High Performance Computer Vision

Benjamin Eagan

August 23, 2012

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2012
Abstract

This report investigates the use of High Performance Computing resources for real-time computer vision applications through two test cases. The first test case explores parallel hand tracking for use in Human Computer Interaction applications using a Shared Memory pipeline. Results confirmed improved capabilities through the use of high-definition video sources. The second test case investigates if Graph-Cut stereo matching can be performed in real-time using High Performance Computing to produce a disparity map of an environment. Accurate real-time disparity maps were not produced in this report, however results suggest it is both possible and promising.
## Contents

1 Introduction .................................................. 1
   1.1 Computer Vision History .................................. 3
   1.2 High Performance Computing .............................. 3
   1.3 Real-Time Computer Vision ............................... 4
   1.4 Project Goals ............................................ 4
       1.4.1 Hand Tracking ...................................... 4
       1.4.2 Disparity Mapping .................................. 5
   1.5 Report Structure ......................................... 5

2 Background Information ........................................ 6
   2.1 High Performance Computing .............................. 6
       2.1.1 Parallel Performance Metrics ....................... 6
       2.1.2 Performance Limitations ............................. 7
   2.2 Parallel Programming ..................................... 8
       2.2.1 Shared Memory ...................................... 8
       2.2.2 Message Passing Interface .......................... 9
   2.3 Parallel Computer Vision ................................ 9
   2.4 Parallel Real-Time Computer Vision ....................... 10
   2.5 Hand Tracking ........................................... 11
       2.5.1 Hand Tracking Algorithms .......................... 11
       2.5.2 Canny Edge Detection ............................... 11
       2.5.3 Feature Tracking .................................... 12
       2.5.4 Continuously Adaptive Mean Shift .................. 13
   2.6 Disparity Mapping ......................................... 14
       2.6.1 Stereo Calibration and Rectification ............... 16
       2.6.2 Fish-Eye Calibration ................................. 18
   2.7 Disparity Mapping Algorithms ............................ 19
   2.8 Summary .................................................. 21

3 Algorithm Selection .......................................... 22
   3.1 Hand Tracking ............................................ 22
   3.2 Canny Edge Detection .................................... 22
       3.2.1 Feature Tracking .................................... 23
       3.2.2 Continuously Adaptive Mean Shift (Camshift) ....... 24
   3.3 Disparity Mapping ......................................... 24
# List of Figures

2.1 An example of an source image seen left, and the Canny Edge detection output pictured right. ................................................................. 12
2.2 An example flock of good features to track representing a hand. ..... 13
2.3 A hand moving faster than the camera shutter speed can capture, providing a blurred and stretched representation. ......................... 15
2.4 An example of the Camshift search window seen as the rectangle, the estimated hand location shown as the oval, and the average centre of the image shown with a small circle. ........................................ 15
2.5 A sample stereo image pair from the University of Tsukuba. Left and right images are of the left and right perspective respectively. ...... 17
2.6 An example of a simple stereo video filming rig. .......................... 18
2.7 An example of the point identification for left and right image sources using a chessboard for image calibration. .......................... 18
2.8 This image shows streets of Edinburgh at night, with a fish-eye image shown above, and a standard undistorted lens seen below. .......... 20

3.1 A before and after representation of a three-channel thresholding process to isolate skin colours. .......................................................... 23
3.2 The Block Matching disparity map is seen left, and the Graph Cut disparity is pictured right. These were obtained using the stereo pair seen in Figure 2.5 .......................................................................... 25

4.1 The four stages of the hand tracking pipeline. .............................. 28
4.2 The left image shows a successful hand location estimate, and the right image shows a failure. .............................................................. 29
4.3 A source image is seen above, with a one dimensional row domain decomposition pictured below for four processors. ...................... 31
4.4 A two dimensional domain decomposition for the image seen in Figure 4.3, and a two dimensional domain decomposition with halo data included with red shading ............................................. 32
4.5 An example of the row artefacts occurring for a row-wise image decomposition compared with the source image. ........................... 33
4.6 The parallel behaviour of the domain decomposition disparity mapping parallelisation. ................................................................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The number of successful frames and frames per second for three levels of resolution over four processors.</td>
</tr>
<tr>
<td>5.2</td>
<td>The frames per second calculation rates for varying processor count and resolution.</td>
</tr>
<tr>
<td>5.3</td>
<td>The speedup for three different resolutions and a varying number of processors.</td>
</tr>
<tr>
<td>5.4</td>
<td>The parallel efficiency for three different resolutions and a varying number of processors.</td>
</tr>
<tr>
<td>5.5</td>
<td>Disparity map for Tsukuba Stereo source, with the number indicating how many MPI processes were used for calculation.</td>
</tr>
<tr>
<td>5.6</td>
<td>Disparity map for skydiving image source with the number above describing the number of MPI processes used for calculation.</td>
</tr>
<tr>
<td>5.7</td>
<td>Disparity map outputs for a calibrated 480p 3D camera source stereo pair using different parallel decompositions, with the number of processors denoted by the numbers above each image.</td>
</tr>
<tr>
<td>5.8</td>
<td>The rate of calculated frames per second vs number of processors for two different source image sizes.</td>
</tr>
<tr>
<td>5.9</td>
<td>The speedup for two resolutions of disparity mapping sources for varying number of MPI processes.</td>
</tr>
<tr>
<td>5.10</td>
<td>The parallel efficiency of disparity mapping for various resolutions and processor counts.</td>
</tr>
<tr>
<td>A.1</td>
<td>Dissertation project work schedule.</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Disparity map performance measurements run with the University of RMIT stereo source for multiple resolutions. 43
5.2 CrayPat results with the minimum, maximum, mean, and median processor run-times for 16 core decomposition, along with a high-water memory mark. 46
Acknowledgements

I would like to thank David Henty, Judy Hardy, and all the EPCC staff for organising such an interesting and informative MSc program. I would also like to thank my fellow HPCers for being such a fun, helpful, and encouraging bunch. Most of all I’d like to thank my supervisor Alistair Grant for accepting my project proposal, and helping me keep focused throughout the Dissertation period. Without his help I would have spent the better part of the winter convinced I had chosen an impossible project.
Chapter 1

Introduction

Let us consider the human body’s ability to catch a tennis ball in flight; a task achievable by most humans with little difficulty. Many interesting skills are required for this, the first being the act of recognising the tennis ball itself. This is achieved by observing the environment with our eyes, and processing this visual data to match object concepts stored in our memories. Once we have identified the moving blob of image data as our tennis ball, we begin to assess the distance and speed of the ball. From this information, a prediction can be formed about where the ball will be in a few seconds time using an internal gravity model [1], and we begin to move our hand to this position.

The act of moving our hand to the predicted location requires a combination of muscles to be tightened or loosened in order to swing the lever of our forearm into position. Once the arm is in proper position, the hand must dampen the impact of the ball to provide enough time for our fingers to close around it before it reflects away. There are many steps to this process, but fortunately most of them occur automatically. This whole process is made possible by the vast capabilities of our brains.

Often the field of robotics attempts to mimic human abilities like catching, along with others including object recognition, bipedal walking, and balancing in uneven or moving terrain [2]. In our attempts to re-create these abilities, it becomes clear just how remarkable our brains are.

For comparison, let us imagine the brain as a computer. Crude assumptions about the brain’s computational power can be made using information from our retina. The retina processes approximately 10 million pixels per second using roughly 100 million neurons in a volume of around 0.05 cm$^3$ [3]. A computer analysing a similar number of pixels per second to identify edges and colours would require 1000 Million Instructions Per Second (MIPS) [3]. Now if we take the average brain size of 1500 cm$^3$ [3] and assume it has a computational efficiency similar to the retina, you get a computational speed of roughly $1.0 \times 10^8$ MIPS.

Although these are approximations, it can be confirmed that this is an acceptable estimate through the evaluation of a famous game of chess that occurred in 1997. This
competition was between the chess grandmaster Garry Kasparov, and the IBM computer named Deep Blue. Garry Kasparov lost the match, yet it was widely regarded as a close contest with two evenly matched competitors.

During this contest, Deep Blue was capable of $3 \times 10^6$ MIPS. Previous estimate predicted that Kasparov was capable of $1.0 \times 10^8$ MIPS. If Garry Kasparov applied roughly 1/30th of his brain power entirely to the game of chess, this would put the two opponents on an even computational playing field. As many other tasks were occurring concurrently with the game of chess (such as staying balanced on his chair), 1/30th seems like a reasonable efficiency [3]. This confirms that the estimate of brain performance at $1.0 \times 10^8$ MIPS is at least a feasible approximation.

In the 15 years that have passed since this chess match, the capabilities of computers have grown significantly. Modern supercomputers are capable of over $1 \times 10^9$ MIPS [4]; a figure that surpasses estimated human abilities by a full order of magnitude. One major difference between a brain and a computer at this level is the cost to run. A supercomputer costs much more to run than a human due to large power requirements involved. In fact, between 30-70% of power is required simply for cooling the system, and moving data throughout the system [5].

Currently the fastest supercomputer in the world, Sequoia, is located at the Lawrence Livermore National Laboratory in the United States. At peak performance, this computer requires 7.9MW of power [6]. Estimates for the UK put the cheapest MegaWatt hour (MWhr) at £76.60 which means it would cost around £14,523 to run Sequoia for just 24 hours.

A large portion of Sequoia’s energy goes into tasks other than pure computation, thus we consider a human bodies total energy requirements instead of just the brain. Unlike Sequoia, humans cannot get energy from an electrical grid. Instead it must be supplied in the form of food. Humans are recommended to consume roughly 2000 Calories per day to survive, equal to an average power of 97 Watts. This means that a human could be run for only a few pounds per day, while the daily cost of running a top supercomputer would be more than ten thousand pounds. In terms of raw computing, supercomputers have surpassed the estimated abilities of the human brain. The cost to run these systems however is much higher. This makes it impractical to use supercomputers for applications as basic as catching a ball.

We have established that due to power requirements, it is too expensive to use top tier computer resources to reproduce trivial human brain functions. We can still reproduce individual abilities such as the use of our senses on smaller computing systems. An example of this can be seen with the mobile phone application called Soundhound. This program uses the microphone of the mobile phone to listen to a song, and identifies what song is being played for the user. The process takes only a few seconds [7]. There are also a number of capabilities using cameras and computers to mimic the use of our eyes. This is a process called computer vision.
1.1 Computer Vision History

Computer vision is broadly defined as the use of visual information from photographs or videos to provide a decision or new representation [8]. This has a number of interesting applications including Human Computer Interactions (HCI), autonomous vehicle navigation, industrial fault detection and security and surveillance. As technology matures, the diversity of computer vision applications continues to grow.

The building blocks towards this technology were first introduced by Leonardo Da Vinci in the 1500s. He realised that the world could be represented as a single planar two-dimensional representation of a three-dimensional world. Additionally, he observed that this was highly variable based on perspective [9]. These two concepts are very important for a number of computer vision applications, including disparity mapping, discussed in detail throughout this report.

Several hundred years later, the first computer vision program was written by Lawrence Roberts [10]. Taking place in 1965, this work involved viewing simple shapes from various perspectives to create three-dimensional ray-tracings of the objects. Through this work Roberts is credited as the father of computer vision. Interestingly this is often overlooked when discussing his career, as he is also considered a father of the internet [11].

Between 1965 and present day, the field of computer vision has matured a great deal. This is largely due to work by Intel, who identified computer vision a computationally intensive field of computer science. In an effort to create and maintain demand for faster processors, they began to develop a freely available computer vision library in 1999. This was built upon a number of smaller university frameworks created independently throughout the United States, and grew to what is now known as Open Computer Vision, or OpenCV. After its release in 2006, the user-base has taken responsibility for developing this code with over 20000 active developers, and over two-million downloads [8].

1.2 High Performance Computing

High Performance Computing (HPC) is a term used to describe the use of cutting edge computing resources on all scales, which includes mobile applications, clusters, accelerators, and top tier supercomputers. HPC is used largely for simulating physical systems, with a large user base from the scientific community as well as engineering, and finance. The field of HPC is constantly evolving, and the performance of the top 500 computers is published once every six months. These results serve as a good indication of what smaller scale systems will be running in only a few years time [6].

A major feature of HPC systems is their use of parallel computing resources. One technique for reaching faster computational speeds is to run the Central Processing Unit
(CPU, also known as a processor) at higher voltages to provide a higher frequency of computation. As the frequency is proportional to the square of the voltage, to double the CPU speed four times more voltage is required. This also results in increased heat due to a process called Joule Heating, and additional energy for heat dissipation is required [12]. Computation on a single processor is called serial computing.

Due to the power and heating problems that occur when trying to increase the speed of a serial program, the use of parallel processing has been adopted to achieve faster computation. With parallel processing, instead of doubling the speed of a single CPU, two CPUs would work together to complete the same task. This keeps power and heat dissipation requirements low, while providing more computational resources. This strategy can be applied for systems with far more than two processors, as there are currently systems with over one million cores [6].

### 1.3 Real-Time Computer Vision

There is a sub-set of computer vision which takes place in real-time, making it particularly interesting for use with HPC due to a unique set of requirements. Real-time computer vision deals with live decision-making typically on the time-scale of between 10 frames per second and 100 frames per second based on the specific use case [13] [14]. Real-time applications do not have the luxury of waiting for an answer to arrive, so to keep computation times within a usable range, image resolution and algorithm complexity must be limited. The use of HPC systems for real-time computer vision could eliminate these restrictions by providing far more computational power in these fixed time frames.

### 1.4 Project Goals

In this report, two test cases will be explored to identify if HPC and parallel programming techniques can be used to improve real-time computer vision applications. This will investigate the use of HPC techniques for increased algorithm complexities, faster results, and higher resolution input images to identify the possible benefits from each approach. Additionally, the parallel behaviour for a variety of architectures will be explored.

#### 1.4.1 Hand Tracking

In this test case a parallel real-time hand tracking program will be explored. Hand-tracking is widely used in HCI where researchers are attempting to replace or complement traditional user interfaces such as laptops and keyboards with natural hand gestures. This technology can be seen in recent video gaming systems with the PlayStation
3 Move and the Xbox 360 Kinect. These use a combination of sensors including a visual spectrum video feed to allow for the user to control game actions with the movements of their body [15].

The Kinect system uses an infrared depth sensor and a visible spectrum camera to identify individuals and gestures [15]. This hand tracking application will attempt to perform a similar task to these, however it will do so using only a single camera source and the use of parallel computing resources.

### 1.4.2 Disparity Mapping

This test case investigates the use of stereo image pairs as seen to extract depth information of their image content in a process called disparity mapping. The term disparity describes the displacement of object locations between one image to the next, and is used to calculated the depth of objects in the two images. This allows for three-dimensional mapping of an environment, and is helpful for navigation purposes.

Current disparity mapping techniques are computationally expensive, and don’t provide sufficient real-time information to serve as a primary sensor for navigation. The use of HPC may reduce the run time for accurate disparity maps to those suitable for real-time applications. Since high resolution cameras are affordable in comparison to laser and ultrasonic depth sensors which cost upwards of £6000 [16], parallel stereo vision disparity mapping could become a low cost and high accuracy solution for navigation.

### 1.5 Report Structure

This report begins by introducing the background concepts required for a basic understanding of hand tracking and disparity mapping. Once this has been established, a description of the algorithm selection process and criteria for each test case will be provided. Following this, there will be a description of the implementation of the algorithms as well as the parallel hardware used for these tests.

Next, the results from these tests will be discussed using terms from the Background Information chapter. Finally, concluding statements will be presented for each of these applications, including a discussion on the suitability of High Performance Computing techniques for use with computer vision.

Each section provides a summary highlighting the essential terms and concepts for reference throughout the report. A glossary of terms can also be found in the Appendix.
Chapter 2

Background Information

2.1 High Performance Computing

2.1.1 Parallel Performance Metrics

The goal for parallel programming is to make use of different methods of work sharing across multiple processing elements to allow for more work to be done, or the same amount of work to be done faster. When discussing parallel programming, there are several terms and theories which are useful for understanding the behaviour of a program as it scales across a varying number of processors.

Ideally, if four processors are used on a job, this should provide four times faster performance than in the serial case. This relationship is described by the following term called speedup where

\[
\text{Speedup}(N) = \frac{T_{\text{serial}}}{T_{\text{parallel}}},
\]

(2.1)

In this relationship, \( N \) represents the number of processors, and \( T_{\text{serial}} \) and \( T_{\text{parallel}} \) describes the time taken to run the job in serial, and in parallel over \( N \) processors respectively. When parallelisation has been implemented effectively, the speedup value should be close to the number of processors used. Sometimes algorithmic limitations prevent this from occurring.

It is possible to get speedup values that are greater than the number of processors used, called super-linear speedup. This occurs when the total cache size of all processors becomes larger than the problem size being used [17].

From Equation 2.1 we see that the speedup value is a function of the number of processors involved. To remove this dependency, a metric called the parallel efficiency is used to show how effectively a problem has scaled independent of \( N \). This is calculated as
Efficiency = \frac{T_{\text{serial}}}{N_{\text{processors}} \times T_{\text{parallel}}} 

= \frac{\text{Speedup}(N)}{N_{\text{processors}}}

(2.2)

where \( N \) once again represents the number of processors, \( T_{\text{serial}} \) and \( T_{\text{parallel}} \) describe serial and parallel run times, and \( \text{Speedup}(N) \) provides the speedup value for a given \( N \).

If your goal is to run an existing program with a larger set of data using parallel resources, this is referred to as weak scaling. If your aim is to use parallel resources to make a fixed sized problem run faster, this is strong scaling.

### 2.1.2 Performance Limitations

Parallelising a program requires that there are sections of the code which can be done independently. This means that a single processing element must be able to do a subset of the total work with no information from the other participating processors. At some point, communication with neighbouring processors is necessary to either arrive at a complete final answer, or begin the next iteration with the correct information.

These communication events often result in a region which must be computed in a specific order, also called a serial region. Frequently serial regions exist in initialisation and finalisation stages of a program.

Consider a program that takes 80s to run on a single processor. If 20s of that must be executed in serial, and the other 60s parallelise with perfect scalability, how long would it take to run on two processors? If the 60s region scales perfectly, this will take 30s now, however the 20s serial region remains. This means with two processors, the total run time would be 50s, achieving a parallel efficiency of 80%.

Now consider running the same job with 10 processors. Once again, the parallel region scales perfectly with a new run time of 6s, and the serial region remains the same with 20s, leading to a total run time of 26s. This is a parallel efficiency of just 30.8%. On 100 processors it would take 20.6s with an efficiency of just 3.8%. With an infinite number of processors, the fastest it could possibly run would be a small fraction above 20s.

The diminishing performance seen here is described by the equation

\[
\text{Speedup} = \frac{N}{(\alpha \times N) + (1 - \alpha)}
\]

(2.3)

where \( N \) is the number of processors, and \( \alpha \) is the percentage of serial code in the program. This is called Amdahl’s law [18], and describes the maximum performance...
of an program with a fixed problem size. From this law we see that for very large $N$, the speedup simply becomes $1/\alpha$.

While this has been presented as a discouraging relationship, it deals only with fixed problem sizes, also known as strong scaling. This leaves a very useful avenue for parallelisation where the problem size is increased as processors are added, which describes weak scaling.

If we return to our sample problem with 20s of serial computation and 60s parallel, we considered the result using 10 processors to provide 30.8% parallel efficiency. If we increase the amount of parallel work from 60s on a single processor to 600s (a value $N$ times greater), we would arrive at a parallel run time of 80s and a parallel efficiency of 77.5%. This response to Amdahl’s scaling limitations was formalised by John Gustafson, and is referred to as Gustafson’s Law. Due to these laws, weak scaling applications are much more common than strong scaling in HPC [18].

### 2.2 Parallel Programming

When considering how to parallelise the hand tracking and disparity mapping computer vision applications, two different approaches were considered. These were Shared Memory programming, and Message Passing programming. The strengths and weaknesses of these techniques with respect to each application will be explored in this section.

#### 2.2.1 Shared Memory

Shared Memory programming is used for Single Instruction Multiple Data (SIMD) programming on architectures with a shared cache. Processing elements in this model are referred to as threads, and communication between threads takes place through the use of shared variables. The prevalent technique for shared memory programming through an Application Programming Interface (API) called Open Multiprocessing (OpenMP) [19]. Shared Memory programming with OpenMP can be introduced to a serial program simply by adding shared variables and using pragmas to add regions of parallelism. This is a straightforward approach to parallelisation, as it can be done gradually without breaking serial functionality.

Many modern processors with multi-cores can make use of Shared Memory parallelism, however using a large number of processors can be difficult and slow due to cache coherency requirements. In a Shared Memory system, each processor has its own private cache as well as a shared cache amongst all processors. When a single processor updates a shared variable used in the shared cache, this must be updated for all processors. As the number of core grows, the amount of time required to keep the cache coherent becomes large, and parallelisation is no longer effective. The largest Shared Memory systems only contain around 100 cores [19].
2.2.2 Message Passing Interface

Message Passing programming is also used for SIMD programming. When using MPI, data is private for each processor. Any results or information must be sent in the form of messages to other processors running the same program. The primary technique for Message Passing programming is the use of an API called Message Passing Interface (MPI). This does not have the same scaling limitations as Shared Memory systems, as there is no issue of cache coherency. Message Passing programming serves as the primary programming model for massively parallel systems with hundreds of thousands of cores.

While Message Passing is much more scalable than shared memory, it is more difficult to implement. Parallelisation with MPI cannot be simply added as an afterthought to a serial program, because the sending and receiving of data must be handled explicitly. This can be done as point-to-point communication between individual processors, or global communications where all processors send or receive simultaneously.

2.3 Parallel Computer Vision

It was observed by Intel in the late 1990’s that computer vision is computationally demanding, making it an interesting area of research for a company looking to sell faster processors. This demand on the processor is due to the number of mathematical operations that must be applied for image processing and analysis, and the fact that the input images tend to contain a large amount of information in the form of dense matrices. As such, computer vision presents an excellent opportunity for parallelisation and HPC. If parallel resources and programming techniques can be successfully exploited, this would allow higher resolution source images to be used, more demanding algorithms, or faster results.

Fortunately, computer vision lends itself very well to parallelisation. This is because images are represented by either a two or three-dimensional matrix based on the number of image channels, and in many computer vision tasks the matrix elements of the are entirely independent of their neighbouring values. When matrix elements are all independent, parallelisation is simple. This is because the image can simply be divided into many smaller sub-domains of the original image, and these sub-domains are then sent to individual processors to perform the desired computer vision task. The result is then returned to the source processor to be reconstructed. This method of parallelisation is called a domain decomposition and will be discussed further in the following section.

Sometimes however, the individual matrix elements are not independent and a simple domain decomposition can not be used. For example, if an edge detection algorithm discovers edges in a neighbouring element, the probability of an edge existing in your own cell is increased. This information allows more accurate interpretations of the
image data, especially if there is distortion or noise in the source image. If a simple domain decomposition technique were to be applied to an edge detection algorithm, there would likely be edge discontinuities at subdomain boundaries. These discontinuities would result in edge information between sub-domains being lost.

This problem can be solved either through a new algorithm design which incorporates sharing of edge info, or by choosing a different parallelisation technique. In this case, an alternative parallelisation technique would be what is called task parallelism. Task parallelism assigns each individual processor a single task to perform, and information is sent between processors based on what operation needs to be performed. When the order of execution for these tasks is fixed, this design pattern is referred to as a pipeline. The scalability of this parallelisation scheme is limited by the number of unique tasks that exist in the program.

Parallel computing is not new for the field of computer vision, as lots of early computer vision work was done using parallel vector computers [20]. As processor speeds increased over the past few decades, applications that once required a vector supercomputer could be performed on a single processor, triggering a shift away from parallel computer vision. Modern processors are becoming increasingly parallel as seen with the Intel Xeon Phi and AMD Bulldozer containing 50+ and 16 cores respectively. Thus, a return to parallel computer vision will be necessary to make use of available resources [21].

2.4 Parallel Real-Time Computer Vision

Due to the real-time computer vision requirement that computation occur in only fractions of a second, it is a particularly interesting application for parallelisation. Currently algorithms are limited in image resolution and image complexity to compensate for this restriction. If parallel resources could be used effectively, this would relax or eliminate this requirement allowing for more complex computation and high-definition input sources.

An interesting example of a real-time computer vision application is the autonomous vehicle named "Stanley" which was the winner of DARPA’s Grand Challenge in 2005. This challenge required an autonomous vehicle to navigate a 142 mile off-road course in under 10 hours with no previous information of the route. For this system, navigation was performed using a combination of laser and computer vision sensors. The lasers provided accurate short range information for distances up to 22m, however it was necessary to look 70m ahead to provide sufficient information to sustain Stanley’s desired speed of 35mph. This was achieved by using machine learning algorithms to determine the colour of the safe driving path. Learning occurred using the short range information which had been identified as safe from laser data. This process was ongoing which allowed for rapid adaptation to changing conditions such as road surfaces and lighting conditions [22].
Perhaps with parallel algorithms, the on-board cameras would be able to detect the driving path for distances greater than 70m which would allow for faster speeds. Additionally with the capability to process higher resolutions in real-time, the same cameras providing long range information may be capable of providing the short range information as well. This would allow the expensive laser technology \cite{16} to be replaced, or allow for greater driving speeds for the vehicle.

2.5 Hand Tracking

Sometimes humans are born without the use of hearing, or lose their hearing for various medical reasons. When this occurs, the individual is left isolated from the use of verbal communication. In response to this, sign language has been developed as an alternative form of communication. Sign language creates a vocabulary much like spoken languages out of gestures instead of sounds, and can be used easily and naturally once learned. The ease of adoption for this language indicates that our hands are natural tools for gesturing and communication. This can be confirmed by watching good public speakers, who tend to make use of hand gestures frequently for emphasis.

Due to our natural tendency to gesture with our hands, along with their dexterity allowing for many different shapes and gestures, the use of hands has been widely studied for HCI applications \cite{23} \cite{24} \cite{25}.

To use hand gestures as a user interface, first the hand location and orientation must be identified in each new frame. This is called hand tracking. Hand tracking is a difficult problem, as the tracking environment can contain objects with similar size or colour to the hand, along with changing lighting conditions and possible camera motion. Additionally, hands can form many different shapes, making it difficult to initially locate the hand.

There are a number of different techniques for accomplishing hand tracking which all share the same basic formula: choose a unique feature of the hand to track, and search each new frame for this signature. With these techniques the level of information passed from one iteration to the next varies between implementations. When information is passed from previous frames, it typically contains the previous hand location or gesture. The changes from frame to frame are small, so this information can be used to improve estimates by considering minimal changes to be the most probable.

2.5.1 Hand Tracking Algorithms

2.5.2 Canny Edge Detection

The Canny Edge Detection method is a lightweight technique for isolating connected contours through grouping areas based on colour value similarities. If there is a dramatic
local change in the Red Green Blue (RGB) image value, then this is identified as an edge. Interface locations between the differing colours are solved for the image, then displayed as an output of just edges. The output from this algorithm can be seen in Figure 2.1 which identifies the outline of the head and shoulders of a subject. Note the pattern appearing on the shoulders showing some rectangular holes through the upper body. This was caused by a red and white stripe pattern on the shirt, which strongly matches the edge criteria of distinct local colour changes.

The edge detection algorithm can be improved through some image pre-processing techniques such as smoothing and blurring the image, which reduces image noise and makes the average colour value between pixels more similar. This pixel averaging makes connected contours easier to locate. The noise reduction decreases the likelihood of identifying false edge locations [8].

Figure 2.1: An example of an source image seen left, and the Canny Edge detection output pictured right.

### 2.5.3 Feature Tracking

Feature Tracking makes use of image information which is unique and easily recognisable between different image frames. The first step is to identify good features to track. These features are selected by finding corners in the image data [26], as well as the lowest eigenvalue for a collection of possible corner candidates in the same location [8].

While this sounds complicated, it is simply a formal way of choosing to track unique objects. This ensures that instead of tracking a white pixel in the middle of a white wall, small coloured regions or intersecting edges are chosen.

It is also necessary that these features exist on the actual object being tracked. This can be done using a specified learning region in which the hand is placed, or automatic hand detection using machine learning can be performed if a model has been provided.

When tracking a hand, it can be expected to have a trackable feature between each of the fingers, and at the finger tips. This collection of points is referred to as a flock.
Figure 2.2: An example flock of good features to track representing a hand.

[27], which can be seen in Figure 2.2. This flock of points serves as the signature for identifying a hand. It is important to find a group of features to track that are well spread out across the object in question, as this ensures a good representation of its true location is obtained.

Feature tracking is more computationally demanding than the edge detection algorithm, but the added algorithm complexity also adds robustness to the solution for complicated scenes. Often this method is combined with a three-dimensional hand model, and the points in the flock are assigned to the corresponding finger or hand of the model. This allows for gesture recognition by tracking the location of individual points and comparing them with the allowed movements of the hand model [8]. This is a common extension to hand tracking.

### 2.5.4 Continuously Adaptive Mean Shift

The final method explored for hand tracking was called the Continuously Adaptive Mean-Shift algorithm (Camshift). This technique is an extension to what is called the Mean-Shift Algorithm. The Mean-Shift algorithm works by finding a single channel colour frequency distribution called a histogram. An RGB image consists of a red, green, and blue image channel by definition, only one of these would be used to create the histogram. When working with RGB, all three channels are dependent on lighting intensity making it sensitive to varying light conditions. In comparison, the hue, saturation, and value colour space (HSV) places all the lighting information in the value channel. This makes the hue and saturation channels good candidates for hand tracking. Ultimately the hue channel is selected, as it is better suited for recognising skin colours [24].
When an adequate channel has been isolated, a histogram is calculated from this channel. This histogram is used to find what is called a back projection. This provides the probability of a match for each pixel with of the source histogram [8].

The core of the algorithm is then executed, from which this technique takes its name. This first identifies a search window, and scans the back projection within this for a weighted average location of all probability values. The window is repositioned over this location, and the new weighted average of probabilities is found once more. This is repeated until the window stops moving, or the maximum number of iterations is reached [8]. The location of the search window in the previous frame is used as a starting search location with each new iteration.

When this searching process is performed using a window with a fixed sized it is referred to as Mean-Shift. When the search window is permitted to adapt its size it becomes known as the Continuously Adaptive Mean-Shift Algorithm (Camshift). This allows for the hand to change its distance from the camera, as the search window will utilise larger windows when the hand is near to the camera, and small windows when it is distant. An example result from Camshift tracking can be seen in Figure 2.4.

The use of previous hand locations between subsequent frames, along with the probability searching technique used in the Camshift algorithm results in a robust hand tracking application. This technique is capable of tracking the hand in challenging situations containing colourful backgrounds and subtly changing lighting conditions, and it is a comparatively lightweight algorithm computationally.

This algorithm is not without faults, as there are several cases where it is prone to failure. One of these arises when dealing with fast moving objects and low frame rates. In this scenario, a hand appears in a blurred and stretched representation as seen in Figure 2.3. This results in a weak probability match with the original histogram. When motion has slowed and the hand is fully recognisable once again, it is located too far from the search window to be found. This makes it very unlikely that the Mean-Shift search window will correctly locate the hand location.

Another difficulty occurs when a hand travels a significant distance into the image background compared to the distance for the histogram image. While Camshift can adapt to some change in depth, if the distance becomes too great the hand is only represented by a few pixels. This provides a weak probability match, and can be easily confused with other background information.

## 2.6 Disparity Mapping

Self-driving vehicles are currently an active area of research. One of the most important details in this work is identifying safe driving paths. There are a number of various sensors used for this purpose, and often a combination of many sensors is selected. One method for obtaining depth information is through the use of disparity mapping, however finding accurate depth maps is a slow process when using a single processor.
Figure 2.3: A hand moving faster than the camera shutter speed can capture, providing a blurred and stretched representation.

Figure 2.4: An example of the Camshift search window seen as the rectangle, the estimated hand location shown as the oval, and the average centre of the image shown with a small circle.
The basic principle behind disparity mapping can be understood simply by holding a finger up about 10 cm in-front of your face, and closing your left eye. Focus on your finger with your right eye, then close your right eye and quickly open your left. Alternate back and forth between open eyes, and you will notice the apparent location of your finger shifting from left to right. The displacement of the finger between your left eye and right eye images is called its disparity, and your brain interprets this to add depth information to your environment.

If you move your finger to around 20 cm from your face and repeat this process, you’ll notice the apparent shift in finger location between left and right eyes is reduced. This shows that the disparity is depth dependent, and inversely proportional to distance from the image source.

These depth calculations are handled intuitively by our brains, but they can be solved mathematically for computer vision purposes. This is done using the relationship

\[
Z(d) = \frac{fB}{d},
\]

where \(Z\) is depth, \(d\) represents the disparity, \(f\) is the camera focal length, and \(B\) is the baseline [28]. Baseline is a term to describe the distance between the two camera sources, and it is this value that provides a unit of measurement for the depth values.

Disparity mapping is achieved in computer vision by first identifying matching pixels between two images such as the pair seen in Figure 2.5, and measuring the disparity in pixel location between image and the other. If images are the same size, this can be done by setting one image as fixed, and measuring the coordinate offset of the corresponding points in the other image. With these values known, Equation 2.4 can then be used to extract the true depth measurement of the object.

In this project, the use of stereo vision disparity mapping to obtain depth information about an environment will be explored. There is also another interesting computer vision technique that makes use of monocular disparities. If two images from the same source are taken with a very small amount of time occurring between images, the disparity between image outlines reveals if any motion toward or away from the camera has occurred. This information can be used to calculate velocities [29].

### 2.6.1 Stereo Calibration and Rectification

It can be seen that Equation 2.4 deals with disparities occurring in only one dimension, however our image sources are two-dimensional. While it is possibly to modify this equation to work in two-dimensions, it is simpler to pre-process the images such that disparities only exist in one dimension. This pre-processing ensures that the two images are aligned along a single axis. If they are aligned horizontally for example, disparities will only occur horizontally. This alignment is chosen to be parallel to the baseline of the stereo image pair.
In this experiment a left and right image pair was used as a stereo source, with the capturing device pictured in Figure 2.6. As the baseline separation between these cameras was a horizontal displacement, a row-aligned configuration was used to eliminate vertical disparities. This process is called rectification. Additionally due to imperfections in digital imaging and limits in manufacturing precision, it was necessary to transform the two images such that they exist on the same image plane. This is referred to as stereo calibration [8].

It is simpler to rectify the images using a calibrated image pair, so the calibration step takes place first. This calibration process results in a rotation and translation matrix which can be multiplied with one of the images. This provides a new image representation which is aligned on the same plane as its stereo image counterpart.

To obtain these values, first a set of images with easily identifiable features are obtained. A common choice for this is a chessboard, as the black and white squares provide easily locatable corners as can be seen in Figure 2.7. The chessboard provides a coordinate system of rows and columns which allows for the identification of specific corners. Rectangular chessboard patterns are typically used for this, which ensures a unique orientation for the board is detected. In Figure 2.7, it can be seen that the board has 7 rows and 10 columns.

The rotation and translation matrices can be calculated using two sets of points for each image. These point sets contain the location of the chessboard corners, in both the image’s x-y coordinate space as well as the chessboard’s row-column coordinate space. This information is sufficient to determine the appropriate rotation and translation matrices, which are then applied to one of the images.

At this point, the image pair should be very nearly aligned on an image plane. However due to distortions in the lens or camera imaging chip mounting, a set of distortion parameters must also be found for both the left and right images. This is taken care of during the image rectification. The rectification process uses the same set of image points as the calibration to provide a mapping for both images so that they are row-
aligned and distortion free (relative to one another). In this process there are a number of possible solutions. The final rectification aims to identify the solution that maximises image overlap, and minimises distortion coefficients.

### 2.6.2 Fish-Eye Calibration

The simplest form of camera operates using just a tiny hole in side of a dark box, and a strip of film on the opposing wall. The image data passes through the hole and arrives inverted on the back wall of the box. This is referred to as a pin-hole camera, and serves as the camera model for the calibration process described previously. While this holds for most cameras, there is a type of lens called a "fish-eye" that breaks these assumptions. A fish-eye lens is used to display a much larger field of view at the cost of increased distortion around the outer regions of the image. This effect can be seen in
Figure 2.8, where both the distortion and added image information can be observed.

The disparity mapping and calibration techniques assume that the images can be approximated as if taken using a pin-hole camera [8]. As the true projection of a fish-eye image however is onto a sphere, the planar projection model does not hold. As a result, further calibration for this type of camera is required.

This calibration can be performed by pre-processing the fish-eye data to determine the size of the sphere which the camera projects onto. Once this projection has been obtained, the previous calibration and rectification steps can be performed. It may seem as if the calibration step should have been completed by projecting the sphere on to a plane, but this simply created a planar representation. It is still necessary to perform the calibration step to ensure the two images are also planar-aligned. [30].

This extra work is worthwhile due to the benefits arising from a larger field of vision. This allows cameras to capture far more information within a short range of the camera, so for purposes such as measuring distance above the ground for vehicles, or depth mapping very small spaces, fish-eyes allow much more environmental data to be represented.

2.7 Disparity Mapping Algorithms

With the source images properly calibrated, it is time to begin the disparity mapping process. There are a number of different techniques for finding disparity maps, however the two most common techniques are Block Matching, and Graph Cut Matching.

Block Matching is a popular algorithm as it is both simple and computationally efficient. This method defines a block in one image containing a subset of pixels, and attempts to find the closest match in the adjacent image. There are a variety of matching criterion that can be used in this comparison, including Sum of Absolute Differences, Sum of Squared Differences, and Normalised Sum of Squared Differences [31]. This searching technique performs well when dealing with highly texturised surfaces, however for smooth objects like tables or water it can be difficult to find matches between images [8] [31].

The Graph Cut Matching method is more computationally demanding than Block Matching. It also delivers much more accurate results. Graph Cut Matching is achieved by identifying a number of possible disparity configurations for the two images, and uses graph cuts to minimise an energy function representing the likeliness of a correct match [32]. This is achieved by identifying several possible configurations $C_i$ containing a set of possible pixel pairs. These configurations are assessed by scanning for possible pixel matches, and giving each configuration an overall energy measurement $E_{total}$. This is based on the sum of energies

$$E_{total}(C_i) = E_{data}(C_i) + E_{smooth}(C_i) + E_{occ}(C_i),$$  (2.5)
Figure 2.8: This image shows streets of Edinburgh at night, with a fish-eye image shown above, and a standard undistorted lens seen below.
where the lowest energy solution is selected to be the best configuration. In Equation 2.5, the $E_{data}$ term represents how well the intensities of matching pixels correspond, $E_{smooth}$ provides how well disparity measurements compare with neighbouring elements, and $E_{occ}$ describes the energy associated with occlusion placements. An occlusion is when an object appears between another object in the distance and the camera, which can be complex for computer vision algorithms to handle [32].

### 2.8 Summary

In this section a number of parallel computing concepts were introduced including speedup, and parallel efficiency. The Shared Memory and Message Passing programming techniques were discussed, along with their possible applications to computer vision problems. In this discussion real-time computer vision was revealed to be a strong candidate for improvements from programming techniques. Finally, a number of methods for hand tracking and disparity mapping were introduced, and their strengths and weaknesses were explored.
Chapter 3

Algorithm Selection

3.1 Hand Tracking

In this section, three different algorithms are explored to determine which is the best fit for this experiment. These algorithms are Canny Edge Detection, Feature Tracking, and Continuously Adaptive Mean-Shift Tracking (Camshift). These were all introduced in the previous section, however it’s important to choose the algorithm for hand tracking carefully to ensure parallel performance results are useful. This is because it would be simple to find a hand tracking algorithm that improves with parallel programming techniques. This work would be unnecessary however if an alternative hand tracking technique performed just as well in serial. Thus, the algorithm with both the best performance and minimal computational cost will be the chosen candidate for these experiments.

3.2 Canny Edge Detection

The Canny Edge Detection hand tracking algorithm creates a new image composed of only edge information as was seen in Figure 2.1. From this edge information, closed contours were identified and scanned to locate a hand shape. Pre-processing was also applied to simplify the image such that it contains only colours likely to represent a hand.

This preprocessing was done using a three channel threshold function to remove the areas of the image outside a specified colour range as can be seen in Figure 3.1. This thresholding was calibrated to skin colour using a graphical user interface with a sample image and sliders to adjust colour range. Once the correct threshold values had been identified, these were saved for the main programs execution. This process must be completed for each new subject or environment.

This is a lightweight algorithm, however it lacks robustness when a colourful or noisy
background is present. The success of the edge detection depends very strongly on the pre-processing steps of colour averaging and skin colour isolation. The skin colour isolation is also a source of error, as it requires lighting conditions to remain constant to achieve success. Most webcams and digital cameras make use of automatic white balance detection, meaning they adjust the tone of the image under varying lighting conditions such that white objects appear white. If this value changes automatically during hand tracking, the skin isolation process would no longer perform successfully. Finally, even with constant lighting, textured and colourful backgrounds can be difficult to filter. This prevents the correct location of the hand from being identified.

3.2.1 Feature Tracking

Feature Tracking attempts to identify the hand not by colour, but rather by easily trackable points. As described previously, a set of these points called a flock are detected on the hand before tracking can begin. This flock of points is used to identify a hand location, and can be tied to a three-dimensional hand model to identify gestures.

This method was seen to be have more successful results than the Canny Edge Detection, however it was challenging to find a strong number of points and differentiating them between frames without a hand model prone to failure.

Creating a hand model would solve this problem, however this involves extensive machine learning, requiring a database of thousands of sample images. Additionally, a goal for the project had been to develop a portable and robust technique. Feature Tracking lacks subject robustness, as it would require a new model for hands that suffered wounds such as a missing finger, and may not work well for children’s hands if using an adult model.
3.2.2 Continuously Adaptive Mean Shift (Camshift)

The Camshift algorithm is a lightweight approach to hand tracking, and similar computation demands to the Edge Detection technique. This technique provided far more robust tracking in challenging situations than the edge detection method, with similar performance to Feature Tracking with reduced computational cost. This is largely due to the choice of hand signature used in the Camshift algorithm. In the Edge Detection and Feature Tracking techniques, the entire input image is scanned for the identifiable feature of edges and corners to identify a hand. The Camshift method uses a probability representation in the form of a back projection, and the peak probability region is identified. Additionally, it makes use of the location of previous frames where the other methods do not.

For these characteristics, the Camshift algorithm provided more robust hand tracking in challenging environments than both the Edge Detection and Feature Tracking approaches. This algorithm can be improved using skin colour isolation techniques learned from the Edge Detection method.

As Camshift offered both robust hand tracking, and high computational efficiencies, it was identified as the best algorithm for use in hand tracking experiments.

3.3 Disparity Mapping

Both Block Matching and Graph-Cut Matching algorithms were considered for disparity mapping in this experiment. As the purpose of this test case was to find the highest accuracy disparity map and attempt to parallelise it for real-time performance, the algorithm selection criterion is simple. The algorithm which provided the most accurate disparity map would be selected for use in this experiment.

A comparison between the Block Matching and Graph Cut stereo matching algorithms can be seen in Figure 3.2 of disparity maps from the Tsukuba stereo image pair seen in Figure 2.5. From these results, it is clear that the Graph Cut algorithm yielded more accuracy, with multiple layers of depth identified instead of simply the layer overlap. This can be attributed to the Graph Cut method’s ability to recognise objects without requiring strong texturisation [32].

It was also seen that the Graph Cut technique took much more time to calculate than Block Matching, which makes it a particularly well suited candidate for gains from parallelisation. If Graph-Cut disparity mapping could be achieved for a set of images using only a fraction of a second, the implications for navigation could be quite large.
3.4 Summary

In this section, the different algorithms from the Background Information chapter were considered for use in these experiments. The Camshift algorithm was identified as the best choice for hand tracking due to its robust abilities and relatively low computational cost. The Graph Cut disparity mapping technique was selected based on its ability to create higher accuracy disparity maps than the Block Match technique.
Chapter 4

Implementation and Experimental Design

In the hand tracking and disparity mapping applications, the computer vision library called OpenCV was used to provide the image processing and decision making functions. Due to a bug in the latest OpenCV version 2.4.2, the previous OpenCV version 2.3.1 was used. A multi-media framework with support for Unix, Windows, and Mac OSX called ffmpeg was also used. This allowed for image encoding and decoding when interpreting the source image or video files, and storing the output in a similar format.

4.1 Hand Tracking

4.1.1 Parallel Implementation

From the three candidates the Camshift algorithm was selected as the best method for hand tracking. This algorithm used a global search of the input image for identifying hand locations, meaning it could not be parallelised by simply dividing the image into sub-domains with a hand search in each smaller window. If this were attempted, the most probable location for each sub-image would be returned, which would be meaningless unless the entire hand was located in single sub-domain.

To remedy this issue, a pipeline approach to parallelisation was selected which would be implemented using OpenMP and Shared Memory programming. With this technique, each thread was assigned a single task to perform over each iteration. The input image would be passed between threads in a specific order of steps, passing relevant information to each new thread. Once the image data had been sent to each thread, the final result would be saved to disk. The number of steps in this program must remain fixed, however the division of steps between threads could be varied.

The steps required for the main function of the hand tracking application are as follows:
1. Capture source image.
2. Convert to hue saturation and Vblue (HSV) colour space.
3. Separate the hue channel.
4. Calculate the back-projection of hue using the hand Histogram.
5. Calculate an image mask using the three-channel thresholding seen in Fig. 3.1.
6. Apply a logical "And" between the back-projection and mask, deleting all image data outside the threshold mask.
7. Perform the Camshift hand tracking algorithm using the source image and back-projection.
8. Draw circle showing hand location onto image.
9. Output image to video file.

The division of these steps for parallelisation across four threads can be seen in Figure 4.1.

While this could be divided across a variety of pipeline sizes, it was important to ensure that equal amounts of time were spent processing on each thread. This was due to a necessary barrier at the end of each iteration to ensure the correct image data was shared for each step. If one processor performs slower than all the rest, the others must sit idle until its task is completed.

Based on the requirement that all threads receive equal portions of work, parallel implementations were only developed using two and four threads. As some functions such as finding the back projection and performing the Camshift search algorithm were fairly expensive, this became the minimum amount of computation that could be performed by each thread. With a fixed number of steps, the largest sensible level of parallelisation was four threads. This could be further parallelised by adding extra tasks, however even four threads provided enough information for the experimental purposes.

### 4.1.2 Experimental Design

The Camshift algorithm has a few identified weaknesses including fast motions, changing lighting conditions, and variable object distances from the source camera. Scenarios with these three conditions were recorded using a high definition source video, which was compressed from 1920x1080 pixels to 1280x720 pixels and 854x480 pixels. These resolutions are described using their height dimension in the form of 1080p, 720p, and 480p with the p indicating that no interlacing occurs for the image sources. This means that each frame contains a complete image representation. A 16:9 width-to-height ratio was also enforced.

The Camshift algorithm was performed on each resolution, and output videos containing hand locations were produced. In these tests, if a greater number of frames success-
fully identified the hand location using higher resolution sources, this would confirm that the use of HPC could benefit real-time HCI. This is because high definition sources are unable to run with a serial implementation, however could be run on a parallel system.

For these purposes it is necessary to clearly state what is considered a success and what is considered a failure. A success was determined as a frame where the estimated hand location existed on the surface of the hand. A failure occurred when the estimated hand location did not lie on the hand. A comparison between a successful and unsuccessful hand location can be seen in Figure 4.2.

Once tests were completed on the different resolutions, performance measurements were taken for a serial implementation, a two thread pipeline, and a four thread pipeline. First the results from these executions were be saved to disk and compared with the serial version to confirm the parallel program was functioning correctly. Once this was confirmed, then the number of processed frames would be divided by the total run-time.
to get a frames per second measurement for each resolution and each implementation. As the real time HCI requirement had been set at 10 frames per second or higher, images were required to be processed at above this rate to be considered a successful candidate for HCI.

4.1.3 Hardware

Initial testing for the hand tracking experiments were performed using the EPCC Shared Memory system called "Ness". Ness was made up of two 16-core Symmetric Multiprocessing nodes with 2Gb of memory per core. As testing never surpassed the capabilities of this system, Ness served as the primary test system. Troubleshooting and development was performed on a personal laptop with a 2.4 GHz Intel Core 2 Duo processor, and 4 GB of memory. The timings from these tests were unreliable because processors were not reserved, though it demonstrated portability to personal systems where this application is highly likely to be deployed.

4.2 Disparity Mapping

4.2.1 Parallel Implementation

The disparity mapping experiment required a different parallelisation strategy than the pipeline used for hand tracking for two reasons. Most importantly, the number of individual tasks performed in the disparity mapping program was less than in the hand tracking case. As a result, the pipeline would be limited to a small number of tasks, making it less scalable. Due to the computational requirements of the Graph-Cut stereo matching technique, a higher level of parallelisation would be required for any benefits to be realised than in the hand-tracking case. Parallel requirements of this algorithm surpass the scalability of both a pipeline, and the Shared Memory programming model.
Disparity mapping was identified as a good candidate for a domain decomposition parallelisation strategy using Distributed Memory programming with MPI. In a domain decomposition, the stereo input images are divided into a number of smaller images and distributed to all the available processors. When the image is divided into smaller images, this is referred to as a decomposition, and smaller images are called sub-domains.

There are a number of ways to perform this decomposition as can be seen in Figures 4.3 and 4.4. These images show an original image with a one dimensional row decomposition, a two-dimensional decomposition, and a two-dimensional decomposition with added halo information. Halos are used when there are dependencies in a decomposition with their neighbouring sub-domains.

Individual processors perform the Graph Cut disparity mapping algorithm on their local image decompositions, and send the resulting disparity map to the controlling processor for a global reconstruction.

When performing disparity mapping, it was discussed in Chapter 2 that image calibration and rectification are first performed to ensure the stereo images are both planar and row-aligned. This was done to ensure disparities occurred only in rows, and suggests that individual rows should be entirely independent. For this reason, a one-dimensional row decomposition was selected as a good parallelisation method. Pre-calibrated image sources from the University of Tsukuba, and The University of RMIT were also examined.

After initial testing it was observed that unless the image source were well calibrated, there were obvious signs of variation from the serial image. In these cases the interface between subdomains was very distinct. This pattern can be seen in Figure 4.5 where a 64 processor one-dimensional row decomposition was used. The distinct sub-domain behaviour indicates there is some form of dependency in the algorithm being overlooked. For comparison, an example of a perfectly calibrated stereo pair can be seen with the Tsukuba Stereo Pair in Figure 2.5, and the corresponding Graph Cut disparity map can be seen in Figure 3.2 [33]. Initially these sub-domain errors were attributed to edge effects, and a halo of variable depth was added to the row decompositions. This reduced the discrepancies between subdomains, however did not solve the problem.

Further investigation revealed the energy minimisation step in the Graph-Cut matching algorithm was the true source of the subdomain interface errors. As sub-domains were finding different minimum energy configurations, the Graph Cut optimisation was providing discontinuous disparity values between decompositions. This explains why a perfect computer generated image would not be as affected by the decompositions as the less calibrated skydiving pair, since the initial disparity energies would be already represent a low energy match.

Unfortunately once this had been identified, re-writing this algorithm to perform a global energy minimisation was too large a task to complete within this project’s duration. The effect was reduced however by using a two-dimension domain decomposition, with an added halo with a depth corresponding to the largest estimated disparity values.
Figure 4.3: A source image is seen above, with a one dimensional row domain decomposition pictured below for four processors.
Figure 4.4: A two dimensional domain decomposition for the image seen in Figure 4.3, and a two dimensional domain decomposition with halo data included with red shading.
As a result, the final disparity map implementation used a two-dimensional domain decomposition with a halo depth equal to the largest disparity value. The stereo images were first captured by the processor on Rank 0, which was responsible for input and output (I/O). Once captured, an appropriate decomposition layout would be selected for the image to allow for even distribution amongst processors, and these images would be sent using point-to-point communication to each rank.

Individual processors would receive this image data, and calculate the disparity for their local subdomain. These local subdomains would then be sent back to rank 0 where they would be reconstructed into one image and saved to disk. This process can be seen in Figure 4.6.
Figure 4.6: The parallel behaviour of the domain decomposition disparity mapping parallelisation.

4.2.2 Experimental Design

As the Graph Cut algorithm was identified as both computationally expensive and as accurate, the goal in this experiment was to use HPC resources to achieve real-time speeds for computation. For these tests, a set of stereo images would be used to find a disparity map, and the output would be compared with that of the serial case. If the images provided the same disparity output image within acceptable real-time time scales, this would be determined a success.

This process would be repeated for images of varying resolutions and calibration efficiency to determine what effect these factors have on both the run-time, parallel efficiency, and disparity map accuracy. In hand tracking examples, a real-time minimum rate of 10 frames per second was identified which was set to allow for natural HCI. When discussing real-time computer vision for navigation the requirements are a little less clear, as the vehicle can simply adjust its speed to match the rate of information flow. This will be taken into consideration when assessing the algorithm performance.
4.2.3 Hardware

Preliminary testing for the disparity mapping application took place initially on a personal laptop with a 2.4 GHz Intel Core 2 Duo processor, and 4 GB of memory using oversubscribed MPI jobs. This allowed for simulated processor counts of up to 64 cores. The EPCC system Ness was also used for additional testing at the early stages. This confirmed portability of the disparity mapping technique.

Final tests were carried out on the UK National Supercomputing resource called HECToR; a Cray XE6 system with nodes made up of two 16-core 2.4 GHz AMD Opteron Interlagos processors with 1GB of memory per processor. These nodes exist in sets of four on blades, and there are 704 blades total bringing the total core count to 90112. Communication occurs using a 3D-torus of processors with a point to point bandwidth of 5GB/s, and a latency of 1.5 \( \mu s \).

4.3 Summary

This section outlined the design of the parallel pipeline implementation used for hand tracking, along with the design and motivation for a two-dimension domain decomposition disparity mapping implementation. The experimental procedures were stated for hand tracking and disparity mapping applications. The criteria for a positive and negative results in these experiments were also described. Finally the hardware used for each tests was described, with the final hand tracking experiments occurring on Ness, and the disparity map results taking place on HECToR.
Chapter 5

Results

5.1 Hand Tracking

The hand tracking results in Figure 5.1 show the number of frames in which the hand was successfully identified for the three different levels of resolution, along with the frames per second achieved at each resolution. In these results, it can be seen that with the lowest resolution 480p image source, 793 frames were successfully identified the location of the hand. When using the highest resolution 1080p source, 910 successful were frames were identified. As there were 930 frames total, this gives a success rate of 85.3% with 480p resolution, and a success rate of 97.8% with 1080p resolution. The intermediate case with 720p resolution achieved a success rate of 95.2%. These results show that the use of high resolution images improved the capabilities of the Camshift algorithm.

The number of successful frames was unchanged between serial and parallel implementations, which reveals the parallel implementation was successful. The parallel performance results seen in Figure 5.2 show that while when dealing with the highest resolution on four processors, only 7.19 frames per second were achieved. This is below the specified threshold of 10 frames per second for real-time HCI. The 720p case provided a frame rate of 20.68 frames per second, and a success rate improvement of 9.9% over the 480p resolution.

It can also be seen that the serial implementation also achieved results above the 10 frame per second minimum for 480p and 720p resolutions. In these cases the parallel implementations still provided large improvements in frame rates, which leads to more natural gesture interactions for the desired applications.

In Figure 5.3 we see the parallel performance for the various resolutions and implementations, along with the parallel efficiency in Figure 5.4. These results reveal a diminishing speedup value when moving from the two to four thread implementation. The maximum speedup on two threads occurred using the 480p source with a value of 1.54, and the maximum speedup on four threads occurred using the 1080p source with a value.
Figure 5.1: The number of successful frames and frames per second for three levels of resolution over four processors.

Figure 5.2: The frames per second calculation rates for varying processor count and resolution.
Figure 5.3: The speedup for three different resolutions and a varying number of processors.

Figure 5.4: The parallel efficiency for three different resolutions and a varying number of processors.
of 2.27. This is an example of Gustafson’s Law, as when the problem size was increased, the speedup also improved.

5.2 Disparity Mapping

In the disparity mapping experiments, the goal was to identify if parallelisation could be both possible and beneficial for the Graph-Cut Stereo Matching algorithm. A successful experiment would result in a parallel implementation providing an identical disparity map to that of a serial program. This would need to be done in suitable time scales for real-time applications. Correctness in these experiments was obtained through visual analysis. An alternative comparison could find the absolute difference between raw disparity data between serial and parallel versions, but this method exceeds the required accuracy for this test case.

Earlier observation identified that level of calibration between source images had an effect on the success of the disparity map output when using a parallel domain decomposition implementation. To better understand this relationship, images with varying levels of calibration were explored.

The results for an ideal calibration seen in Figure 5.5 were found using the computer generated Tsukuba pair. This shows the disparity map output for a two-dimensional domain decomposition across a varying number of processors. These results reveal that from for up to 8 processors, there was no apparent different between the disparity output and the serial disparity mapping. For processor numbers greater than this, there were slight fluctuations seen around the face of the statue, and the lens of the camera in the background. While the interface between sub-domains was not apparent for most decompositions, these small variations serve as indicators that the energy minimisation step is providing slight inconsistencies.

When tested with a poorly calibrated stereo set of skydiving images, the result of this energy minimisation became more dramatic as can be seen in in Figure 5.6. In these results, the disparity map only remained consistent with the serial output when run two processors. When using between four and 16 processors, the result remained similar to the source, but with some obvious artefacts. When the processor count increases further to 64 and 128 cores, the resemblance of the outputs to the serial case was reduced. The parallel occurred in rectangular blocks equal to the number of processors, indicating an broken dependency between subdomains.

Stereo samples using a Panasonic AG-3DA1 HD 3D camera [34] were also explored. This set of stereo images was more calibrated than the previous case with the skydiving images, however not 100% calibrated like the Tsukuba pair. Perfect calibration can only be guaranteed when using computer generated images [33]. The results in Figure 5.7 demonstrate that when using a well calibrated stereo pair with real world data, the results for domain decompositions with up to 64 processes remained very similar to the results for a single processor.
Figure 5.5: Disparity map for Tsukuba Stereo source, with the number indication how many MPI processes were used for calculation.
Obtaining accurate disparity map outputs for the parallel implementation was an essential requirement for this experiment. It was equally important however to identify if computation speeds made this a practical real-time solution. These results can be seen in Figure 5.8 which shows the number of frames per second calculated for of image sources with different resolutions with a varying number of processors.

The results seen in Table 5.1 show that the faster frame rates were provided by the lowest resolution source with a value of 2.52 frames per second. The frame rate for the 240p, 480p, and 720p reached a maximum when run on 32 processors. The 1080p stereo pair achieved a maximum when run on 64 processors.

The speedup from these experiments can be found in Figure 5.9 which shows that maximum speedup for the 240p, 480p, 720p, and 1080p stereo pairs as 14.29, 11.10, 9.12, and 22.60 respectively. The parallel efficiency seen in Figure 5.10 had a maximum value for all resolutions when run on four processors. This shows that the parallel efficiency for the two lower resolution sources reduced consistently as the number of processors increased, however the 1080p source plateaued at around 53% efficiency when run on 16 and 32 processors. This value decreased once again to 35.3% when run on 64 processors. This reduction was likely due to the 32 core node sizes on HECToR. With a 64 processor job, two nodes were required, resulting in a loss of efficiency due to increase communication overheads between nodes.

As disparity mapping had been expected to parallelise very well, parallel efficiency values for more than four processors of 60% or lower were than expected. To under-
Source

Figure 5.7: Disparity map outputs for a calibrated 480p 3D camera source stereo pair using different parallel decompositions, with the number of processors denoted by the numbers above each image.
Table 5.1: Disparity map performance measurements run with the University of RMIT stereo source for multiple resolutions.

Core Count vs. Image Resolution (Frames Per Second)

<table>
<thead>
<tr>
<th>Cores</th>
<th>Resolution</th>
<th>240p</th>
<th>480p</th>
<th>720p</th>
<th>1080p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.176</td>
<td>0.0458</td>
<td>0.00795</td>
<td>0.00175</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.551</td>
<td>0.151</td>
<td>0.0243</td>
<td>0.00544</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.32</td>
<td>0.385</td>
<td>0.0588</td>
<td>0.0151</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2.52</td>
<td>0.510</td>
<td>0.0724</td>
<td>0.0230</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.19</td>
<td>0.249</td>
<td>0.0724</td>
<td>0.0397</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.0480</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.0262</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.8: The rate of calculated frames per second vs number of processors for two different source image sizes
Figure 5.9: The speedup for two resolutions of disparity mapping sources for varying number of MPI processes.
Figure 5.10: The parallel efficiency of disparity mapping for various resolutions and processor counts.
Table 5.2: CrayPat results with the minimum, maximum, mean, and median processor run-times for 16 core decomposition, along with a high-water memory mark.

<table>
<thead>
<tr>
<th>Process Time</th>
<th>Process HiMem</th>
<th>PE=[mmm] Thread</th>
<th>PE=</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.199942</td>
<td>101.120</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53.337170</td>
<td>1159</td>
<td>pe.0</td>
<td>thread.0</td>
<td></td>
</tr>
<tr>
<td>42.524273</td>
<td>30.723</td>
<td>pe.13</td>
<td>thread.0</td>
<td></td>
</tr>
<tr>
<td>42.523480</td>
<td>30.418</td>
<td>pe.6</td>
<td>thread.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 shows that the processor with rank 0 was taking more than 10s more than all other ranks to complete for a single frame on 64 processors. Additionally, Rank 0 was using nearly 35 times more memory than the other ranks. It had been expected that Rank 0 would be causing some level of load imbalance due to its role handling the input and output data for this application. All ranks were required to receive a stereo image pair and compute the disparity map for these sources. This was then sent back to the controlling rank before beginning the next iteration. Rank 0 was also required to capture the initial images, send decompositions to the appropriate ranks, receive disparity maps, and reconstruct the final output. These additional tasks are what caused this load imbalance, however such a large magnitude of load imbalance had not been anticipated.
5.3 Summary

This section outlined the results for the hand tracking and disparity mapping experiments. In the hand tracking experiments it was seen that while higher definition video sources lead to improved hand tracking accuracy, the four thread parallel Camshift implementation was incapable of reaching real-time computation speeds for resolutions above 720p.

The disparity mapping experiments revealed that the calibration of the source images played a major role in the success of the disparity output. It was also seen that the frame rates achieved by the parallel implementation were between 0.01 and 2.5 frames per second. The serial performance for these cases was between 0.002 and 0.18 frames per second. This shows a clear improvement over the serial case, however results still remain below the guideline requirements of real-time computation. Finally, a large source of load imbalance was identified due to Rank 0’s I/O handling for all images.
Chapter 6

Conclusions

6.1 Hand Tracking

The results seen in Figure 5.1 clearly demonstrate the use of high definition input images for the Camshift algorithm resulted in a greater number of successful hand tracking frames. The success rates for 480p, 720p, and 1080p images were 85.3%, 95.2% and a 97.8% respectively, which shows a 12.5% improvement from 480p to 1080p. This confirmed that the use of HPC could be beneficial for hand tracking, but it was also necessary to confirm if this algorithm could be parallelised as required.

For these experiments a serial program, a two thread pipeline, and a four thread pipeline were investigated. In all cases, the 480p and 720p resolutions achieved real-time computation speeds. For the 1080p resolution, the serial version achieved 3.17 frames per second, the two thread implementation achieved 4.78 frames per second, and the four thread version achieved 7.19 frames per second. None of these results are sufficient for real-time interactions, however the parallel results showed improvements over the serial case.

While this particular experiment did not achieve the necessary processing speeds for the highest resolution case, it confirmed further work to do so would be worthwhile. Parallel efficiencies of around 70% we seen for these tests on the high resolution source indicate there may still room for more efficient parallelisation.

Even with an efficiency of 70% the parallel implementation could be quite helpful. We saw the accuracy from 420p to 720p increased by 9.9%, and moving 720p to 1080p provided an improvement of 2.6%. While 1080p resolution was not supported for real-time functionality in these experiments, the shift from 480p to 720p provided the majority of improvements for successful frame identification. The 720p image was also capable of running at 20.68 frames per second in parallel, an improvement of 8.62 frames per second over the serial case. This improvement would be very useful for natural HCI.
6.2 Disparity Mapping

The results from the disparity mapping experiments confirmed that HPC resources can be used to greatly reduce the required run time for the Graph Cut stereo matching algorithm. These results improved the frame rate for the University of RMIT stereo pair examined here at 240p resolution from 0.18 frames per second to 2.52 frames per second. While this is an improvement, it is still below the real-time threshold of 10 frames per second, and is not enough to allow for Graph Cut disparity mapping to serve as a primary navigation tool for autonomous vehicles.

The parallel performance of this program was worse than expected, with parallel efficiencies of around 50% or lower when using 16 cores or more. This shows room for potential improvement. If an efficiency of 90% could be achieved, this would allow the 240p source to be analysed in real time using 64 processors.

The source of this performance can likely be evidenced by the results seen in Table 5.2. This shows the run-time for various processors for the skydiving pair on 16 processors, and reveals that Rank 0 required 20% more time than all other processors. The memory requirement for Rank 0 was over 1000 Mb, while the other processors only required 30 Mb.

Rank 0 was expected to add some load imbalance to this implementation, as it was responsible for all the input and output tasks, along with sending and receiving image decompositions at the beginning of each iteration. The magnitude of this imbalance was not anticipated. This is a clear area for improvement, with a number of possible solutions. If this load imbalance were to be remedied, it is likely that the maximum achievable frames per second could grow significantly.

Difficulties were also encountered when dealing with the energy minimisation process performed by the Graph Cut algorithm. The minimisation attempts to compare several disparity arrangements for the image pair, and select the solution which has the most similar data, smooth neighbouring data behaviour, and minimal occlusions. When this is calculated for a subdomain, the benefit of neighbouring data is lost, resulting in discontinuities in disparities between subdomains. This behaviour was seen with the skydiving image pair.

When well-calibrated image pairs were used, this effect was greatly minimised. This yielded accurate results for up to 128 processors in the case of the Tsukuba Stereo Pair, and 64 processors with the RMIT pair.

The Graph Cut disparity matching algorithm demonstrated promising potential to improve the capabilities of real-time depth mapping when combined with HPC techniques and resources, but the implementation used in this report requires further work before it will be a useful tool. This work would need to address the issue of a local energy minimisation providing inconsistent results, as well as the significant load imbalance caused by the I/O technique.
6.3 Parallel Computer Vision

Throughout this report a number of computer vision applications have been discussed and the benefits of parallelisation have been explored. This report clearly shows that HCI and navigation applications stand to benefit from HPC resources and techniques, and this list is certainly not exhaustive. The field of machine learning already makes use of parallel resources for vision applications, and there is already some existing parallel support for threaded image processing for filters and colour conversions. This is done using the Intel’s Integrated Performance Primitives for multithreading.

We have also identified some possible programming techniques and bottlenecks which would effect future parallel computer vision applications from different fields. These include the potential for I/O bottlenecks, domain decomposition techniques, possible difficulties arising from global search methods, and probability estimates. Hopefully these results can help to improve HCI and navigation applications, as well as increase the level of parallelism in all computer vision techniques.

6.4 Future Work

6.4.1 Hand Tracking

The hand tracking results seen here revealed that parallel programming and resources improved capabilities over the serial case, however the parallel efficiency was only around 70%. The current implementation could be extensively profiled to identify if a better parallel efficiency could be realised. The actual Camshift algorithm could also have loop parallelism added to its search algorithm instead of relying entirely on task parallelism as we have in this report. New forms of parallelisation could allow for scaling beyond only four processors, which could possibly allow input resolutions much larger than just 1080p.

Another approach could be to incorporate all of the hand tracking methods discussed throughout this report using task parallelism, and obtain a weighted average between all three. In this case when one of the tracking methods fails, there could be incorporated failure detection and resolution techniques. This method of parallelism would be limited only by the number of useful tasks that could be conceptualised.

6.4.2 Disparity Mapping

These disparity mapping experiments served as a sound test case to determine if future work would be worthwhile. The implementation is still not entirely complete, with several possible directions for future work. One improvement would be to modify the program such that the Graph Cut algorithm’s energy minimisation occurs globally. This
impact can be mitigated through the use of well-calibrated sources, but this is not a robust solution. If the minimisation could take place globally without impacting the parallel performance drastically, there would be a large improvement in accuracy for under calibrated sources.

The current implementation uses a single processor for all I/O tasks, which creates a bottleneck. This could be remedied using MPI's parallel IO feature called MPI IO to split the I/O tasks amongst all the processors equally. This would allow for much faster processing times. Based on the added overhead of around 20% for a single iteration seen in Table 5.2, if each frame could be calculated 20% faster this would result in a frame rate with a 240p source of roughly 3 frames per second. Additional would be realised through the use of MPI IO, as the cost of message synchronisation would be reduced as well.

### 6.4.3 General Purpose Graphical Processing Unit Programming

For these experiments, traditional processors were used for the core processing tasks. An interesting alternative to this would be the use of General Purpose Graphical Processing Units (GP-GPUs), which are well suited to processing image data as they were initially developed, to generate the image data for video games. These provide a low-power solution for high performance computing which would be well-suited to navigation applications that are not tethered to the electrical grid.

### 6.4.4 Computer Vision Benchmarking

The final area that could be interesting for future work would be the development and standardisation of a computer vision benchmarking tool. There are a large number of papers discussing hand tracking and disparity mapping, but it is difficult to compare between results because the implementations can be widely varied, and it is rare to find examples showing the failure cases. With an established set of tests for each computer vision function, along with performance metrics, the field of computer vision could begin to quantify more clearly what the best methods and functions for a given purpose are.

### 6.5 Project Overview

Overall the risk assessment and schedule developed in semester two for the Project Preparation course served as an excellent guideline for this Dissertation. Nearly all of the identified risks ended up occurring, however the allotted time for solving these problems along with the proposed mitigation techniques served to keep things on schedule.
This project was ambitious in its goals; requiring two computer vision applications to be written from the ground up, with robust and optimised final products. This also included writing a number of supporting applications, such as a hand colour calibration tool, a stereo camera calibration program, and a stereo video splitting program to separate a pair of stereo videos into individual files. Due to these challenges, the mitigation strategy was simply to reduce the depth of the experiments.

This was seen with the hand tracking application, as the original goal also planned to include gesture recognition as well as the weighted average of multiple tracking techniques. Partway through the summer it was decided to simply focus on hand tracking, as this focus provided sufficient information to assess the main experimental goals.

At this point the work on disparity mapping began. In this experiment, the results from the one-dimensional row-decompositions were a major setback. At this point it became clear that the final program would not be a finished product.

In summary, there were a number of setbacks along the course of this project, however it was nothing the risk assessment had not identified. In response to these setbacks the depth of analysis for both hand tracking and disparity mapping was reduced, however the primary goal of this project was still addressed. In both cases it was demonstrated that there are clear benefits to parallelising real-time computer vision. Hopefully further work can be done by myself or future students, as it is an exciting and unexplored field of work.
Bibliography


Appendix A

A.1 Glossary

Central Processing Unit (CPU) - An integrated circuit that performs the computational requirements for a program’s execution.

Computer Vision - The use of image information for processing and analysis to provide a new representation or decision

Disparity - The distance between an objects representation from one stereo image source to the other.

Disparity Map - A visual representation of disparities between a stereo image pair with image colours or shades corresponding with specific disparity values.

Distributed Memory Programming - A parallel programming technique where communication takes place in the form of messages between processors, and each processor has its own set of local data.

Domain Decomposition - A parallel design pattern where a set of data is divided equally amongst processors, and the same task is performed on each of these divisions.

Frames Per Second - A measurement of how many frames of the source image could be calculated per second.

Hand Tracking - Identifying the location and orientation of a hand in each frame of a video.

High Performance Computing (HPC) - The use of computing resources such as accelerators, multi-core processors, and massively parallel systems for cutting edge optimisation and program performance.

Human Computer Interaction (HCI) - A field of computer vision which attempts
to replace or compliment current user interface devices such as a keyboard and mouse using natural controls like voice and hand movements.

**Message Passing Interface (MPI)** - An API used for sending information between processors for Distributed Memory programming.

**Open Multiprocessing (OpenMP)** - A Shared Memory programming API which uses pragmas to identify parallel regions and define shared variables.

**Oversubscribe** - A parallel processing technique where additional cores are simulated through the use of software to test program functionality for different sizes of systems.

**Parallel Efficiency** - This is a parallel performance metric independent of the number of processors $N$ that describes how effectively parallelisation has taken place.

**Real-Time Computing** - Computation occurring simultaneously with live events. Typically requires 10-100 frames per second of computer vision analysis.

**Shared Memory Programming** - A parallel programming technique that makes use of a shared cache between participating processors, and communicates using shared variables amongst all processors.

**Serial Region** - A portion of a program that must be run on a single processor or in a fixed order.

**Speedup** - A term used to described the performance increase realised when run on $N$ processors.

**Stereo Image Pair** - A pair of two images with a fixed distance of separation called a baseline, with the same image contents from slightly different perspectives.

**Sub-Domain** - A term to describe the smaller used on each processor in the Domain Decomposition method.
A.2  Project Preparation Risk Assessment and Schedule

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Risk</strong></td>
<td>Computer vision algorithms may be more complex than initially expected.</td>
</tr>
<tr>
<td><strong>Liklihood</strong></td>
<td>High</td>
</tr>
<tr>
<td><strong>Mitigation</strong></td>
<td>Additional time will be budgeted for development, two phases of development used with Phase Two as optional, and complimentary.</td>
</tr>
<tr>
<td><strong>Impact</strong></td>
<td>Medium (4 weeks)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Risk</strong></td>
<td>OpenCV may not be portable for the various architectures to be used in testing.</td>
</tr>
<tr>
<td><strong>Liklihood</strong></td>
<td>Low</td>
</tr>
<tr>
<td><strong>Mitigation</strong></td>
<td>System compatibility will be tested early on. The OpenCV community will be consulted in case of problems.</td>
</tr>
<tr>
<td><strong>Impact</strong></td>
<td>Low (&lt; 1 week)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Risk</strong></td>
<td>Motivation and focus may be lacking as has been seen in reviewing past dissertations.</td>
</tr>
<tr>
<td><strong>Liklihood</strong></td>
<td>Medium</td>
</tr>
<tr>
<td><strong>Mitigation</strong></td>
<td>Careful time management and scheduling. Multiple work tasks will be scheduled to allow for breaks from a specific problem.</td>
</tr>
<tr>
<td><strong>Impact</strong></td>
<td>Low (1 week)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Risk</strong></td>
<td>Inconclusive results may arise due to qualitative nature of computer vision test metrics.</td>
</tr>
<tr>
<td><strong>Liklihood</strong></td>
<td>Medium</td>
</tr>
<tr>
<td><strong>Mitigation</strong></td>
<td>Clear definition of assessment techniques, attempt to quantify qualitative elements.</td>
</tr>
<tr>
<td><strong>Impact</strong></td>
<td>Low (1 week)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Risk</strong></td>
<td>Miscellaneous technical difficulties such as compression format requirements.</td>
</tr>
<tr>
<td><strong>Liklihood</strong></td>
<td>High</td>
</tr>
<tr>
<td><strong>Mitigation</strong></td>
<td>Flexible time budgeting and early testing.</td>
</tr>
<tr>
<td><strong>Impact</strong></td>
<td>Medium (2 weeks)</td>
</tr>
</tbody>
</table>

**Programming Tasks:**

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Serial Hand Tracking:</strong></td>
<td>Develop a serial hand tracking and gesture recognition algorithm as discussed in Phase One.</td>
</tr>
<tr>
<td>2. <strong>Parallel Hand Tracking:</strong></td>
<td>Develop a parallel version of the previous serial hand tracking algorithm.</td>
</tr>
<tr>
<td>3. <strong>Dense Stereo Vision:</strong></td>
<td>Create a dense stereo vision disparity mapping algorithm as discussed in Phase Two.</td>
</tr>
<tr>
<td>4. <strong>Performance Measurements:</strong></td>
<td>Perform extensive performance measurements and gather results for analysis in the report.</td>
</tr>
</tbody>
</table>

58
Figure A.1: Dissertation project work schedule.

Report and Presentation Tasks:

<table>
<thead>
<tr>
<th>1. Introductory Material:</th>
<th>Write the sections of the report dealing with background and literature review.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Algorithms and Implementation:</td>
<td>Complete the algorithm and implementation sections of the report.</td>
</tr>
<tr>
<td>3. Results and Performance:</td>
<td>Complete results section using performance measurements of the algorithms.</td>
</tr>
<tr>
<td>4. Presentation:</td>
<td>Prepare a presentation based on this report.</td>
</tr>
<tr>
<td>5. Finalise Report:</td>
<td>Proof read the report and try to eliminate as many errors as possible.</td>
</tr>
</tbody>
</table>

A.3 Image Samples and Outputs

See attached storage device.