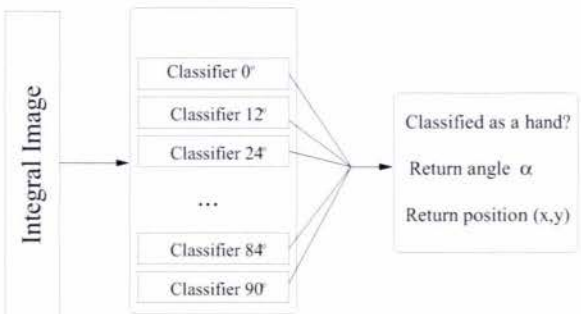

Figure 2.6: Parallel Classifiers, source [Barczak and Chemudugunta, 2006]

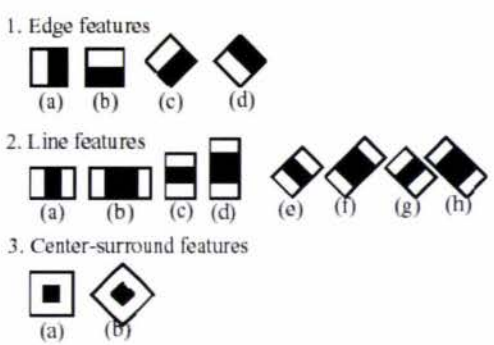

Figure 2.7: Haar Features, source [Intel, 2007]

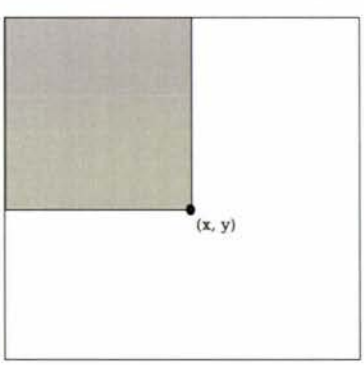

Figure 2.8: Summed Area Table, adapted from [Viola and Jones, 2002]

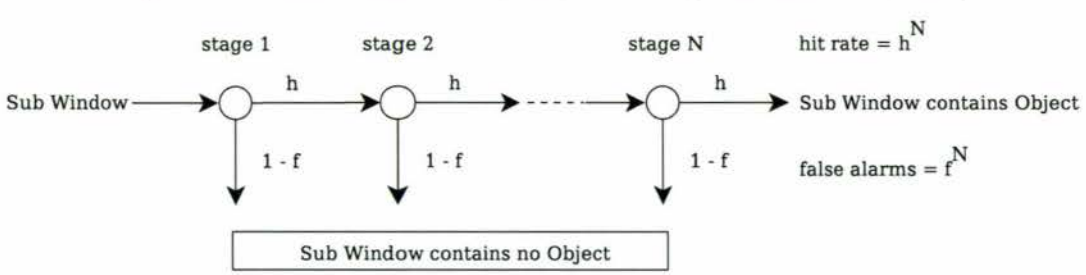

Figure 2.9: Cascade of Boosted Classifiers, adapted from [Lienhart and Maydt, 2002]

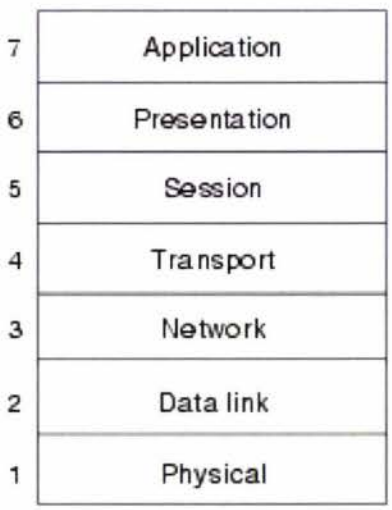

Figure 2.10: The OSI Model, source [CiscoSystems, 2006]

2.2 Parallel Computing on Clusters

Clusters are a set of discrete computing units, each with their own memories and processors, and usually incorporate some form of interconnect between the them. They are a result of an AdHoc movement to obtain more performance and supported by the fall in prices of high performance commodity computing componan ts such as computers and network hardware. Clusters of computers have been traditionally bound by I/O bottle neck that is the network but as network interconnect speeds increase this is no longer a limiting factor and can be seen in the decline of the use of large and expensive shared memory systems.

There are new ways of assessing performance than just peak flops and cost / flop, but more from an energy perspective that is watts/flop. It's not a purely environmental concern as the bigger the watts/flop the bigger the installation facility will need to be and the larger the cooling will need to be, increasing the cost / flop.

Lunchbox clusters are a new class of supercomputers which are smaller in size and have less power and infrastructure requirements then do their larger counterparts, i.e. they demonstrate good metrics such as flop / watt and price / giga-flop in comparison.

Mini-Clusters were first created by Mitch Williams of Sandia Laboratory in 2000. They are usually made by stacking together a set of embedded computer / single board computers connected via Ethernet and run the Linux kernel with customised operating system. The smaller size enables them to sit along side a desktop computer for testing of parallel applications before they are deployed on bigger clusters.

2.3 Motivation and Base Work

In [Barczak and Dadgostar, 2005] a framework for using multiple highly specialised classifiers to detect hand gestures at multiple orientations is presented. This approach differs in that while many solutions try to generalise the solution to include multiple orientations without

the added cost of image transforms, this solution allows classifiers to still be highly accurate and robust.

The work in [Barczak et al., 2005b] is used as basis for implementation forms the basis of this thesis. The paper presents a prototype of a parallel mobile platform that is designed to run concurrent object classifiers. Concurrent classifiers allow for robust detection of objects that are otherwise hard to train for using a single classifier. The design of the platform is based on a Beowulf (a class of parallel computers) detailed in [Barczak et al., 2003].

In addition to the characteristics of a typical Beowulf clusters, the system must also have qualities of an embedded system, relative low power consumption and size so that the system can be portable. The system also requires the ability to capture video without any external support, via and inbuilt USB interface/camera.

The work in [Barczak et al., 2005b] also presents an analysis of the performance of multiple classifiers on both single and multiple processor machines. Two models are presented, how good a particular classifier is at detecting objects with relation to processing requirements and positive detection rate, and what happens when two or more classifiers are pipelined to use the same pre processed image data (integral images). The model on multiple classifiers showed that there is a significant benefit to cascading multiple classifiers and that if parallelised with each of the nodes running more than one classifier then, the speed up is very close to linear.

An empirical model of a common communication library MPI, is presented showing that they are restrictive to the application of object detection by their performance. Specifically the broadcast call used to distribute images MPI_Bcast is not a true broadcast protocol but uses a tree like structure to distribute data which is not the most efficient strategy [Barczak et al., 2005b].

2.4 Hardware Infrastructure

The work in [Webb, 1993] presents a real-time parallel image processing based on a comparatively esoteric architecture running an stereo vision algorithm. According to [Webb, 1993, real time processing problems are different to scientific computing in the context of parallel computing. An increase in the number of processors increases the latency since, message sizes will be smaller and there will be more messages; but this can usually be compensated by increasing the data set size. However, because the problem sizes in most vision processing systems are fixed, for example there is no advantage in processing at a higher resolution].

Latency is an important factor, because the problem sizes are finite (the number of subwindows to be examined by one or many classifiers), it must be minimised to increase the time that can be spent computing the tasks before the timeslice is up.

2.5 Software Infrastructure

A thorough description of networking protocols can be found in [Tanenbaum, 1996]. Briefly networking protocols are stacked on top of each other so each layer is more or less unaware of the other layers present 2.10. This allows applications to concentrate on implementing against just one layer and not worry about the lower level layers or conversely, the lower

level implementations just to provide basic interfaces and not worry about what application is being run.

The work published in [Liu et al., 2002] on a UDP protocol for Internet Robots presents an overview around the rational for a new protocol for remote teleoperation of robots. The paper shows that TCP is not a good candidate for real-time communication and feedback of data because of its characteristics such as exponential back off. However, when designing UDP protocols they must not be overly aggressive, as this would threaten the stability of the internet. The ability co-exist with TCP traffic is called TCP-friendliness [Liu et al., 2002]. Because in this work there is no sharing of network it is not so important to be friendly. An interesting argument is also presented in [Liu et al., 2002], that is of relation to real-time video/audio systems to real-time robot teleoperation, that even though they are named real-time, only real-time teleoperation is truly real time where as video/audio transmission is buffered and so only quasi real time. Indeed, the whole image is buffered before any computation begins, being able to overlap the smallest possible completion unit in communication e.g. a subwindow with computation would be ideal (however, they overlap at different scales).

In [Gu and Grossman, 2005] a very good discussion of UDP based protocols is presented. Some of the findings that are presented is that acknowledgement in UDP protocol is very expensive if not handled correctly, because there is a context switch between user and kernel space, the resulting performance is lower than an in kernel implementation. A framework for what things a UDP protocol must take into account is also presented. This finding is re-iterated in [Majumder, 2004] on high performance libraries for MPI in UDP, that after adding reliability it results in performance worse than that of TCP. Therefore UDP if used, must be kept light, with minimal acknowledgements between and sender and receiver (i.e. keep the protocol overhead small).

Reliability is not orthogonal to performance [Donaldson et al.,]. Here Donaldson stresses that reliability must also be taken into account. For this application however, reliability is not so important (at least for the transmission video) if a packet is lost the next packet can be used to decode video.

The work published in [Tinetti and Barbieri, 2003] shows a current implementation of efficient broadcasting over Ethernet clusters in the MPI Library using the application While the existence of this paper was not known during the development of a UDP protocol, this paper provides some perspectives why a more efficient implementation does not yet exist in many common MPI libraries even though the concept of multicasting over Ethernet networks is well known; That is because of portability.

2.5.1 Interconnect Software Protocols

In [Geusebroek and Seinstra, 2005] grid-like computer networks are used to perform object recognition. A robotic dog uploads an image to a middleware component that dispatches the job of performing some kind of a dictionary match and then returns the result to the dog which then outputs the class of the object using an speech synthesiser. The concept is novel and opens up the opportunity to be able to harvest a lot of computing power via computers located on the internet. There is many such grid gateways now in operation, including one at Massey University, known as BestGrid.

Web Services are a form of language independent remote procedure calls, used widely to tie large / disparate systems together. In comparison to many RPC frameworks, web services are considered slow and heavy because of the verbosity of the underlying data exchange format, XML. However the work in [van Engelen, 2003] points to the feasibility of using WS as a communication layer. gSoap is a WS stack written in C/C++ and has many extensions that make it attractive for real-time applications, for example the ability to pack binary data instead of having to serialise it in XML Format.

gSoap also contains an implementation for UDP-over-SOAP [Combs et al., 2004], [van Engelen, 2003]. UDP-over-SOAP is an extension to the SOAP specification that allows SOAP messages to transported using UDP datagrams. This was done out of recognition that many applications match the semantics of UDP for example not needing delivery guarantees and the ability to make multicast transmissions. Because the characteristics required for this platform are similar, the use of SOAP would be a good approach also.

As mentioned in [Tinetti and Barbieri, 2003] Parallel Image Processing has been an area that has been studied for a long time, but no portable solution has emerged. This is largely due to the lack of support by languages for parallel programming. Further the difficulty to generalise in the presence of many types of architectures results in fragmented efforts.

In [Ramachandran et al., 2003] a cluster programming middleware for Interactive Stream-Oriented Applications, Stampede, is presented. Stampede attempts to hide the implementation details of multimedia applications by providing a specialised data structures called a space-time data-type. It also provides some abstractions relating to channels which have similar semantics as those presented in [Huang, 2006], another framework for Parallel Programming.

2.6 Conclusions of the Literature Review

The Viola Jones classifiers are used for the task of object detection. This class of object detection algorithms solves many problems of variation in an image, for example of location and size. This is achieved by using a moving-scaling kernel approach, this can be done rapidly because all computations are performed on a precomputed integral image.

Training classifiers is an expensive process, requiring weeks of compute time. Once trained however they are very accurate. This works deals only with the run time characteristics of classifiers, as the accuracy and detection capabilities have already been reported [Viola and Jones, 2002].

The image on which the detector runs has an effect on the performance. This is advantageous because image areas not representing the target object are eliminated early. However, in this context of multiple concurrent classifiers leads to load imbalance among processes.

The same paradigm that was used to build cheap Parallel Clusters can be used to also build cheap, fast and small machines. Lunchbox clusters are a new class of machines that have a similar design physiology to their larger counter-parts except that they demand much less space and power, and so allows them to be portable.

2.6.1 Implementation Details

MPI presents latency by synchronisation included in MPI broadcast operation, but it will still be considered because it integrates such an approach into a well known library. The paper [Tinetti and Barbieri, 2003] does not describe well how synchronisation was implemented, even with a broadcast medium no known algorithms for global synchronisation exist and can only be achieved using point to point algorithms (or complexity less than or equalling that of O (N), where N is the number of nodes synchronising at one time) [Schneider, 1982].

Grids and Web Computing currently for the application of real-time recognition this approach is not suitable mainly due the issue of latency and secondly the current hardware would not be able to harvest the processing power of an external networking computing device since it is not able to connect wirelessly.

The network will be a dedicated network with no other extraneous nodes on it to interfere with the communication. Hence, contention is not a major factor; also there are two independent physical channels available through which communication can take place. Also reliability is not such a major factor especially for broadcasting video, further more there are no known ways to provide reliability between multiple hosts but only on point to point networks.

Simple UDP will be used in favour of other encapsulation and transmission systems (for e.g. SOAP messages), simply because SOAP messages and such are too verbose and add overhead. Also UDP is a convenient way to broadcast data to many hosts at once. So there is a possibility to employ it for broadcast operations, which has been identified to be a bottle neck in the system.

2.6.2 On UDP

Packets on TCP/IP networks are fragmented; the size of fragmentation is a function of the amount of data on the output queue as well as protocol window settings. The maximum size of a packet (UDP or TCP) is limited to the payload size of its container, the IP packet. Because IP packets only have a 16 bit value to say how big its payload is the maximum size is limited to approximately 64K.

To circumvent this limitation a new standard was proposed in the form of RFC2675 [Borman et al., 1999] which allowed a special flag to be set within the header that would direct the stack to read the length of the tag in another header placed inside the payload. Current Literature does not show any real implementations of the standard, so appears that it is not a well documented feature. This is probably due to the fact that smaller packets are just better suited for the internet and it does not make sense to break the sharing capability created by having packet switching in the first place.

There is some ambiguity Related to Jumbograms, jumbograms in popular literature are related to changing the MTU of the Ethernet layer, typically raising it from the usual 1500 to 9000 bytes to improve CPU utilisation and efficiency [Gerdelan et al., 2007]. It is recommended that jumbograms be used to refer to IPV6 packet enlargement where as jumboframes are used for low level frame enlargement.

Chapter 3

Method / Material

In this chapter we examine the process of selecting materials and what is selected. This includes both hardware and software.

3.1 Hardware

Table 3.1: Cluster Specifications

CPU	VIA C3 / Eden (x86 compatible
Memory	DDR266 512 MB
Network	Dual LAN, VIA VT6105 LOM 10/100 Base-T Ethernet
USB	4 USB 2.0 Connectors
Features	On-board (Sound, Video, IDE) Controllers On-board (I2C, LVDS, Serial, Parallel) Connectivity
Power Supply	DC 12V to ATX Converter
Switch	2x Linksys 5 port Ethernet

Initial research has indicated there are many platforms that could be used, among the options are embedded ARM/MIPS based computer boards, single board computers (SBC) to desktop based computer parts.

One of the aims of the project, to reduce the time and money spent on esoteric hardware that it must be off the shelf, and able to run the Linux operating system. For our system we choose a mini-ITX form factor x86 based computer board by VIA. The specs are shown in table 5.1. The system is able to boot a i386 Linux Kernel without any modifications. The target is to have a customised distribution and special communication libraries, with support for applications built with other communication libraries like MPICH.

3.2 Physical Construction

Originally the nodes were built with threaded rods and fastening nuts, however this was not a very good solution as the nuts would frequently become loose and cause the boards to move

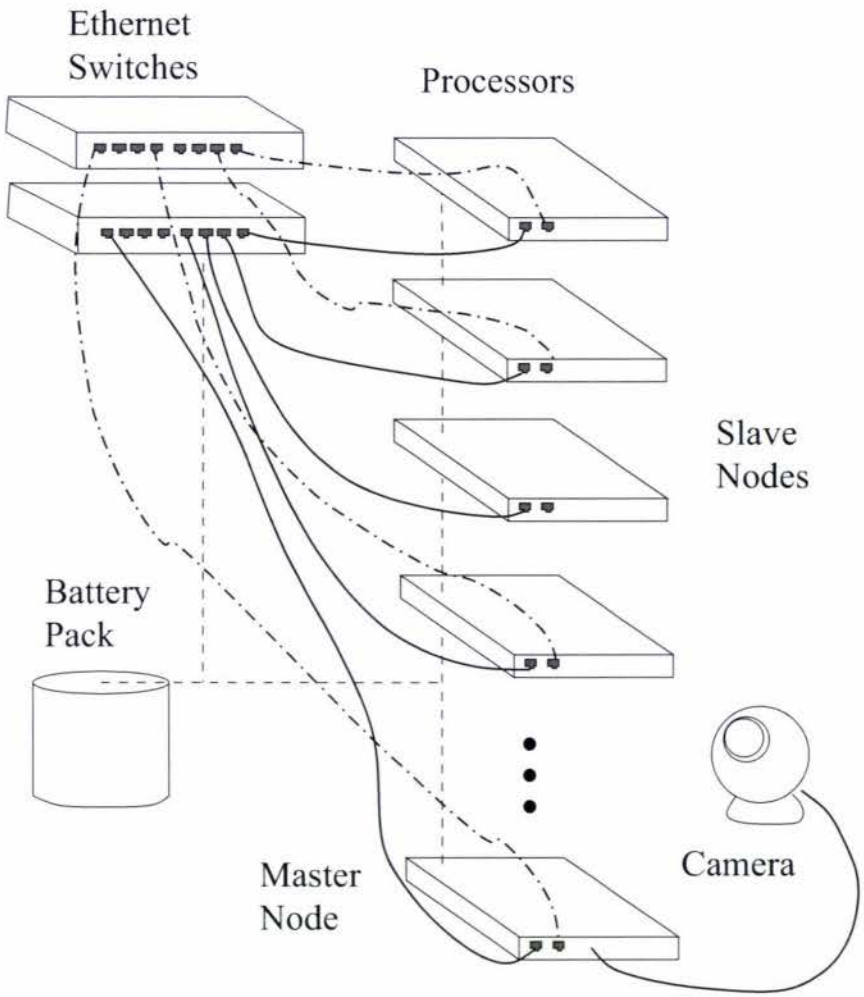

Figure 3.1: Hardware platform, source [Barczak and Chemudugunta, 2006]

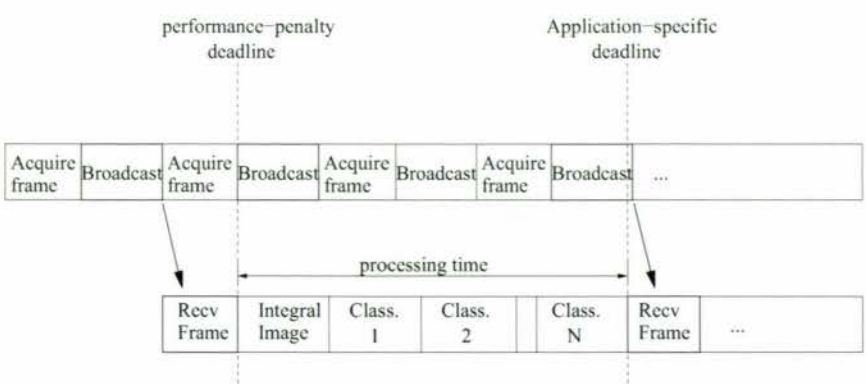

Figure 3.2: Real-time Constraints, source [Barczak and Chemudugunta, 2006]

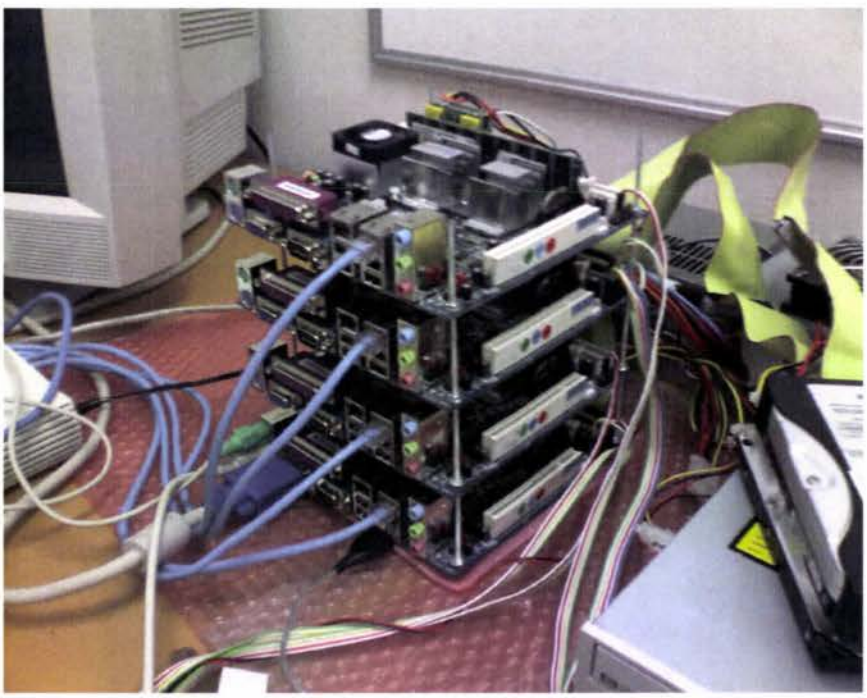

Figure 3.3: Prototype

from side to side during movement. Later the motherboards were placed in mini pizza boxes, this worked will but still the whole system was not a unit in one.

Finally some nylon spacers were found that did the job perfectly, they are stackable and hold the motherboards in place tightly without stressing the board materials itself. Figure 3.3 shows the final prototype with threaded rods and hard disks and CD-ROM drive during testing. The picture shows the relative arrangement of each of the boards.

3.3 Parallel Knoppix

ParallelKnoppix [Creel, 2007] was used to test the cluster. Like other Knoppix based distributions ParallelKnoppix is a Debian based Live-CD based distribution. A selected master node requires a CD-ROM drive while the slave nodes only require a PXE boot capable NIC. The master node first loads up and an initial setup process is required to select a few options such as which interface to start boot servers on and then the slave nodes are booted via network. PK also has the nice feature that it LAM boots the nodes to create a working environment for MPI programs to run.

A ParallelKnoppix distribution can be customised allowing for drivers and additional packages to be added in a process called remastering. The default version did not include camera drivers as well as libraries such as OpenCV and its dependencies.

3.4 Testing Tools

3.4.1 Qemu

Qemu [Bellard, 2007] is a virtual machine like VMware [Vmware, 2007] and Xen [Xen, 2007] with an easy to use command line interface, the ability to create images in the fly. This makes it ideal as a tool for testing distributions and has proved as an invaluable tool. Virtual Machine testing allows to quickly test something without having to move the image onto a the machine.

3.4.2 VDE Ethernet

Virtual distributed emulates a virtual networking setup and has virtual counterparts to cables and switches. Together with Qemu, VDE can be used to test a network of virtual machines without having to put it on the real machine. This is an excellent tool to test various boot up strategies such as network boot.

3.5 Distribution

Because of the distributed nature of development of the Linux platform a large number of options exist in the quest to decide which option to choose. Most distributions are written with an application in mind, and a target architecture to use. In general the easier it is to install and setup the lesser the flexibility. So, many developers are faced with the task of deciding whether to build an distribution from scratch or customise a pre-existing distribution.

Some of the popular distributions like Redhat and Debian come with a large Repository of software but minimum installs are very large in comparison to a custom built distribution. On the other hand, small distributions come with very limited choices for pre compiled software and no development tool chains installed, and applications compiled on a development box will have used different version of compilers and libraries.

The easy way and least extensible way is to take a distribution off the shelf, strip it to optimise for size. This is very hard to do and can leave the system architecture in a disabled such that no more software can be installed or added. This approach is not very scalable as with more complicated software packages like X11, where there are potentially many dependencies and it is hard to know whether removing something will affect the working of it so this can be a tedious process.

There are two classes of distribution, recipe based and packaged based. Recipes are scripts that dictate how a distribution should be built from scratch, where as packaged based distribution are pre packaged components. Instead of changing a distribution made out of packages it is better to edit the scripts that were used to make the packages because this is more scalable approach and allows the distribution to be rebuilt easily.

Most recipe based systems are also packaged based in that they generate packages which are then extracted to the target system; this is done to cache some of the work by enabling pre-compiled binaries to be used rather than having to generate everything from scratch (which is a very lengthy process).

Gentoo Linux focuses on performance by compiling all programs from scratch. Gentoo Linux is a collection of e-builds which are a collection of bash scripts which tell the installer how to compile and install programs.

The rocks build system is a general purpose build system which allows the customisation of all of the components, like Gentoo rocks is a set of recipes which tell the installer how to compile and install programs.

Debian Live is a distribution builder as well as a live helper like the linux-live project used to create and setup bootloaders for a variety of devices. Debian Live also has some built in knowledge on how to remove extra packages without effecting the performance of the system, further debian live has the added advantage of being able to provide netboot facility without manual modification of boot scripts.

3.5.1 TTY Linux build system

While most embedded distributions tend to be small and feature packed, it is hard to customise them. Usually to save space much of the documentation is removed, things are changed to decrease free space but at the expense of obfuscating the distribution. They don't contain a compiler to save space so a new package cannot be compiled which is needed in ensure that it works correctly. It is possible to compile a package under a different distribution (desktop system) and transfer this onto the root system of the distributed build system, however this has the problems of the fact that the versions of libraries such as libc may vary.

The TTY Linux build system is an embedded Linux system. It include the original build script used to build the distribution, a recipe. TTYLinux is therefore a package-based as well as recipe based distribution. Recipes are used to build packages which are then installed onto the root file system.

Using recipe based distributions has the advantage that the target distribution aligns closely with the development computer, CPU platform, libraries and utilities used. This means that the distribution can be made highly customisable in a scalable way as well as allowing easier transfer of programs from the development system to the target system.

3.5.2 Methods to Reduce Packages Sizes

Sometimes just having one dependant package can drag in hundreds of packages by recursive dependency filling. This can be avoided by choosing carefully at compile time what those dependencies are, in certain cases some dependencies maybe be eliminated or added to 'recommends' rather than to 'requires'. 'Recommends' and 'Requires' in the debian package format refer to what other packages are to be installed together, anything in the 'Requires' section is a hard dependency without who presense the application package will not function, where as 'Recommends' includes common packages that 'go along' with that package but do not necessarily effect the functioning of the package. Package boundaries are logically separated on packages, however sometimes this is not enough they need to be further split down, for example into documentation, libraries and executables.

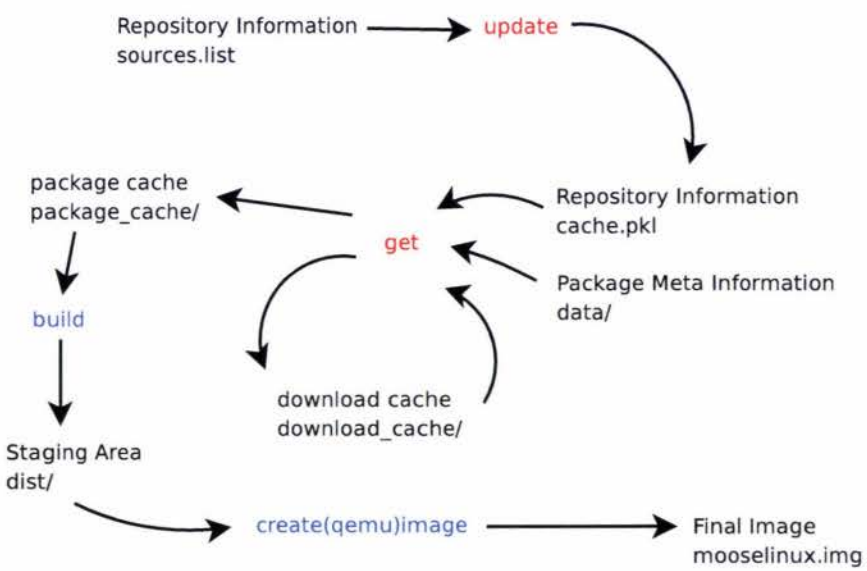

Figure 3.4: Moose Build System

3.6 Moose Linux

Using the knowledge of some of these distributions I decided to build my own distribution that targeted and focused to the application and development environment that is being using, Debian/Ubuntu, OpenCV and MPI.

3.6.1 Build System Overview

The build system is a combination of (Simple) Python and Bash together. The parts highlighted in red are python scripts, in blue are bash scripts and the rest of the graph shows resources (directories/files).

3.6.2 Distribution

Moose Linux uses Debian Repositories (either Ubuntu or Pure-Debian) as a source for packages. The first stage of the process involves getting a list of packages available from the server listed in sources.list. A small utility, 'update' performs this task, indexing package name and downloading location and outputting this as a pickle so it can easily be searched by 'get'.

'get' takes a single argument, the name of the package (also the key value for the pickle stored in the previous step). It uses the argument to look up where the package resides and proceeds to download it, this can either be local or any URL supported by the curl library. A download cache is utilised to reuse downloads on subsequent builds. When the package is successfully downloaded to download_cache/ it is extracted using dpkg-deb -x which retrieves all of the files in the package.

After all of the contents of a particular package are extracted, global rules such as stripping all documentation (/usr/share/doc, trees) are applied as well as any package specific rules

listed in Package Meta Information, data/. Here, one can place additional rules such as selecting exactly which files to choose (white list). The selected contents are then packaged up in a simpler archive, tar.gz and placed inside package_cache. 'update' and 'get' can be run independently.

3.6.3 Packaging

'build', a bash script does the work of taking the cached packages (stripped of documentation and other files) and putting them into an staging area, dist/. The advantage of leaving the staging area open to the user is so one can chroot into the environment and make desired changes before creating an distribution image using createimage.

'createimage' creates an in-place disk image formatted as an EXT2 disk (using the loop device) of the dist/ directory. One can chroot into the staging area make quick changes and create an image without redoing the entire process (although this it is better to put permanent changes into data/ as additional files / instructions for better clarity).

3.6.4 Testing

'createqemuimage' creates fully self contained bootable image for use with qemu so the image generated can easily be tested without the need to put it on a real computer. This is the most useful of features as trail and error testing can become very tedious when the software has to be transferred to another storage medium such as a flash disk.

3.7 Measuring Time

CPU Time is the wall time Process time is split into two fields,

process time: how long the process was executing
system time: how long the system is active within the given time slice.

These times are also commonly known as the real, user and system times.

```
#include <time.h> // Time API

clock(): get real time

times(): get real time using utime and system time using stime

utime: the time the process has been executing
stime: the time the operating system spent executing on behalf of the process

both utime and stime are of type struct tms
```

The code used for benchmarking through out was validated using 'time': a Unix utility that measures time of a process from start to finish reporting also the real, user and system times.

Chapter 4

Implementation

4.1 Synchronisation by Broadcast Messages

Synchronisation is the process where by all some set of processes all reach the same point in their execution and will not continue if all of the nodes in the group have left the stage. In the MPI framework this is known as an barrier call.

Ethernet is a very common interconnect among the HPC community, several studies have shown [Majumder and Rixner, 2004] fast ethernet interconnects for e.g. 10 GB/e are comparable to the more costly interconnects such as Myrinet. It is possible to develop optimised communication routines and parallel algorithms taking into account the underlying hardware available to computers [Tinetti and Barbieri, 2003], and many parallel algorithms are based on broadcasts [Tinetti and Barbieri, 2003].

A synchronisation algorithm must have the ability to differentiate between subsequent barrier calls, because one node may have passed to the next barrier while another is still leaving, given that the process skew is large enough [Schneider, 1982]. Synchronisation algorithms can take advantage of underlying architectures such as broadcast and multicast available on Ethernet, InfiniBand and Myrinet [Buntinas et al.,] [Schneider, 1982].

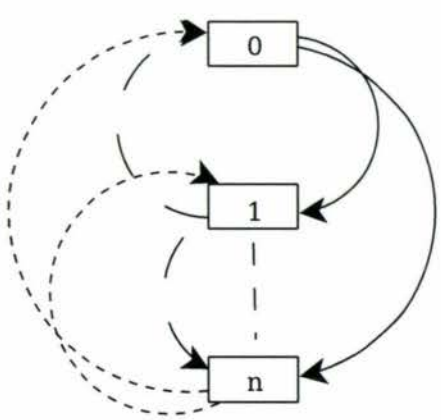

Figure 4.1: Broadcast Synchronisation

Because this is being implemented on the UDP layer it must be assumed that any messages maybe lost. Furthermore there is no phase three involved because the there is no master, once a node realises it is not needed anymore to synchronise it continues.

4.1.1 Naive Approach

Each node carries an array of boolean flags, the algorithm is as follows: Wait for everyone to to check in and stop when all nodes have checked in Until all of the nodes have checked in continue broadcasting node id.

In this approach the node takes into account only the communication between it and the other nodes (going from the other nodes to the node) if all messages from the node transmitting were blocked - the node would receive all of the messages from the other nodes and continue, while the other nodes would be blocked.

4.1.2 Knowledge Rows

Instead of the node only broadcasting its own Id, it also broadcasts its knowledge array, that is, what nodes it has heard from. This allows the node to check whether its messages were received by the all other nodes before continuing.

Also the node now contains a matrix, containing for each node in the group its current knowledge row. This works in the implementation however there have been some cases where the final message is lost:

```
node B sends A a message // A knows of B
Node A sends B a message // B knows of A (A knows B knows of A)
// Node A quits because all of the groups know about it and
//that it knows about all of the nodes

// if the message is lost then Node B stalls

?Node B sends A a message? message may be lost!
```

4.1.3 Results

The above synchronisation technique for using does not work all the time, however it exposes some characteristics of using broadcast channels. Firstly, Broadcast Synchronisation makes nodes very ineffective; conventional algorithms that use point to point messages are able to block when a certain node it is responsible for receiving a token. Where as when using broadcast channel, the node must constantly broadcast tokens until all of the nodes have successfully synchronised. This makes nodes inefficient to other processes running on the system at the time of synchronisation. Also it has been found that the speed is much better than using MPI_Barrier (point-to-point).

The application allows for some amount of non-synchronous operation because the problems that each node is working on is independent of the other.

It is more efficient to use MPI_Barrier to achieve synchronisation. This is found in the implementation of [Tinetti and Barbieri, 2003], there are implicit barriers before and after, firstly to make sure all of the nodes are ready for a data broadcast and the second to make sure no node has lost any information. In this application it is better to drop the frame and capture the next than to re-request another frame (see sec. 2.6.1).

4.2 Broadcasting Protocol

The User Datagram Protocol allows broadcasting of data to multiple nodes where as stream oriented protocols such as STCP and TCP are inherently designed for point to point type communications. The approach that was first considered was putting the network cards in promiscuous mode which allow them to receive all network traffic on the network. One of the drawbacks is that if the send address is of a valid node on the network the switch memorises it and will not send the same packet to any other node. This problem can however be circumvented by sending to an address that is never going to be on the network however the broadcasting method is used. The following details the construction of the light weight protocol, some of its rationale and finally its strengths and weaknesses in comparison to other protocol stacks.

Broadcast messages can be sent by sending to either a multicast address (class E) (as in IPV6, broadcasting is not supported, instead the multicast technique must be utilised [Hall, 2007]) or by sending to the highest address in the host part of the IP address space. For example, if the network is of the setup like this: 192.168.0.0/24 (first 24 bits of the IP address reserved to the network portion), then its broadcast address is 192.168.0.255. When this packet reaches layer 2 of the protocol stack, because broadcast packets don't have a particular destination they have their MAC address set to FF:FF:FF:FF:FF:FF. This also tells switches that this is a broadcast packet which is echoed across all ports. DHCP utilises this technique to acquire DHCP leases during network start up, and in certain cases (like the approach used in this work) a critical stage of system boot up.

Broadcasting together with spoofing can be used as an attack, someone can easily fake an IP address and request a ping response from a set of computers by the way of a broadcast ping request, causing a lot of traffic to be sent to the victim. In general the ability to rewrite packets is seen as a dangerous thing. On a Linux Kernel using the glibc interface any traffic being sent to a broadcast address must first have an option explicitly set on the socket before sending data. This operation does not require superuser privileges (root), however ensures that nothing is accidentally broadcasted.

UDP allows for broadcasting but also it is datagram oriented, that is there is no notion of stream-ness or data continuity in between packets and sometimes referred to as a connectionless protocol. As we will see this can both be advantageous as well as disadvantageous (see sec. 2.6.2). This is advantageous because more often than not, even in applications that require stream-ness a command terminator is included to indicate where a particular unit of transaction finishes, this is built in to UDP in a way because the unit is the packet. The primary disadvantage semantically speaking is that if data is seen as continuous rather than as a datagram the user must build her own protocol on top of UDP. It can also be noted that FTP uses a stream protocol (TCP) to transfer files, but instead of using a terminator or marker to signify EOF the connection is dropped.

IP packets can only carry a limited amount of payload, this isn't a problem with stream oriented protocols such as STCP or TCP and only an issue with UDP based protocols (see sec. 2.6.2). If data bigger than the payload is needed, then continuation (stream-ness) semantics must be built on top by the user. The protocol designed here called Image Transfer Protocol or ITP, is a specially designed lite-weight protocol for transferring image sequences over UDP. The intention was not to use compression and typical image sizes are much bigger than that of the packet size, even with grayscale images. The protocol described here handles this problem as well as providing some features such as multiplexing and ordering.

Figure 4.3 shows the structure of the packet. Context Id is used to provide multiplexing capability, the idea is that if ever in the future there needs to be two separate groups of images sequences broadcasted on the same port, then this value can be used to group image sequences together. Sequence id's are used to order the frames in a sequential manner, and also serve as a way to id frames on the sequence. Sequence id along with size and offset parameters are important for data consistency.

Algorithm 1 shows the general pseudo code for the send routine, the initialisation stage consists of initialising a packet buffer with the some initial headers. The loop consists of putting the correct data into the packet, adjusting the header values such as offset and sending the packet.

Algorithm 1 Image Send Routine send_image

```
while n ¡ img_size do
  assemble packet
  assemble other stuff
  ret = send(fragment)
  n += ret
end while
```

Algorithm 2 shows the general pseudo code for the recv routine, the initialisation stage consists of initialising packet and image buffers. The loop consists of receiving data from the broadcast channel, checking the headers for example to see that there are no missing fragments, and appending data fragments to image buffer.

UDP when overloaded to perform TCP like functions, that is provide in-order reliable delivery performs poorly in comparison to TCP (see sec. 2.5). Hence the protocol is kept very short. Also for the application of real-time video processing it does not matter if frames are lost or corrupted, it can simply be replaced with another frame. The protocol is ideal for this application. Some performance measurements are taken to measure the CPU load and protocol efficiency in the following chapter including throughput rate, sender CPU load and receiver CPU load.

4.3 LSPIP: Loosely Synchronised Parallel Image Processing Library

This library was developed for this work to bring together the image broadcasting mechanism as well as a framework for designing parallel image processing applications. This framework

Algorithm 2 Image Receive Routine recv_image

```
while true do
  receive packet
  if local sequence number is zero and packet offset is zero then
    set current packet to that of received packet sequence number
  end if
  if local sequence number is not zero and local sequence number not equal to packet
  sequence number then
    drop fragment and restart by setting current sequence number to zero
  end if
  if packet offset is not adjacent to local packet offset then
    drop fragment and restart by setting current sequence number to zero
  end if
  if packet offset equals image size then
    return image to caller
  end if
  n += ret
end while
```

Figure 4.2: Enabling Broadcasting using BSD Sockets API

```
int opt=1;
//char opt = '1';
unsigned opt_size=sizeof(int);
if (setsockopt(send_socket, SOL_SOCKET, SO_BROADCAST, &opt, opt_size) < 0) {      if
        printf("setsockopt error: %s\n", strerror(errno));
        exit(1);
}
```

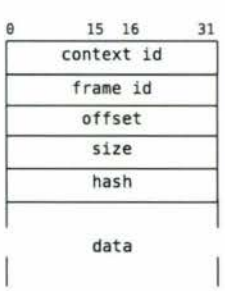

Figure 4.3: Packet Structure

is really a set of base programs which enable the development of re-usable non-synchronous applications.

```
Source - image source, can be replaced either by camera or video.
Process - receives image, processes and returns broadcasts results.
Sink - receives both images and results and renders to a screen
or a video sequence, also provides using statistics (fps).
```

This modular architecture has the advantage that any of the components can be restarted without effecting the others, of course if the source is modified then the process and sink are effected.

The sink component uses internally the use of thread via the pthreads (a standard POSIX threading API) library. If the process sends a result very close to or after the acquisition of a new frame from the source, the result is never seen as a bounding box drawn over the object. Hence a FIFO buffer with automatic decay of old object is used. This allows for results to be seen even if they are not valid per se. Although in video sequences a result even if from an older frame is usually relevant as motion sequences change very slightly in their content from frame to frame.

The library has been designed to operate with OpenCV in mind and so interoperate with IplImage (the native storage format used through out the library) structure and functions take a pointer to a pre-allocated IplImage to write/read.

```
API
---
int init_udp_send_unicast(char *address, int port);
int init_udp_send(char *address, int port);
int send_image(IplImage *img, unsigned int context_id, unsigned int frame_id);

int init_udp_recv(int port);
int recv_image(IplImage *img);
```

The server (in this case the source) would call the function init_udp_send(ADDRESS, PORT), where address is a broadcast address and port is predefined to be 4001. After this the server can simply send an image by capturing if from whatever source (camera/file) and calling the send_image() routine.

The client (process and sink) would call the init_udp_recv(PORT) and then call recv_image() to receive any images send by the source process (located on any computer in the cluster). The function takes care of all of the error detection and decoding the image data into proper RGB or grayscale values before returning.

Note the (interesting) reversal, usually 'servers' tend to call the recv routine to bind to ports but here it is the other way around, data providers start by sending and clients bind to receive data.

Similar functions exist for the aggregation of results and the summary / code is available in the appendices.

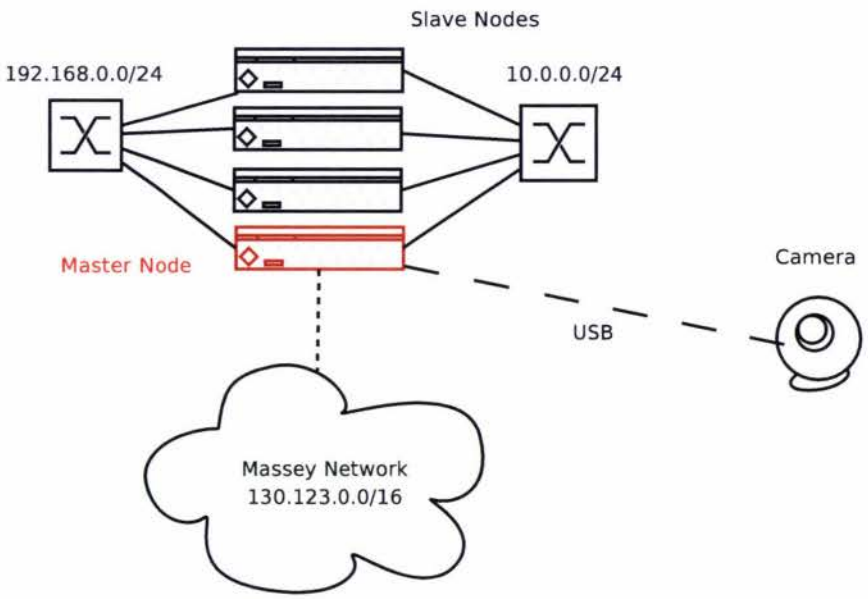

Figure 4.4: Network Layout, source [Barczak and Chemudugunta, 2006]

In figure 4.4 the network layout of the cluster is shown. The master node would the source here because it is connected directly to the USB camera. Each of the slaves run the 'process' process and the sink process can be run anywhere where there is a display attached. The diagram also shows that the master node contains an aliased network interface card in order for it to be attached to the network allowing to login/transfer files from the development computer.

4.4 Parallelising Detectors

The Viola Jones Detector shows a high degree of parallelism for multi-resolution object detection in images, and also when multiple classifiers are present in the detection pipeline. A single Viola Jones detector is a pipeline, organised as layers the sub window in question must pass through all of the stages of before it is marked as an object. Also, [Barczak et al., 2005b] shows how a set of classifiers can be parallelised by tying the input image to multiple processors with the added advantage that when there is more than one classifier per node the integral image can be reused.

4.5 Single Classifiers

From chapter two, Figure 2.3 shows how ViolaJones detector works. Also from chapter two, the Image frame cannot be decomposed by the basis of data as common Image processing do because of the scale the windows overlap each other so it would hard to decompose the image in a fair way. Also if the image is decomposed geometrically it could mean that the object is now split and this would prevent its detection all-together.

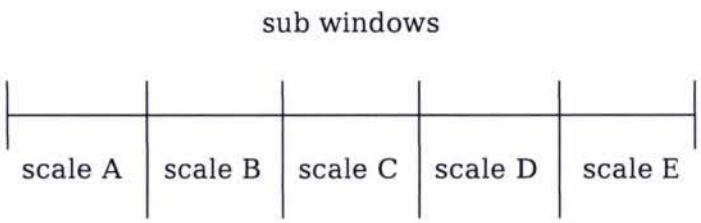

Figure 4.5: Subwindow List

A single classifier can be parallelised using a list of haarjobs, a haarjob is a data structure used to store the output from the subwindow generator. Because it is not easy divide the output of the subwindow generator purely based on its input parameters, the output is used to split the subwindow list among the nodes. This is effectively functional decomposition, because the data remains the same at all nodes, but the work of finding / eliminating subwindows is divided among multiple nodes.

Figure 4.5 shows how this list may appear, recalling also the generator algorithm presented in the literature review, for each scale such that the kernel size multiplied by the scale factor is not bigger than the image move the scaled kernel over the entire image with the given translation factor.

The amount of processing that a particular cascade performs is dependant on the image properties underneath the subwindow (see sec. 2.1.2). The more likely is the object of being detected, the more stages will pass and the more processing capacity will be consumed.

Because there are many parameters that control the subwindows generator, it is difficult to partition the jobs generated by this block by simply dividing the parameters. So instead, each of the node generates the whole list using the same parameters and then using a given strategy selects which subwindows it will work on.

```
typedef struct HaarJob {
int x,y;
double scale;
} HaarJob;
```

The HaarJob data structure specifies which scale and offset (x, y) the kernel is to be run.

4.6 Multiple Classifiers

When the number of classifiers is equal to or larger than the number of processors than each of the nodes can be dedicated to running an individual classifier. Figure 4.6 shows the general approach, a set of classifiers work on the input image and finally the results are aggregated at the end. In this particular case the different classifiers designated to each of the angles for face detection would be used on each of the nodes, however, any arbitrary set of classifiers can be run for example different classifiers for different objects. There are two approaches a fully synchronised parallel approach and a non-synchronised approach.

In a fully synchronised approach every node works in lock step where each of the nodes is doing the same task, either, get object - process image - send results. This approach can be implemented using parallel programming libraries like MPI with relative ease, however there are some performance pitfalls. One of the performance pitfalls identified in [Barczak et al., 2005b] was that the data delivery mechanism for the call MPI_Bcast is not a true broadcast, the time of distribution of a frame of video will vary depending upon the number of processors that are on the cluster. (cite the paper show MPI_Bcast approach with Ethernet broadcast) shows however that there exists a compatible call that takes advantage of the broadcast nature of Ethernet networks. Even though there exists such a call, it isn't completely free of the node scaling problem, one need to take into account that all nodes would have to synchronise with each other before and after the broadcast transmission to setup and check that the data has been transferred correctly. Efficient synchronisation strategies exist, such as the tree like strategy, but these are still not true broadcast. The first problem is of latency caused by protocol overhead.

The second problem is dependant on correctness. Because of nature of the detection method, two classifiers do not complete at exactly the same time. In a fully synchronous method of the nodes will be sitting idle until the other completes, when it could just pick up the next frame and continue detection. Allowing a node to continue while the other node is still computing an older frame means that the node computing the older node may never compute the current frame, depending upon how the input queue is managed. If then node consistency take longer than the frame rate than the system becomes non real-time. A non-synchronous approach has two advantages over a synchronised approach, faster data distribution the ability for a faster node to continue processing.

4.6.1 Distribution of Classifiers among Nodes

The way classifiers are distributed among various nodes can lead to interesting results. With hand classifiers there are 9 angles present, with both + and - orientations, if the assumption is made that there is only ever one object in the image then if in small sub group if classifiers they are grouped by orientation than the detection of one classifier can be used to abort the rest of the pipeline saving speed. This can be extended to allow for multiple objects by working at the sub window level, eliminating subwindows that are detected and passing the rest of them to the next cascades.

4.7 Dealing with Results

As published in the article in [Barczak and Chemudugunta, 2006], two different types of constraints are identified, performance penalty deadline and application specific deadline. Performance deadlines are real-time deadlines, that is if a node spends too long computing an older frame it automatically incurs a penalty as it must wait till another frame arrives there by reducing its affective processing fps.

However, application specific deadlines are different, they are deadlines set by the implementer, even though the result might be out of date, it is not completely useless as it maybe used to infer something about a certain object in the image sequence, like past position or in the interpolation of some future position.

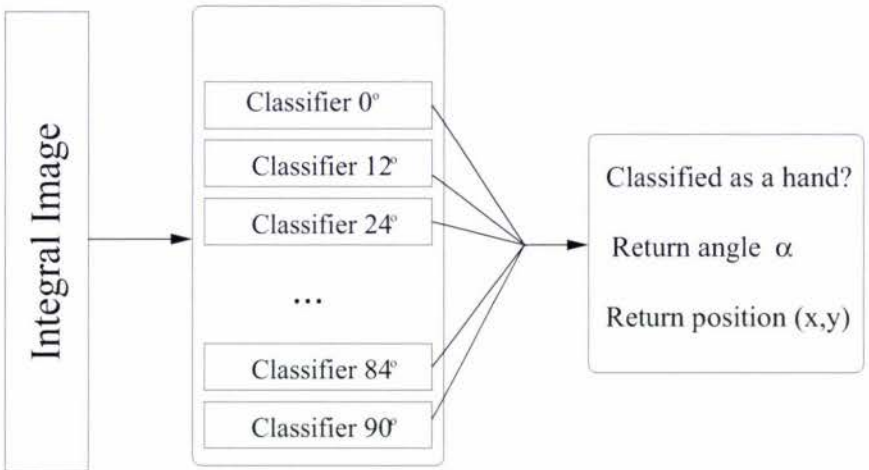

Figure 4.6: Parallel Classifiers, source [Barczak and Chemudugunta, 2006]

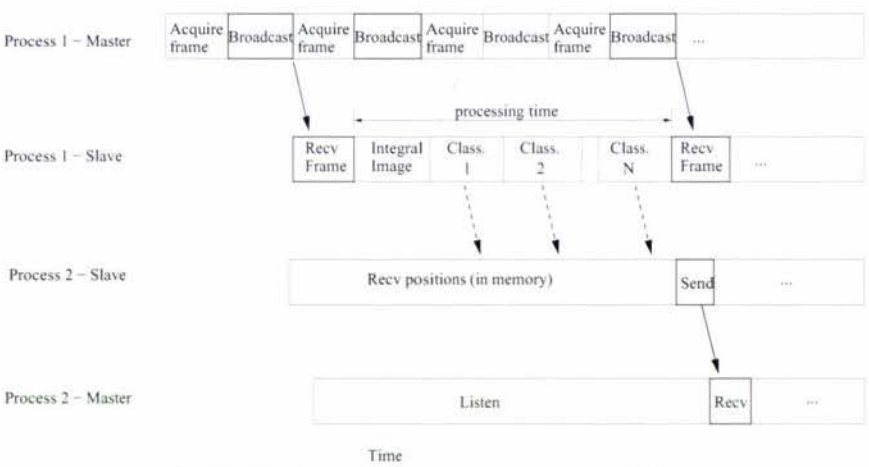

Figure 4.7: Communication Pattern 1, source [Barczak and Chemudugunta, 2006]

Figure 4.7 shows the communication pattern of a rather synchronous approach, in this approach all of the results are queued until the end and sent. Figure 4.8 shows a non-synchronous approach: the head node can receive results at any point in time. The advantage for sending the result as soon as it is available is that it results in quicker reception and it spreads the workload of sending results to the master node from multiple slaves.

4.8 Measuring System Efficiency

The performance characteristics of even serially executing code is hard to measure as it all depends on the image being presented, the classifier and its training method and a variety of other factors. To exhaustively examine every possibliity is computationally unfeasible. Therefore the performance models in [Barczak et al., 2005b] have to make assumptions such

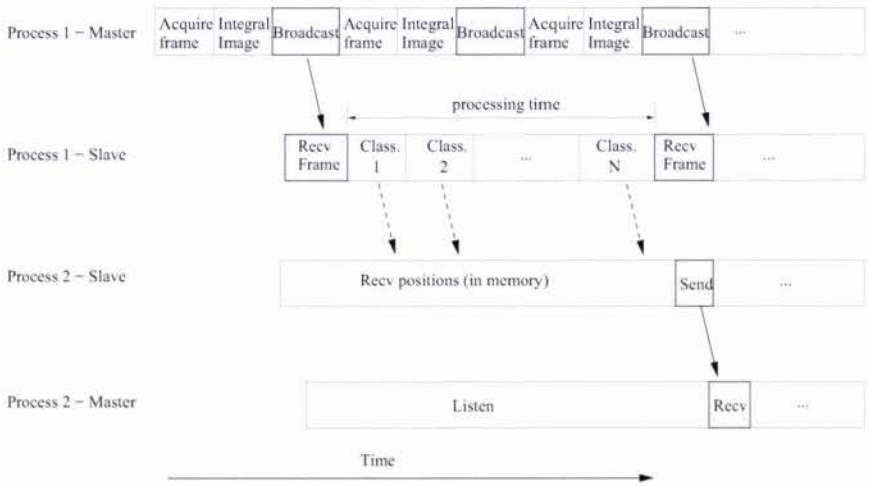

Figure 4.8: Communication Pattern 2, source [Barczak and Chemudugunta, 2006]

as the percentage of layers at each of the stages increase linearly. This is not an unreasonable approach as in fact the number of layers at each stage for a particular group of classifiers trained for hands detection have more or less the same amount of layers in each of the stages.

There is some difficulty in understanding what is going on in an non-synchronised application, because each of the nodes could be doing different things at the same time, therefore an approach of taking the average of the system is taken. The average frame rate is the average of the number of frames being processed in each of the different processors. There is also another measure for measuring the systems performance, the timely-ness of the answer, that is because this system is intended to be a real time system the timely-ness of the answer is very important. Timely ness is measured by whether or not the result received from a slave node is still within the bracket of the frame, i.e. until the next frame is available from the input source and all measurements become invalid.

This would be very easy in a synchronous application because the observed rate would be the rate at which the system was performing with no in consistent results. However in a non synchronous system it is harder to say how well the system is performing apart from empirical observations.

Two important metrics are important, the fps and the ops (objects per second). The fps tells us how many frames each of the nodes each of the nodes is able to process, this itself cannot be collectivised, how well is the system performing.

Objects per second is the number of objects that any one of the nodes is computing per second. This can be collectivised, that is the number of objects being computed across the whole network can simply be found by adding up all of the individual metrics. This can then be compared against a previous measurement to say how well the system is performing.

Since no correlation can be made to what layer a subwindow passes and the likeness to the object in question, one must only take into consideration those subwindows that have passed entirely and have been marked as objects by a previous classifier. That is, the process of finding an object does not get easier as the number of classifiers increase.

4.8.1 Testing Method, Image Sequences

To get a better understanding of how performance changes, different sequences with positive samples are injected into the sequence to cause a particular classifier to stress itself. It would be interesting to note also the disparity between the upper and lower level bound of frame rate on each of the nodes.

4.8.2 Testing Method, USB Camera

A USB camera can also be used to test the system, with some conditions it should be possible to generalise the experienced performance to most real-life conditions. For example, using the face detector and observing the performance with a few people in the frame would be a good enough real-world test.

4.9 Summary

In this section the Image protocol has been explained, its designed around the UDP protocol. The protocol and the programs used in the system are generalised into an application library, LSPIP that enables any object detection algorithm to work using the same code base.

The challenges for parallelisation and the strategies used to solve them are explained, including single classifiers and multiple classifiers. And finally testing methods are used.

The next chapter more details on test conditions and presents some results and findings of experiments.

Chapter 5

Results and Discussion

5.1 Network Performance

The following result are conducted on separate groups of clusters. The sisters is an cluster built at Massey University [Barczak et al., 2003]. In brief the cluster consists of 8 Dual Athlon compute nodes with 2 GB of RAM and interconnected via a gigabit switch. In the following experiment the efficiency of both MPI broadcast and UDP based broadcast calls are compared.

Figure 5.1 shows the performance of the UDP protocol vs the MPI broadcast call. It shows that for the purpose of video broadcasting with the available hardware it is far better to use than MPI broadcast routines. The graph also shows that the broadcast times are independent to the number of nodes in the cluster (up to 8 nodes only here). The reason for this is that modern intelligent switches are able to map MAC address/Physical Port number pairs in memory, however because the library routines broadcast on the broadcast address the switches revert to acting as simple hubs, repeating everything on every port.

5.2 CPU Performance

All of experiments detailed here forth are conducted on the embedded hardware. The following experiment only uses one of the four board available (as all of the nodes are equal in computer power) to examine the rate at which it can run the classifiers on the machine.

Table 5.1: Cluster Specifications relevant to performance

CPU	VIA C3 / Eden (x86 compatible
Memory	DDR266 512 MB
Network	Dual LAN, VIA VT6105 LOM 10/100 Base-T Ethernet

Figure 5.2 shows the performance of each of the classifiers with respect to a chosen scale factor and resolution of the input image. The input image is an average image, which consists of one face and no false detections.

The graph 5.2 shows that for high resolution images a large scale factor must be used to minimise computing time. At a nominal scale factor of 1.3 (shown to achieve good detection

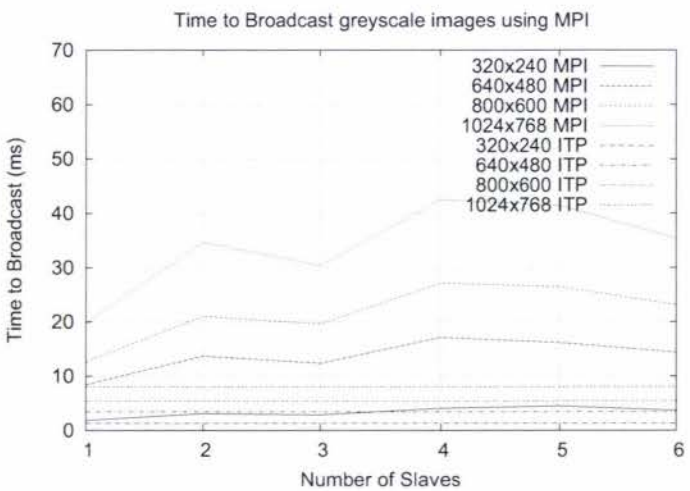

Figure 5.1: Broadcast Performance of MPI vs UDP Protocol

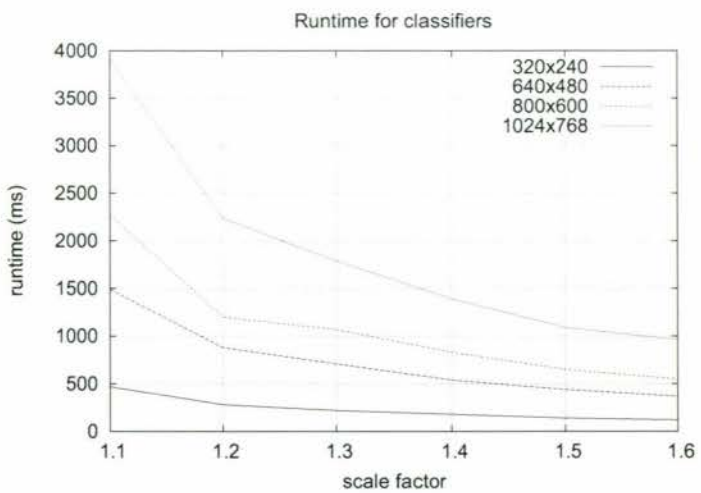

Figure 5.2: Classifier Performance on a Single Node

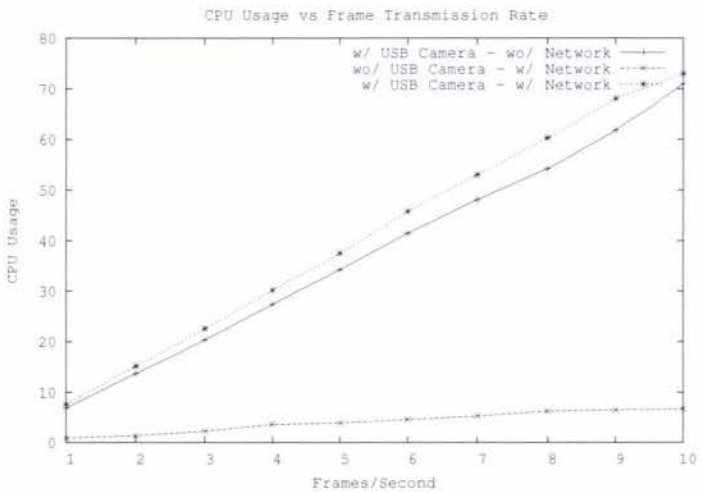

Figure 5.3: CPU vs Send FPS (640x480)

results) the results shows that only the two lower resolutions (320x240, 640x480) are capable achieving real-time performance (if, real is computing in less than a second time). It should be noted that the original Viola Jones [Viola and Jones, 2002] results are based on low resolution images and many implementations like OpenCV prescale the image into a smaller image before commencing detection.

5.2.1 USB Camera

USB cameras are comparatively cheaper and widely available however they are very expensive in compute cycles. USB is host-driven, meaning the main system CPU has to do all of the work of coordinating and moving data to/from devices.

Figure 5.3 shows that USB is an inefficient system for delivering images, especially to such low powered compute node. The bottom line (wo/ USB - w/Network) shows the compute cycles used when an NULL image is sent using the protocol, i.e with the USB capture functionality turned off within the application but leaving open the broadcasting. The middle line (w/ USB - wo/Network) shows the load of USB capturing but with no broadcasting in the loop. At 10 FPS the load is around 70% leaving very little room for any useful computation on the head node. The third line shows the full load experience by the 'source' process on the head-node. It confirms the load of sending an NULL image being consistant with USB, i.e. if the bottom line is added to the middle line the result is roughly the top most line (w/ USB - w/Network). However, USB can be used if the performance is sacified, or if can be implemented in a way that minimises its overhead.

All further results are shown with image sequences buffered to RAM rather than acquiring images via USB, to show the the speed up characteristics of cluster without biasing the headnode. This methods also allows experiments to be repeated with the same input data for comparison.

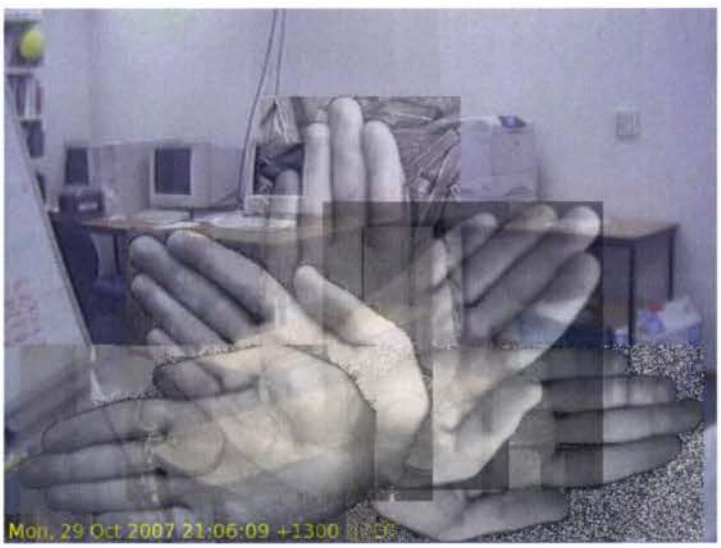

Figure 5.4: Test Sequence

5.3 Experimental Conditions

The following experiments and the results presented are performed on the finished cluster setup. The layout of the cluster can be found in figure 4.4 and the details of the components are listed in 5.1.

The head node connected runs the 'source' and 'process' modules, where as all of the compute nodes run the 'process' module only, except the last node that runs the 'sink' module in addition to the 'process' module.

All measurements were taken with hands sequence 5.4, overlay-ed over a background image as the accuracy wasn't really what was being tested for but the efficiency of the system as a whole. The sequence consists of the following workload, -90 to 90, there are a total of 190 images in the test sequence.

The experiments were repeated for three resolutions (hands sequence), and loading nodes with varying numbers of classifiers sharing the same integral image. The nodes were loaded serially i.e. when an additional classifier was added it would be added to the node next in the line (or round robin). This resulted in the nodes having classifiers that were not locally affinated. A possible strategy would be to explore the performance when nodes are loaded using classifiers that are locally tuned.

Because the whole system is not synchronised, the send fps is capped at 10 frames per second. The entire sequence of 190 image therefore takes 19 seconds to complete broadcasting. Each of the nodes reports the number of frames processed in this time segment. This value can then be divided by 19 to obtain an FPS value.

5.4 Average System Performance

Figure 5.5 shows the performance of the system with varying number of classifiers.

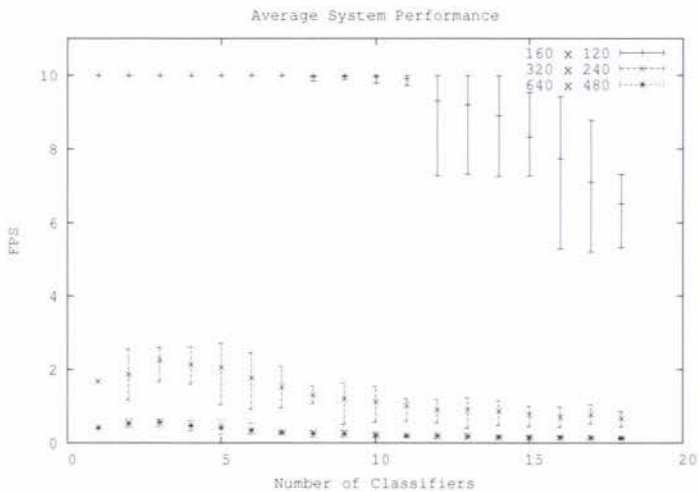

Figure 5.5: Average System Performance

Figure 5.5 shows performance levelling out as the number of classifiers increases. This is also a reflection of the characteristics of the classifiers running on a single nodes. This behaviour of holding steady after a certain number of classifiers does not continue forever, i.e. eventually the frames per second will drop to 0, when measured within some reasonable time interval. In this instance its measured as the number of frames processed within the 190 frame transmission sequence time (19 seconds).

Also the measurement for the lowest resolution seems to indicate that there is an asymptote close to 13 classifiers that will hit the the ground, however there is an another upward asymptote (which cannot be seen here, as the number of classifiers that can be loaded is bound) that would show similar behaviour as the curves for the other two resolutions, the curves will flatten out and remain roughly constant even after addition of more classifiers. This behaviour can be seen in figure 5.6, performance per node for 160 by 120.

5.5 Individual Node Performance

Each of the compute nodes running the 'process' process is able to give output of how many frames it is able to capture (infact this is the only way to record what is happening in an asynchronous system). The following results show what each node experienced in isolation with the rest of the system.

For Nodes 1 and 2 there is an extra measurement of 5 classifiers, this is due to the fact that there are only 18 classifier available and only 4 nodes. By using the round robin classifier distribution strategy Node 1 gets classifier 17 and Node 2 gets classifier 18.

Figure 5.6 shows very good performance as this is for the lowest resolution. It should be noted that for number of classifiers betwen 1 and 3 (on the x-axis), the performance is report is at maximum, 10 frames per second (the input rate), however it is possible the nodes can run faster (for this particular resolution) then this if the input rate was increased.

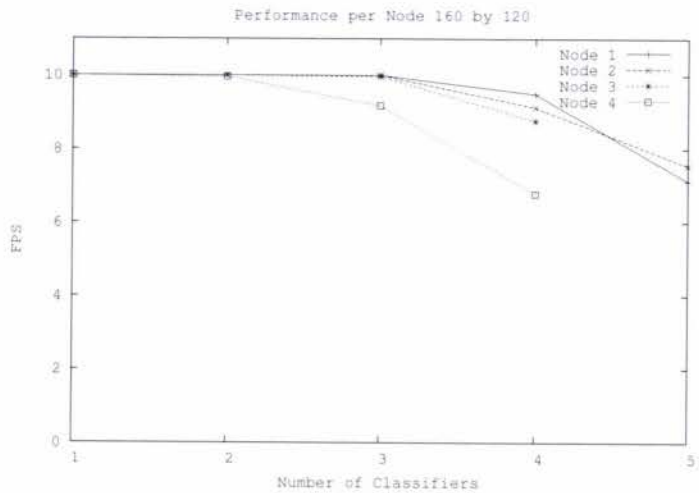

Figure 5.6: Individual Node Performance for 160x120

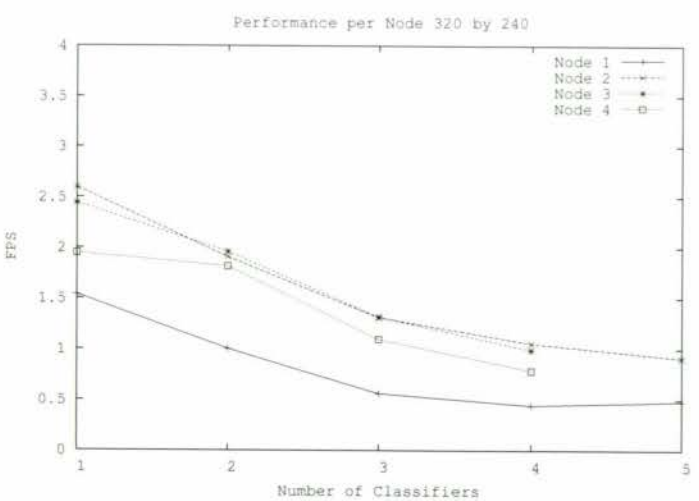

Figure 5.7: Individual Node Performance for 320x240

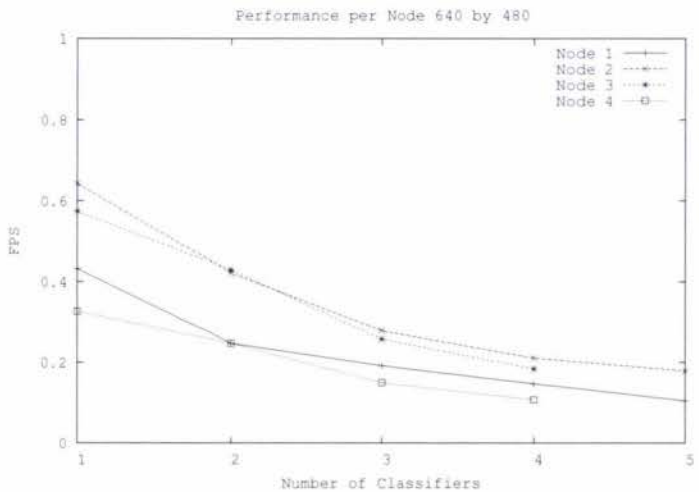

Figure 5.8: Individual Node Performance for 640x480

These figures 5.6, 5.7, 5.8, also show that the nodes operate independently to each other, whether or not other nodes are operating with the same parameters the output frame rate remains the same. This can also been seen in figures on scalability in the following section.

Also as can be seen from the individual node performances, the performance at a resolution of 640 by 480 (figure 5.8) is very poor and saturates to floor very quickly.

5.6 Scalability

Scalability refers to how scalable the system is as a whole [Wilkinson and Allen, 1999], that is if more resources were added how well does the system use these resources. The aim of this exercise is to typically observe the limit of the system when things saturate so there is no more gain in adding resources.

The figures (5.9, 5.10, 5.11) show the scalability of whole system with respect to the number of nodes. They show that the system is very scalable in terms of the number of nodes, though it only contains 4 nodes, measurements of the protocol have shown similar results for number of nodes up to 8 (see figure 5.1).

The slight dip in the frames per second on all three figures towards (number of nodes greater than 3) is due to the fourth node being slower. This can be seen also in figures 5.6, 5.8 where the individual node performance of node 4 is consistently slower than the other nodes. Node 4 is assigned the task of rendering the results, this slows down the node and pulls down the average of the frames per second when node four is involved.

This is especially visible in figures 5.11 and 5.10 where the node 4 has to do a lot of work rendering the image to the screen. If one were to arrange the nodes such that the slower nodes were added first then it would seem like the scalability is greater than linear.

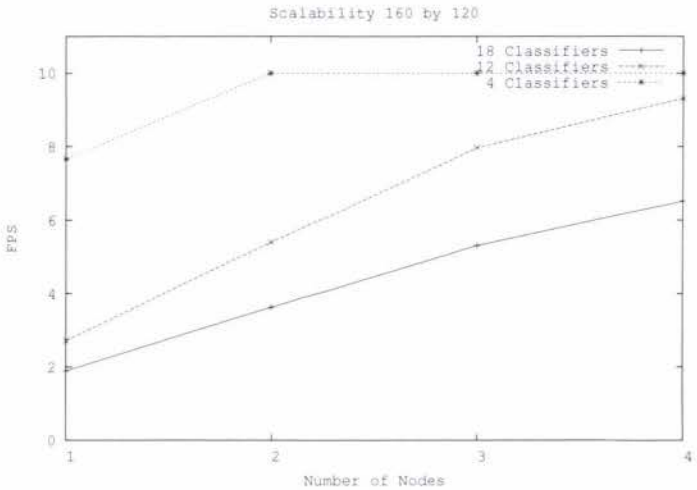

Figure 5.9: Performance Scalability for 160x120

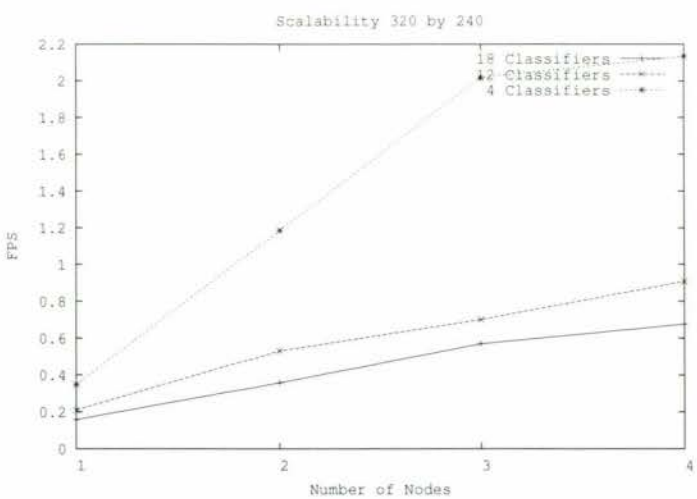

Figure 5.10: Performance Scalability for 320x240

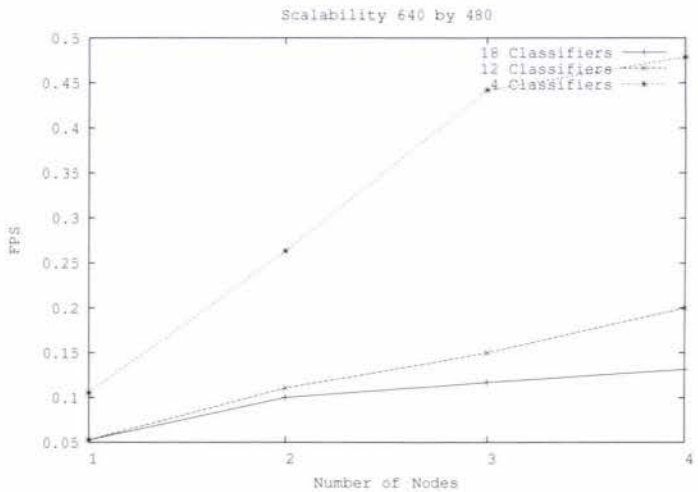

Figure 5.11: Performance Scalability for 640x480

5.6.1 Resolution Scaling

Resolution scaling refers to how the performance is affected when the problem size is increased. In this instance the resolution is varied (keeping constant all of the other variables such as scaling and translation factor).

All of the four graphs (5.12, 5.13, 5.14 and 5.15) show that performance is best at the lower resolution of 160 by 120 and degrades rapidly as it is increased. As the number of nodes increased, the rate at which performance degrades also seems to decrease. However, this may be in part to the increased frames/second rate when more nodes are added.

5.7 Performance Analysis

Going from the lowest to the highest resolution it can be seen that the II (integral image, from this point onwards they will be referred to as II) computation phase takes almost 16 times longer while the number of subwindows to be examined is 42 times that it is before.

Table 5.2: II Computation Time and Sub Window Load for each of the Resolutions

Resolution	II Compute Time	Number of Sub Windows (Kernel = 24x24, Scale = 1.3, Translation = 5)
160 x 120	i	s
320 x 240	4 i	7 s
640 x 480	16 i	42 s

There seems to be large differences between 320x240 and 160x120. Table 5.7 shows the relation between various resolutions and their work loads. Here 'i' is a symbolic of the number of pixels in the in the lowest resolution, where as s is symbolic of the number of subwindows

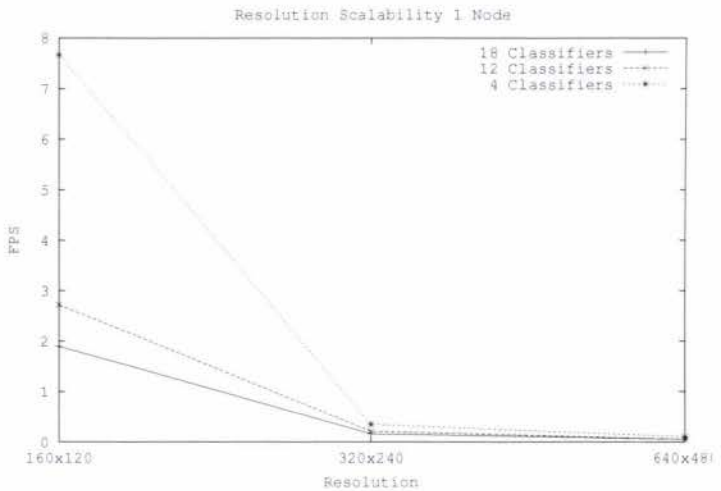

Figure 5.12: Resolution Scaling for 1 Node

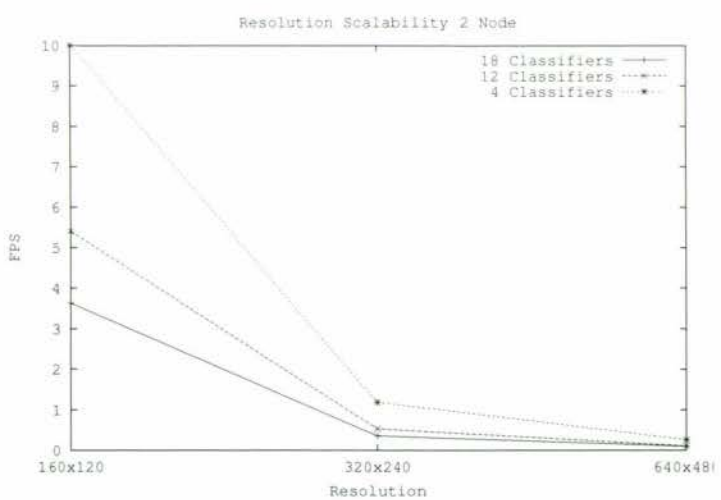

Figure 5.13: Resolution Scaling for 2 Nodes

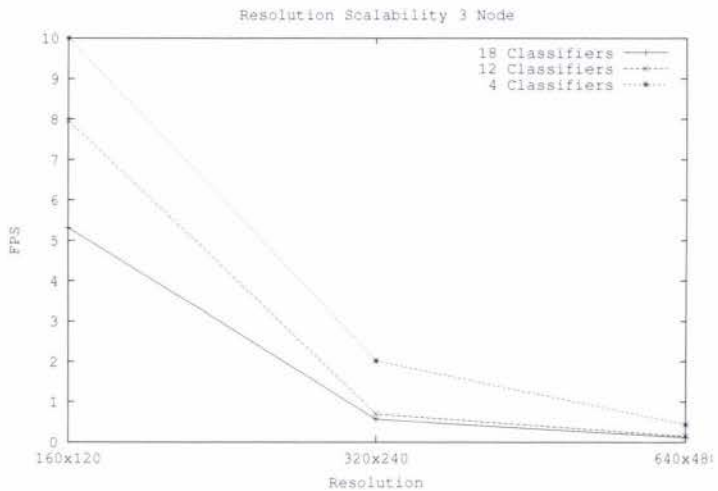

Figure 5.14: Resolution Scaling for 3 Nodes

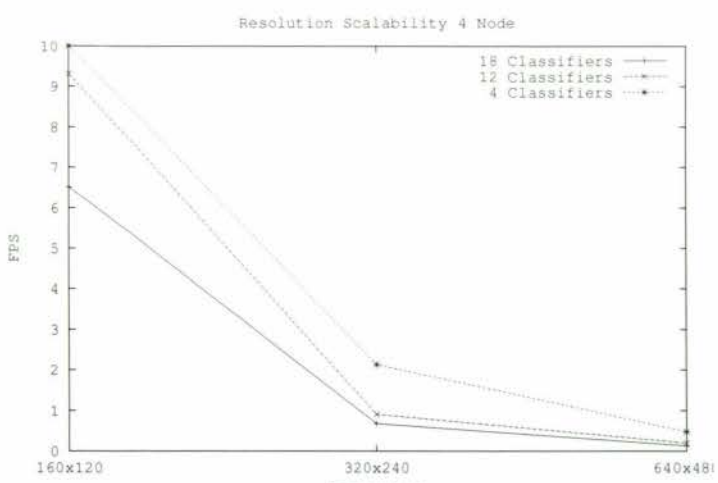

Figure 5.15: Resolution Scaling for 4 Nodes

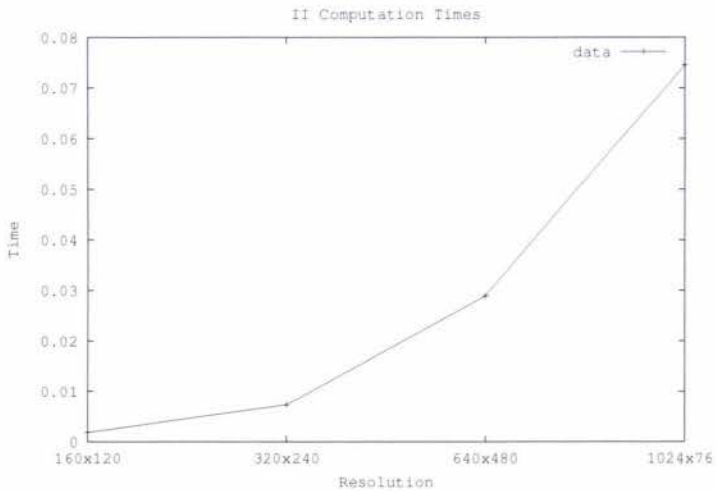

Figure 5.16: Integral Image Computation Times

generated. The process of generating an integral image is linear in relation to the number of pixels and this can be observed in figure 5.16 (note: multiply width*height).

The II computation phase for 320x240 is four times bigger than 160x120 and 640x480 is 16 times bigger than the smallest resolution, however it isn't just the picture size but the number of subwindows at each resolution. The work load for the number of subwindows for resolutions is not linear, so it grows rapidly as the resolution is increased.

Chapter 6

Conclusion and Future Work

6.1 Conclusions from Results

The UDP broadcasting protocol is efficient and able to scale very well to increasing number of hosts in comparison with MPI for the purpose of video broadcasting.

USB is a host driven protocol meaning the CPU has to do everything on its behalf, like moving data to and from the device. USB is an inefficient system for delivering images, especially for such low powered nodes, however it can be used but sacrificing some performance.

Average System Performance shows performance evening out over as more classifiers are added. This also shows in individual node results also. Both are also indicative of the performance of multi-classifier cascades that are able to reuse integral image computation.

The nodes perform poorly at higher resolutions, for best performance the system must be used with resolutions between 160 by 120 and 320 by 240.

Individual Node Performance shows that the nodes operate independently. This is good because the slower nodes do not slow down the whole system to operate at the same rate, however this makes finding out what is happening difficult as each node must be queried separately.

The system is very scalable, this is in part due to the asynchronous operation but also the efficiency of the image broadcast protocol. Adding more nodes also reduces the performance degradation of system as the resolution is increased.

Analysis of performance shows that the load increases non-linearly and explains why there is a large performance drop when going from 160 by 120 to 320 by 240.

The results indicate that the architecture of using networked computers is viable for performing real-time vision processing; with a little more computer power, it should be possible to run more classifiers or increase the resolution given the scalability of the system is very good.

6.2 Perspectives

Chapter 3 highlights some of the challenges faced during the building of the distribution for the cluster. It is important to have the software platform working very early on in a project.

The focus on reducing distribution (linux) space, while crucial to the successful deployment of the system takes up a large amount of time. Currently the embedded distribution space is very fragmented. There are many possible build systems and distributions that can be used.

Always opt for an open system, one where the build system and documentation are readily available. Commercial targeted distributions such as MontaVista should be avoided whenever possible. Choosing an open/large system allows development to be carried out on the same distribution that is chosen for the target and makes it easier to run software on the target system without porting and dependencies issues.

The current build system that is named 'mooselinux' and developed for this thesis is reusable and can be used for anyone desiring to build a customised distribution without loosing the ability to modify and add to it later. With this build system both Ubuntu and Debian can be used, both which contain large repositories of pre-compiled software and libraries.

6.3 Future Work

6.3.1 Parallelising Single Classifiers

The method explored in this thesis relies on having more classifiers than the number of nodes available. However the question of whether it is possible to parallelise a single classifier is interesting. While single classifiers are accurate, they are unable to adapt to changes of the object when rotation is present. Single classifiers can be trained with object data at multiple orientations, however this has shown poor detection rates and takes longer to train due to the additional training samples required.

Initial results show that it is possible to parallelise an individual classifier. By splitting the subwindow list (generated by the subwindow generator algorithm presented in chapter 2), each node can be assigned a part of the list to compute.

6.3.2 Strategies of Applying Multiple Classifiers

If the task of deciding what to run is more expensive or the savings are menial then actually running the task is better. With this in mind we go ahead and observe a policy for distribution, its requirements on the underlying protocol, its overhead and finally the system output - comparing it to the generic version.

6.3.3 Moments Based Detection Algorithms

Moments based detection system shows good detection rates that uses precomputed integral image like the Viola Jones detector, but with the added advantage that training times are remarkably smaller [Barczak and Johnson, 2006]. It would be interesting to explore the performance characteristics of this algorithm on this system.

6.3.4 Other Hardware Platforms

The current system, while being portable could still be made smaller. A new platform would most likely be ARM based processor packaged as an System On Chip that is able to provide many of the same peripheral interfaces as were available on the x86 solution. The ARM route is attractive as they are cheap, lower power and support the same software as the x86 platform. Debian and Ubuntu both support the same software repository for both x86 and ARM platforms. The build system 'mooselinux' would be able to adapt to this easily as all that would be required for the change would be to modifying the repository location from x86 to ARM.

DSP Hardware

The process of finding out whether a subwindow contains an object of interest is in itself a parallelisable task. The task of detection can be summarised to, high-speed lookups, arithmetic operations and control flow manipulation. DSPs hardware are a special class of processors that is designed to work one or more streams of data at a high rate (having multiple ALU units). DSPs also contain very localised memory that gives them the ability to access memory rapidly (an II can be stored, giving a speed up).

Further these units are also very efficient and us very little power. Many commercial devices such as cell-phones employ specialised hardware in the form of a DSP for complex and resource intensive video processing while leaving control to the main processor. A similar architecture would be beneficial for this application as it would decrease the power usage of the system while giving more computational power and maintaining mobility of the system.

GPGPU Hardware

One of the interesting hardware platforms that has emerged is the use of GPU hardware for image processing. GPGPUs are very similar to DSPs however they differ in that their pipeline is reasonably fixed. This is starting to change however with programmable units. Modern graphics hardware are very powerful in the raw amount of data they are able to process. As applications demand greater amount of hardware acceleration, powerful GPU will be embedded onto mainboards (as is the case already with top of the line laptops, 2008). This would provide essentially an off the shelf solution for using special purpose hardware [Fung and Mann, 2004].

Appendix A

UDP Image Broadcast Header File

```
#ifndef UDP_IMAGE_H
#define UDP_IMAGE_H

#include "hseih.h"
#include <opencv/cv.h>M̂
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <stdlib.h>
#include <memory.h>

#define MAX_UDPIMAGE_SIZE 64000
//#define MAX_UDPIMAGE_SIZE 32768
#define THROTTLE 000000

typedef struct {
    unsigned int context_id; // 4 bytes
    unsigned int frame_id;   // 4 bytes
    unsigned int offset;     // 4 bytes
    unsigned int size;    // 4 bytes
    unsigned int hash;    // 4 bytes
    unsigned char data[MAX_UDPIMAGE_SIZE];
} udpImage;

#define UDPIMAGE_STRUCT_SIZE (MAX_UDPIMAGE_SIZE+20)
#define DEBUG_UDP_BCAST 1
```

```
//#define ADDRESS "192.168.0.255"
#define ADDRESS "127.255.255.255"
#define PORT 4001
#define WIDTH 640
#define HEIGHT 480
#define DEPTH 1
/*
extern int send_socket;
extern struct sockaddr_in ca;
extern int recv_socket;
extern struct sockaddr_in sa;

extern int current_frame_id;
extern int last_frame_id;
extern int current_frame_offset;
*/
int init_udp_send_unicast(char *address, int port);
int init_udp_send(char *address, int port);
int send_image(IplImage *img, unsigned int context_id, unsigned int frame_id);

int init_udp_recv(int port);
int recv_image(IplImage *img);
#endif
```

Appendix B

UDP Image Result Collection Header File

```
#ifndef UDP_RESULT_H
#define UDP_RESULT_H
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <stdlib.h>
#include <memory.h>

typedef struct {
    unsigned int context_id; // 4 bytes
    unsigned int frame_id;   // 4 bytes
    int x1;
    int x2;
    int     y1;
    int     y2;
} udpResult;

//#define RESULT_ADDRESS "192.168.0.1"
#define RESULT_ADDRESS "127.255.255.255"
#define RESULT_PORT 4002

static int send_socket_result;
static struct sockaddr_in ca_result;
```

```
int init_udp_send_result(char *address, int port);
int send_result(udpResult *res);

static int recv_socket_result;
static struct sockaddr_in sa_result;

int init_udp_recv_result(int port);
int recv_result(udpResult *res);
int recv_result(udpResult *res, char *host);

#endif
```

Appendix C

LSPIP: 'Source' Program

The following program sources from a list of images, allowing offline testing without a camera.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <math.h>
#include <string.h>
#include <ctype.h>

#include <opencv/cv.h>
#include <opencv/highgui.h>                                          10

#include <udp_image.h>

#include "timer.h"
#include "filelist.h"

IplImage *resizedImage = 0;
IplImage *grayScaleImage = 0;

IplImage *fetch_new_image(CvCapture *capture) {                      20 fetch_new_image
    if ( !capture ) return NULL;

    cvGrabFrame(capture);
    IplImage *image = 0;
    image = cvRetrieveFrame(capture);

    if (!image) {
        printf("error reading image\n");
        exit(1);
    }                                                                30
```

```
    if ((image->width != WIDTH) || (image->height != HEIGHT)) {
        if (resizedImage == 0) {
            resizedImage = cvCreateImage(cvSize(WIDTH, HEIGHT), image->depth, image->nChannels);
        }
        cvResize(image, resizedImage);
    }

    if ((resizedImage != 0) && (resizedImage->nChannels != DEPTH)) {
        if (grayScaleImage == 0) grayScaleImage = cvCreateImage(cvSize(WIDTH, HEIGHT), 8, 1);
        cvCvtColor(resizedImage, grayScaleImage, CV_BGR2GRAY);
        return grayScaleImage;
    }

    if (image->nChannels != DEPTH) {
        if (grayScaleImage == 0) grayScaleImage = cvCreateImage(cvSize(WIDTH, HEIGHT), 8, 1);
        cvCvtColor(image, grayScaleImage, CV_BGR2GRAY);
        return grayScaleImage;
    }
    return image;
}

int main(int argc, char *argv[]) {
    CvCapture *capture = 0;
    if (argc > 1) {
        printf("capturing from file %s\n", argv[1]);
        capture = cvCaptureFromAVI(argv[1]);
    } else {
        printf("capturing from camera\n");
        capture = cvCaptureFromCAM(0);
    }

    if (capture == NULL) {
        printf("error init capture device\n");
        exit(1);
    }

    init_udp_send(ADDRESS, PORT);

    IplImage *img = 0;
    int n = 0;

    while (1) {
        img = fetch_new_image(capture);
        if (!img) {
            printf("error capturing frame\n");
        }
```

```
        send_image(img, 1001, n);
        //printf("Sleeping for 1 second\n");
        //sleep(1);                                                    80
    //incorporate fpslimiter into this
        n++;
    }
}
```

Appendix D

LSPIP: 'Process' Program

The following program processes an image from the network using Viola Jones Algorithm, this can be replaced with any detection algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <math.h>
#include <string.h>
#include <ctype.h>

#include <opencv/cv.h>
#include <opencv/highgui.h>

#include <udp_image.h>

#include "timer.h"
#include "filelist.h"

IplImage *resizedImage = 0;
IplImage *grayScaleImage = 0;

IplImage *fetch_new_image(CvCapture *capture) {
    if ( !capture ) return NULL;

    cvGrabFrame(capture);
    IplImage *image = 0;
    image = cvRetrieveFrame(capture);

    if (!image) {
        printf("error reading image\n");
        exit(1);
```

```
    }                                                                          30

    if ((image->width != WIDTH) || (image->height != HEIGHT)) {
        if (resizedImage == 0) {
            resizedImage = cvCreateImage(cvSize(WIDTH, HEIGHT), image->depth, image->nChannels);
        }
        cvResize(image, resizedImage);
    }

    if ((resizedImage != 0) && (resizedImage->nChannels != DEPTH)) {
        if (grayScaleImage == 0) grayScaleImage = cvCreateImage(cvSize(WIDTH, HEIGHT), 8, 1);  40
        cvCvtColor(resizedImage, grayScaleImage, CV_BGR2GRAY);
        return grayScaleImage;
    }

    if (image->nChannels != DEPTH) {
        if (grayScaleImage == 0) grayScaleImage = cvCreateImage(cvSize(WIDTH, HEIGHT), 8, 1);
        cvCvtColor(image, grayScaleImage, CV_BGR2GRAY);
        return grayScaleImage;
    }
    return image;                                                              50
}

int main(int argc, char *argv[]) {                                             main
    CvCapture *capture = 0;
    if (argc > 1) {
        printf("capturing from file %s\n", argv[1]);
        capture = cvCaptureFromAVI(argv[1]);
    } else {
        printf("capturing from camera\n");
        capture = cvCaptureFromCAM(0);                                         60
    }

    if (capture == NULL) {
        printf("error init capture device\n");
        exit(1);
    }

    init_udp_send(ADDRESS, PORT);

    IplImage *img = 0;                                                         70
    int n = 0;

    while (1) {
        img = fetch_new_image(capture);
        if (!img) {
            printf("error capturing frame\n");
```

```
        }
        send_image(img, 1001, n);
        //printf("Sleeping for 1 second\n");
        //sleep(1);
    //incorporate fpslimiter into this
        n++;
    }
}
```

Appendix E

LSPIP: 'Sink' Program

The following program processes image from the network as well as results from the 'process' stage and render these to the screen. This program can alternatively be replaced with a video writer that can save the results to an video file.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <math.h>
#include <string.h>
#include <ctype.h>

#include <opencv/cv.h>
#include <opencv/highgui.h>                                          10

#include <udp_image.h>

#include "timer.h"
#include "filelist.h"

IplImage *resizedImage = 0;
IplImage *grayScaleImage = 0;

IplImage *fetch_new_image(CvCapture *capture) {                      20 fetch_new_image
    if ( !capture ) return NULL;

    cvGrabFrame(capture);
    IplImage *image = 0;
    image = cvRetrieveFrame(capture);

    if (!image) {
        printf("error reading image\n");
```

```
        exit(1);
    }

    if ((image->width != WIDTH) || (image->height != HEIGHT)) {
        if (resizedImage == 0) {
            resizedImage = cvCreateImage(cvSize(WIDTH, HEIGHT), image->depth, image->nChannels);
        }
        cvResize(image, resizedImage);
    }

    if ((resizedImage != 0) && (resizedImage->nChannels != DEPTH)) {
        if (grayScaleImage == 0) grayScaleImage = cvCreateImage(cvSize(WIDTH, HEIGHT), 8, 1);
        cvCvtColor(resizedImage, grayScaleImage, CV_BGR2GRAY);
        return grayScaleImage;
    }

    if (image->nChannels != DEPTH) {
        if (grayScaleImage == 0) grayScaleImage = cvCreateImage(cvSize(WIDTH, HEIGHT), 8, 1);
        cvCvtColor(image, grayScaleImage, CV_BGR2GRAY);
        return grayScaleImage;
    }
    return image;
}

int main(int argc, char *argv[]) {
    CvCapture *capture = 0;
    if (argc > 1) {
        printf("capturing from file %s\n", argv[1]);
        capture = cvCaptureFromAVI(argv[1]);
    } else {
        printf("capturing from camera\n");
        capture = cvCaptureFromCAM(0);
    }

    if (capture == NULL) {
        printf("error init capture device\n");
        exit(1);
    }

    init_udp_send(ADDRESS, PORT);

    IplImage *img = 0;
    int n = 0;

    while (1) {
        img = fetch_new_image(capture);
        if (!img) {
```

```
            printf("error capturing frame\n");
        }
        send_image(img, 1001, n);
        //printf("Sleeping for 1 second\n");
        //sleep(1);                                                    80
    //incorporate fpslimiter into this
        n++;
    }
}
```

Appendix F

Mooselinux Build System: 'Get'

The following python program is part of mooselinux distribution builder.

```
#!/usr/bin/python

import os
import sys
import urllib
import zlib
from gzip import GzipFile
import pickle

global cache
cache = {}

PACKAGE_CACHE = "package_cache"
DOWNLOAD_CACHE = "download_cache"
DATA = "data"

def check(pkg):
    print "checking " + pkg
    return 0

def build(pkg):
    print "building " + pkg
    try:
        filename = cache[pkg].split('/')
    except:
        print "package not found"
        return

    filename = filename[len(filename) - 1]
    if (not os.path.exists(DOWNLOAD_CACHE + "/" + filename)):
```

```
        download(pkg)
    try:
        os.system("dpkg-deb -x download_cache/" + filename + " tmp")
    except:
        print "error building package " + pkg
        exit
    os.chdir("tmp")
    os.system("rm -rf usr/share/man")
    os.system("rm -rf usr/share/doc")
    try:                                                                    40
        if (os.path.exists(". ./" + DATA + "/" + pkg + "/files")):
            os.system("tar czfT " + ". ./" + PACKAGE_CACHE + "/" + pkg + ".tar.gz " + ". ./" +
        else:
            print "*** WARNING *** No pruning information found, applying global rules"
            os.system("tar czf " + ". ./" + PACKAGE_CACHE + "/" + pkg + ".tar.gz . ")
    finally:
        os.chdir(". .")
        os.system("rm -rf tmp/*")

def download(pkg):                                                          50 download
    print "downloading " + pkg
    filename = cache[pkg].split('/')
    filename = filename[len(filename) - 1]
    urllib.urlretrieve(cache[pkg], DOWNLOAD_CACHE + "/" + filename)

def get(pkg):                                                               get
    # if (os.path.exists(PACKAGE_CACHE + "/" + pkg + ".tar.gz")):
    #     if (check(pkg) == 0):
    #         build(pkg)
    # else:                                                                 60
    build(pkg)

def build_local(pkg):                                                       build_local
    package_name = pkg.replace("moose_", "")
    if (os.path.exists("local/" + package_name)):
        os.chdir("local/" + package_name)
        try:
            os.system("tar -czf . ./. ./" + PACKAGE_CACHE + "/" + pkg + ".tar.gz .")
        finally:
            os.chdir(". ./. ./")                                            70
    else:
        print "package not found"
        return

if __name__ == "__main__":
    if (len(sys.argv) < 2):
        print "usage: get package"
```

```
else:
    if (not os.path.exists("cache.pkl")):
        print "cache not found, run update"
    else:
        cache = pickle.load(open("cache.pkl", "rb"))
    if (sys.argv[1].find("moose_") >= 0):
        build_local(sys.argv[1])
    else:
        get(sys.argv[1])
```

Appendix G

Mooselinux Build System: 'Update'

The following python program is part of mooselinux distribution builder.

```
#!/usr/bin/python

import os
import sys
import urllib
import zlib
from gzip import GzipFile
import pickle

def parse_package_file(f, base):                                    10 parse_package_file
    cache = {}
    for line in f.readlines():
        line = line.strip('\n')
        tokens = line.split(' ')
        if (tokens[0] == "Package:"):
            current = tokens[1]
            cache[tokens[1]] = ""
        if (tokens[0] == "Filename:"):
            cache[current] = base + tokens[1]
    return cache                                                    20

def update():                                                       update
    cache = {}
    f = file("sources.list")
    for line in f.readlines():
        line = line.strip('\n')
        tokens = line.split(' ')
        if (len(tokens) < 4):
            continue
        print "type, uri, dist", tokens[0], tokens[1], tokens[2]    30
```

```
        reps = tokens[3:]
        for rep in reps:
            print "fetching " + rep + " . . . "
            urllib.urlretrieve(tokens[1] + "dists/" + tokens[2] + "/" + rep + "/binary-i386/Packages.gz
            f = GzipFile(rep + ".packages.gz", "r")
            cache.update(parse_package_file(f, tokens[1]))
            f.close()
        f = open('cache.pkl', 'wb')
        pickle.dump(cache, f)
        f.close()                                                                    40

if __name__ == "__main__":
    update()
```

Appendix H

Mooselinux Build System: 'Build'

The following bash script is part of mooselinux distribution builder.

```
#!/bin/bash -eux

##############################################################################
#                              ttylinux build script
##############################################################################

#
# directory locations
#
TOPDIR=`pwd`
DISTDIR="$TOPDIR/dist"

PACKAGES="moose_basesystem moose_busybox moose_linux-2.6.20 "
PACKAGES+="bash libc6 libgcc1 libncurses5"

# exit with error message
#
error_exit()
{
    set +x
    echo
    echo "ERROR: $1"
    echo
    exit 1
}

#
# check whether preconditions for build are met
#
check_sanity()
```

```
{
    cd "$TOPDIR"

    [ -e ./build ] || \
    error_exit "you need to be in the directory with build"

    [ "`id -u`" = "0" ] || \
    error_exit "you need to be root for the build to work"
}

build_packages()
{
    for i in $PACKAGES
    do
        ./get $i
    done
}

unpack()
{
    rm -rf "$DISTDIR"
    mkdir "$DISTDIR"
    for i in $PACKAGES
    do
        tar xzf "$TOPDIR/package_cache/$i.tar.gz" -C "$DISTDIR"
    done
}
################################################################################
#                              build sequence
################################################################################

check_sanity
build_packages
unpack
```

Appendix I

Timing Routines for Benchmarking

The following listing shows the implementation of a timer used to benchmark all of the protocol and algorithms in this work.

I.1 Header File

```
#ifndef TIMER_H_
#define TIMER_H_

#include <time.h>
#include <sys/times.h>
#include <math.h>

#define CLK_TCK CLOCKS_PER_SEC*10000

class Timer                                          10
{
public:
    Timer();
    void start();
    void stop();
    virtual ~Timer();
    double getCpuTime();
    double getProcessorTime();
    clock_t getTics();
    clock_t getTicsUp();                             20
private:
    struct tms buffer;
    clock_t tics_up, tics;
    double f, w;
};
```

#endif */*TIMER_H_*/*

I.2 Implementation

```
#include "timer.h"

Timer::Timer() {                                                    Timer::Timer

}

void Timer::start(void) {                                           Timer::start
    tics_up = times(&buffer);
    tics=buffer.tms_utime+buffer.tms_stime;
}                                                                   10

void Timer::stop(void) {                                            Timer::stop
    tics_up = times(&buffer) - tics_up;       // wall time
    tics = buffer.tms_utime + buffer.tms_stime - tics; // combined processor time
}

double Timer::getCpuTime(void) {                                    Timer::getCpuTime
    return (double) tics_up / (double) CLK_TCK;
}
                                                                    20
double Timer::getProcessorTime(void) {                              Timer::getProcessor
    return (double)tics / (double)CLK_TCK;
}

clock_t Timer::getTicsUp() {              // get wall time in tics     Timer::getTicsUp
    return tics_up;
}

clock_t Timer::getTics() {           // get combined processor time in tics   Timer::getTics
    return tics;                                                    30
}

Timer::~Timer() {                                                   Timer::~Timer

}
```

Bibliography

[Xen, 2007] (2007). Xen virualisation. Web. http://www.xen.org.

[Barczak and Johnson, 2006] Barczak, A. and Johnson, M. (2006). A new rapid feature extraction method for computer vision based on momentsa new rapid feature extraction method for computer vision based on moments. In *Proceedings of the Image and Vision Computing IVCNZ2006*, pages 395–400.

[Barczak and Chemudugunta, 2006] Barczak, A. L. C. and Chemudugunta, R. (2006). Experiments with a mobile cluster for real-time object detection. In Mukhopadhyay, S. C. and Gupta, G. S., editors, *3rd International Conference on Autonomous Robots and Agents (ICARA 2006)*, pages 303–308, Palmerston North, NZ.

[Barczak and Dadgostar, 2005] Barczak, A. L. C. and Dadgostar, F. (2005). Real-time hand tracking using a set of cooperative classifiers based on haar-like features. *Research Letters in the Information and Mathematical Sciences*, 5:29–42.

[Barczak et al., 2005a] Barczak, A. L. C., Dadgostar, F., and Johnson, M. J. (2005a). Real-time hand tracking using the viola and jones method. In *SIP 2005*, pages 336–341, Honolulu, HI.

[Barczak et al., 2005b] Barczak, A. L. C., Johnson, M. J., and Messom, C. H. (2005b). A mobile parallel platform for real-time object recognition. In *ENZCon05*, pages 153–158, Auckland, NZ.

[Barczak et al., 2003] Barczak, A. L. C., Messom, C. H., and Johnson, M. J. (2003). Performance characteristics of a cost-effective medium-sized beowulf cluster supercomputer. In *LNCS 2660*, pages 1050–1059. Springer Verlag.

[Bellard, 2007] Bellard, F. (2007). Qemu processor emulator. Web.

[Bianchini et al., 2004] Bianchini, M., Maggini, M., Sarti, L., and Scarselli, F. (2004). Recursive neural networks for object detection. In *Joint Conference on Neural Networks*, volume 3, pages 1911–1915. IEEE International.

[Borman et al., 1999] Borman, D., Deering, S., and Hinden, R. (1999). RFC 2675: IPv6 Jumbograms. Technical report, Network Working Group.

[Buntinas et al.,] Buntinas, D., Dhabaleswar, Panda, K., and Sadayappan, P. Fast nic-based barrier over myrinet/gm.

[CiscoSystems, 2006] CiscoSystems (2006). *Cisco Systems Manual on Internetworking.* Cisco.

[Combs et al., 2004] Combs, H., Gudgin, M., Justice, J., Kakivaya, G., Lindsey, D., Orchard, D., Regnier, A., Schlimmer, J., Simpson, S., Tamura, H., Wright, D., and Wolf, K. (2004). Soap-over-udp. Technical report, Industry Consortium.

[Creel, 2007] Creel, M. (2007). Parallel knoppix. Web. Create a HPC cluster in 5 minutes.

[Crow, 1984] Crow, F. C. (1984). Summed-area tables for texture mapping. *ACM Computer Graphics*, 18(3):207–212.

[Donaldson et al.,] Donaldson, S. R., Hill, J. M. D., and Skillcorn, D. B. Performance results for a reliable low-latency cluster communication protocol.

[Fung and Mann, 2004] Fung, J. and Mann, S. (2004). Using multiple graphics cards as a general purpose parallel computer : Applications to computer vision. In *17th International Conference on Pattern Recognition.*

[Gerdelan et al., 2007] Gerdelan, A. P., Johnson, M., and Messom, C. H. (2007). Performance analysis of virtualized head nodes utilising cost-effective network attached storage.

[Geusebroek and Seinstra, 2005] Geusebroek, J. and Seinstra, F. (2005). Object recognition by a robot dog connected to a wide-area grid system. In *The International Conference on Mutimedia & Expo.* IEEE. www.science.uva.nl/ mark/pub/2005/GeusebroekICME05.pdf.

[Gu and Grossman, 2005] Gu, Y. and Grossman, R. L. (2005). Optimizing udp-based protocol implementations. *Third International Workshop on Protocols for Fast Long-Distance Networks.*

[Hall, 2007] Hall, B. (2007). Beej's guide to network programming using internet sockets. Web. http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html.

[Huang, 2006] Huang, Z. (2006). View-oriented parallel programming. Technical Report OUCS-2006-08, University of Otago. http://vodca.otago.ac.nz/.

[Intel, 2007] Intel (2007). *OpenCV CV Reference Manual.* Intel.

[Kloss, 2008] Kloss, G. (2008). Gaining Colour Stability in Live Image Capturing. In *Proceedings of the 6th New Zealand Computer Science Research Student Conference.*

[Lienhart and Maydt, 2002] Lienhart, R. and Maydt, J. (2002). An extended set of haar-like features for rapid object detection. In *Proceedings of the International Conference on Image Processing*, pages 900–903, Rochester, USA. IEEE.

[Liu et al., 2002] Liu, P. X., Meng, M., Ye, X., and Gu, J. (2002). An udp-based protocol for internet robots. In *Proceedings of the 4th World Congress on Intelligent Control and Automation*, pages 59–65, Shanghai, P.R. China. IEEE, IEEE.

[Majumder, 2004] Majumder, S. (2004). High performance mpi libraries for ethernet. Master's thesis, Rice University.

[Majumder and Rixner, 2004] Majumder, S. and Rixner, S. (2004). Comparing ethernet and myrinet for mpi communication. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7, New York, NY, USA. ACM.

[Pass and Zabih, 1999] Pass, G. and Zabih, R. (1999). Comparing Images Using Joint Histograms. *Journal of Multimedia Systems*, 7(3):234–24.

[Ramachandran et al., 2003] Ramachandran, U., Nikhil, R. S., Rehg, J. M., Angelov, Y., Paul, A., Adhikari, S., Mackenzie, K. M., Harel, N., and Knobe, K. (2003). Stampede: A cluster programming middleware for interactive stream-oriented applications. In *IEEE Transactions on Parallel and Distributed Systems*.

[Schneider, 1982] Schneider, F. B. (1982). Synchronisation in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4:179–195.

[Tanenbaum, 1996] Tanenbaum, A. S. (1996). *Computer Networks*. Prentice Hall; 3rd edition (March 6, 1996).

[Tinetti and Barbieri, 2003] Tinetti, F. G. and Barbieri, A. (2003). An efficient implementation for broadcasting data in parallel applications over ethernet clusters. In *AINA '03: Proceedings of the 17th International Conference on Advanced Information Networking and Applications*, page 593, Washington, DC, USA. IEEE Computer Society.

[van Engelen, 2003] van Engelen, R. A. (2003). Pushing the SOAP Envelope with Web Services for Scientific Computing. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 346–354, Las Vegas, NV.

[Viola and Jones, 2002] Viola, P. and Jones, M. (2002). Robust real-time object detection. *International Journal of Computer Vision - to appear*.

[Vmware, 2007] Vmware (2007). Vmware. Web. http://www.vmware.com.

[Webb, 1993] Webb, J. A. (1993). Latency and bandwidth considerations in parallel robotics image processing. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 230–239, New York, NY, USA. ACM Press.

[Wilkinson and Allen, 1999] Wilkinson, B. and Allen, M. (1999). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall.

[Yang et al., 2002] Yang, M.-H., Kriegman, D. J., and Ahuja, N. (2002). Detecting faces in images: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(1):34–58.