

|epcc|



Finding Wally

Parallel Object Recognition



Cian Booth
MSc. Dissertation Report

Abstract

A report on the potential of parallel computer vision for everyday use. Discussed within are algorithms and libraries that could make this a possibility. Special focus is put on the computer vision library OpenCV. Where's Wally? puzzles are used as a test case for parallel object recognition, with OpenMP used for parallelism. The program experiences moderate speedup up to 16 cores. The reliability of the program's solutions was not generic, requiring tweaking to become correct.

Contents

1	Introduction	1
1.1	Computer Vision	1
1.2	High Performance Computer Vision	2
1.3	Potential Use Cases	2
1.4	Where's Wally? as a Test Case	4
1.5	Goals	4
1.6	Overview of Report	5
2	Background Information	6
2.1	Computer Vision	6
2.2	Object Recognition	6
2.3	Shape and Colour Analysis	6
2.3.1	Feature Analysis	8
2.4	Computer Vision Libraries	8
2.5	Parallel Programming	9
2.6	Parallel Computer Vision	10
2.7	Shared Memory Parallelism	11
3	Algorithms	14
3.1	Colour Analysis	14
3.1.1	Colour Extraction	14
3.1.2	Blur	15
3.1.3	Sharpen	17
3.2	Region Detection	17
3.2.1	Naive Solution	17
3.2.2	Flood Fill	18
3.2.3	Connected Component Labelling	20
3.3	Line Width Estimation	22
3.3.1	Average Distance to a Zero Valued Pixel	23
3.3.2	Standard Deviation in Distance to Zero Valued Pixels	26
3.3.3	Combining the Methods	28
3.3.4	Testing	31
3.3.5	Parallelism	31
3.4	Shape Analysis	32
3.5	Feature Detection	32
3.5.1	Scale-Invariant Feature Transform	32
3.5.2	Speeded Up Robust Features	33
3.5.3	Matching: Brute Force Matcher	33
3.5.4	Matching: FLANN	33
3.5.5	Matching: Relative Keypoint Distances	34
4	Implementation	36
4.1	Function Implementation	36
4.2	Pattern: Red and White Stripes	38
4.2.1	Weaknesses	39
4.2.2	Testing	41
4.2.3	Parallelism	41
4.3	Pattern: Blue Trousers	41
4.4	Find Glasses	42

4.5	Find Features	42
5	Results	44
5.1	Extracting Greyscale from Image	44
5.2	Line Width	45
5.3	Region Detection	46
5.4	Find Features	46
5.4.1	Reliability	47
5.4.2	Speedup	50
5.5	Red And White Stripes	50
5.5.1	Reliability	51
5.5.2	Speedup	51
5.6	Find Glasses	52
5.6.1	Speedup	52
5.7	Blue Trousers	52
5.8	Speedup	53
5.9	Using All Techniques	53
5.9.1	Speedup	53
5.10	Genericism	54
6	Conclusion and Evaluation	57
7	Evaluation	58

List of Figures

1.1	A map of Edinburgh	3
1.2	Characters from Where's Wally?	4
3.1	Gaussian Blurring	16
3.2	Steps in a naive region detection algorithm	18
3.3	The Flood Fill Algorithm	19
3.4	The Connected Component Labelling Algorithm	21
3.5	Graphic description of parallel region detection	22
3.6	Examples of the zero-distance of black lines	24
3.7	Relationship between line width and mean zero-distance	25
3.8	Relationship between line width and standard deviation .	27
3.9	Maximum zero-distance reduction to produce sane mask	29
3.10	Sensitivity of mean and standard to errors	30
3.11	Test images for estimate_black_line_thickness	31
4.1	Isolating yellow from an image using get_colour_in_image	38
4.2	Example of finding red and white striped regions	39
4.3	Fail case for the 'Red and White' pattern	40
4.4	Test images used to check the red and white pattern . . .	41
5.1	Time and speedup of extracting greyscale from an image	44
5.2	Time and Speedup of parallel line width analysis	45
5.3	Time and Speedup of parallel region detection	46
5.4	Find Features example output	47
5.5	Find Features stripes example output	47
5.6	Reliability of the Find Features pattern for known Objects	48
5.7	A Comparison of Object Resolutions	49
5.8	Reliability of the Find Features pattern for unknown Objects	49
5.9	Time and speedup of the Find Features pattern	50
5.10	Reliability of the Red and White Stripes pattern with optimal settings	51
5.11	Reliability of the Red and White Stripes pattern with sub-optimal settings	52
5.12	Time and speedup of the Red and White stripes pattern .	53
5.13	Speedup of the Find Glasses Pattern	54
5.14	Speedup of the Blue Trouser Pattern	55
5.15	Complete program speedup	56
5.16	Extending the program to find Odlaw	56

List of Tables

3.1	Example of Colours in Hexadecimal format	14
3.2	Process of sharpening an image	17
3.3	Table of estimated line widths of test images	31

Acknowledgements

Many thanks go to my supervisor, Alistair Grant, for directing me to such an interesting project, and keeping me on the rails.

1 Introduction

The field of computer vision studies the use of computers to process and analyse images. Some aspects allow robots and other machines to process visual data. Others are an attempt to simulate the human ability to perceive visual information. This portion of computer vision allows users to reduce the time taken to produce information about an image. It can also help to increase the confidence that results related to that information is correct.

1.1 Computer Vision

Recognising an object is one of the most important things that human vision is able to do. This means that the brain is able to decipher the information received from the eyes, identifying any objects the information describes. For the average human can expect to see the local environment and immediately recognising objects within. The act of object recognition is a subconscious one[1]; humans do not have to actively think about their environment to understand what objects exist there. As a subconscious act, the complexity associated with human object recognition is not known, but can be estimated. The level of information contained in a single 'frame' of vision is extremely high. A normal room can contain many thousands of distinct shapes and surfaces, which must be pieced together to form a cohesive whole. Light levels vary from one position in a room to another, so objects may appear different at different angles. The human brain is able to analyse this data in real time while making decisions about objects in the image.

The adage "a picture is worth a thousand words" is particularly meaningful here; creating and expressing a logical definition of an object would be an extensive and verbose task. This logical expression would have a high level of complexity, needing precise descriptions including colour, shape and texture. It is a complex task to begin to define these values in a meaningful way.

Implementing this on a computer is complex. The normal description of a computer is a machine that is designed to calculate mathematical operations more accurately or faster than a human could. Modern computers are binary devices, designed to store and act upon precise values. Human vision, as currently understood, does not readily map onto the types of operations computers excel at.

Despite these difficulties, computer vision is an impressively mature field. Algorithms including Scale-Invariant Feature Transform (SIFT)[2] have been developed, which are able to find known objects under various transformations. It is used in numerous areas, from robotics control, automation defect recognition in manufacturing and tracking user motions with devices like the Kinect. Computer vision has the potential to be useful in countless fields, and in many aspects of everyday life.

1.2 High Performance Computer Vision

One of the major obstructions for widespread use of computer vision is the limiting relationships between speed, accuracy and genericism. For this report, speed is defined as the wall time (externally measured time taken to complete) of the program. Accuracy is the confidence with which results can be said to be true, and genericism is how easily the techniques can be adapted to different problems. The human eye can detect most objects quickly, accurately and generically. Computer vision is not so advanced. Achieving real time speeds comes at the cost of accuracy or generality. Conversely, creating a fast or generalised system comes at the cost of greatly increased completion time.

For each way that a human can describe an object, multiple techniques emerge to locate the object. Each technique has varying requirements and reliability. Some need specific descriptions (e.g. a sample image) and produce very reliable results. Others do not require such precise descriptions, but can produce results that are not as dependable. By combining these techniques, the reliability and robustness of computational object recognition should be improved. Using all available techniques can considerably increase the runtime of the program. This can be mitigated by computing techniques in parallel.

An suitable method of parallelism for this type of problem is the task farm. This means that tasks (here, identification methods) will be run concurrently. Task farming is useful in this case, as it allows the use of serial libraries and algorithms that can complicate or prevent parallelism. Computing results this way, a task farm allows computer vision programs to produce results quickly, reliably and generically.

1.3 Potential Use Cases

Parallel object recognition is a powerful technique that could be useful for many different areas.

The UK Missing Person Bureau released data on missing persons in 2010-2011 [3]. During this period, two thirds of missing people in the UK were under the age of 18. This group is particularly vulnerable to abduction and abuse when left unsupervised. Although the majority of missing people were found within the 5 miles of their homes, up to 21% of people were further out. A 5 mile radius is a large area to look for one person, especially in an urban area. Further out it becomes very difficult to search efficiently. Figure 1.1 shows the area that a 5 mile radius contains encompasses most of the urban area of Edinburgh city. Almost 4/5 of missing people would likely be found within the first 16 hours within a 5 mile radius. According to information released from a survey of young runaways [4], 34% of said they had been harmed or in a risky experience more than once, and 11% expressly said they had been hurt or harmed. It is critical to find at risk individuals before they are harmed. Although computer vision is to facilitate searches, the volume of data produced can exceed the capabilities of existing programs and systems. Furthermore, surveillance data from CCTV equipment produces data at real time speeds, so faster than real time processing speeds are desired. Real time speeds means that the program is able to analyse the data as

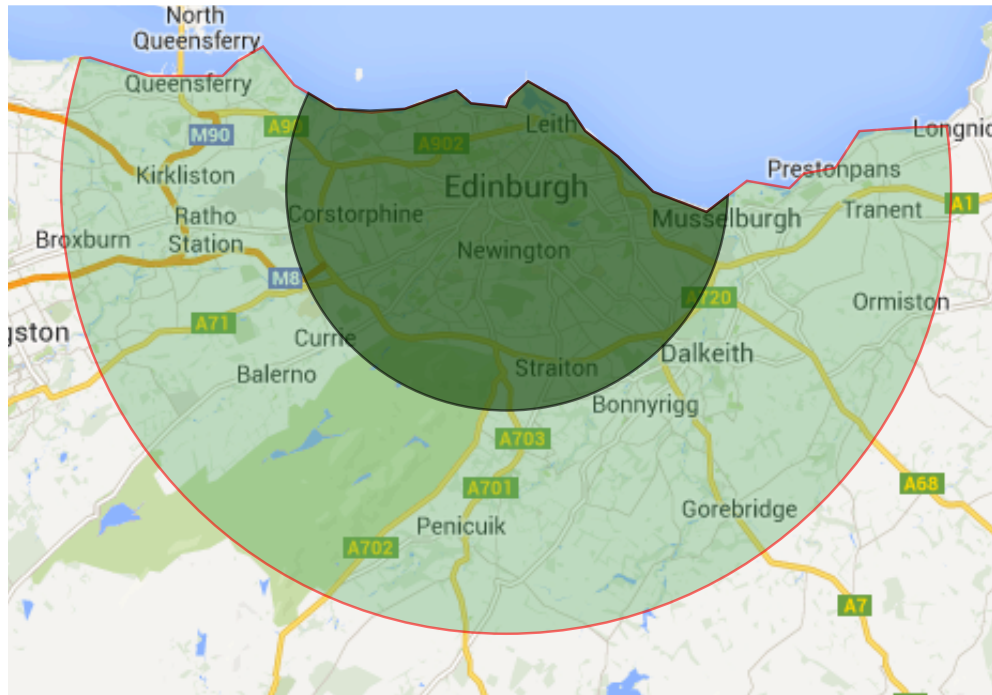


Figure 1.1: A map of Edinburgh, with 5 and 10 mile radius areas displayed. A 5 mile radius circle includes the majority of the urban areas of Edinburgh city. Base image courtesy of Google Maps.

fast as it is being produced. This may involve reducing the amount of data being input i.e analysing 1 in every 30 images. Implementing a parallelised version of image recognition tools should greatly reduce the time an image takes to search. Governmental agencies, such as the police, have access to very large computer systems[5], potentially allowing for a high degree of parallelism.

Another potential use can be found with surveying populations of wild animals. Wildlife conservation is a delicate task, which could benefit from rapid computer vision techniques. To accurately know which species are endangered, it is important to have an accurate count of the members of the species for a given region. Actively surveying the population by means of physically interacting with members can have adverse effects on the population. Nielsen [6] discusses the negative effects of electrofishing on rare fish populations. She discusses the lack of non-invasive methods of surveying the population, without which population counts cannot be maintained. Directly surveying endangered species can be inefficient, slow or dangerous to either the researcher or the animal in question. Passive techniques, such as photography, allow the researcher to estimate populations without interacting with the environment. Ideally a researcher would be constantly vigilant and able to immediately identify each species correctly. This is rarely the case; a single human is fallible and a team is often beyond the funding of the endeavour. Instead, with access to any modern laptop and a digital camera, parallel computer vision may be able to assist in many ways. A video feed would allow observation for as long as the battery lasts, and a database of the features of regional species would help with identification. Parallel species recognition would allow multiple species to be surveyed at a time. It would allow laymen to monitor the survey of the

populations, freeing up experts for more specialised tasks.

An everyday use of parallel object recognition is nationwide traffic monitoring. Using existing roadside cameras, such as CCTV or speed cameras, a network could be built that monitors traffic on a large scale. This would help to improve commute times and general congestion issues, by advising drivers of congested areas and providing alternate routes. Parallelism could be applied to this situation. The sheer quantity of data for a large scale system like this would prevent real time analysis for a serial program.

An use that is both simple and approachable can be found in Where's Wally? puzzles. This will be examined closely in the following section.

1.4 Where's Wally? as a Test Case

Where's Wally? puzzles are a good test for parallel computer vision. Each puzzle is a cartoon image filled with various characters, who wear simply coloured clothing, see Figure 1.2(a). One of these characters is the eponymous Wally, who dresses distinctly from most others characters, see Figure 1.2(b). Similarly dressed characters exist, Figure 1.2(c), to add complexity in finding Wally correctly. The cartoon nature of the characters means that shapes are boldly coloured and generally bordered by a black line. As Where's Wally? is a puzzle, Wally will be hard to find; he can be obscured, camouflaged or simply small. Creating a program to solve Where's Wally? is non-trivial, requiring a combination of computer vision techniques. Thus the puzzle provides a good base to develop parallel object recognition.

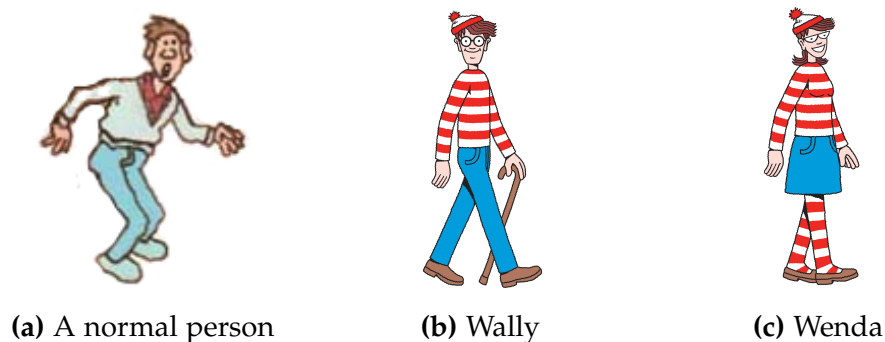


Figure 1.2: Characters from Where's Wally?

1.5 Goals

This report presents Where's Wally? as a testbed to determine if High Performance Computer Vision can be feasibly used in everyday life. This includes the production of a suite of functions that, though tuned to locating Wally, could be used to find other characters. The use of directive based parallelism, will be added, to determine if a user-friendly system is viable. Directive based parallelism uses preprocessor flags to indicate areas that could use parallel techniques. This is as opposed to using specific libraries and function calls to implement the parallelism.

A system that is intended for everyday use, must be simple and usable. This means that any developer should be able to produce a program

for general use. For example, HTML is an extremely simple and usable tool, it requires very little expertise and can be used in a large number of situations.

The more expertise required to implement parallel computer vision techniques, the fewer people are capable of creating solutions. In turn, this limits the man-hours available for producing programs, restricting implementations to the most pressing of problems. This limits the everyday usability of parallel computer vision, as there would be far fewer everyday programs.

The genericism of the suite of functions will be tested on non-Wally puzzles, to determine how generic the solution is.

1.6 Overview of Report

This report will begin by discussing the underlying information required to comprehend parallel object recognition. This includes literature reviews. The next section discusses the patterns used to recognise Wally. Each includes an analysis of the algorithm selection and a details of the level of parallelism which can be exposed. This will be used to produce a Where's Wally? solver, which examines the parallelism used to increase the speed of the Where's Wally solver.

The report will comment on the results produced by the patterns and the solver. Following this, will be the concluding statements and ideas, along with recommendations for future use. The report will close with an evaluation of the project as a whole. Differences between the preparation phase and the actual report period will be noted here.

2 Background Information

2.1 Computer Vision

Computer vision is a wide ranging field with a large variety of algorithms and libraries available for use. It is involved with topics such as artificial intelligence[7], machine vision[8], and image processing. Computer vision draws upon a large range of established fields, such as mathematics, physics and neurobiology. Many of the most complicated tasks use techniques developed for linear algebra, such as the transpose operation. Others rely on the understanding of human cognition to produce algorithms that far outperform previous efforts.

Computer vision techniques can be used for numerous tasks, including motion analysis, image restoration and object recognition.

Motion analysis is used to determine object velocities in an image stream. This could be due to the motion of the camera, motion of visible objects or a combination of the two. This has a number of applications, but has seen a rise in public use with devices like the Xbox Kinect[9]. The Kinect enables the user to interface with the Xbox through by tracking the motion of their body. Users do not need additional controls to select menu options, scroll windows or even play games. This is a good example of how computer vision can be applied, and a good indication of how ready it is for everyday use.

Image restoration is the recovery of corrupt or otherwise marred images. An interesting application is the recovery of medical images[10]. Obtaining high resolution images from MRI scans normally requires a large number of measurements. This is time consuming, which is distressing for the patient and expensive for the hospital. Taking a smaller number of measurements on purpose can help to improve the situation without forgoing quality. This is because image restoration allows the construction of high resolution images with only a small amount of input data.

The computer vision topic that concerns Where's Wally? images the most is object recognition. This is covered in depth in the section 2.2.

2.2 Object Recognition

Object recognition is a key technique in computer vision. In 1965, Roberts wrote the first known paper on computer vision[11].

Finding Wally in these puzzles is by definition a problem of object recognition. Wally must be correctly identified from other characters, furniture and even food. The image needs to be analysed so that the correct Wally can be located. Three of the most directly useful techniques for defining objects in a cartoon style image are shape analysis, colour analysis and feature analysis.

2.3 Shape and Colour Analysis

One way of analysing an image is to break it down into shapes and colours. In linguistic terms, it is often simplest to depict an object by

describing it's shape and colour, i.e. "the red box" or "the green hand". This is conceptually simple to explain, understand and program.

Everyday object recognition is generally be done through digital cameras, scanners and similar devices. Most images saved this way are stored as raster images (e.g. PNG, BMP, GIF), which is a 2D array of colours that directly map to pixels on the screen. This is because the input devices do not have the capability of recognising objects in the images they produce. This differs from vector images (e.g. PDF, SVG, SWF), which store the location and colour of geometric primitives (squares, circles, triangles etc.). It is often much simpler to perform shape and colour analysis on vector images. This is because the shapes and colours are explicitly stored, so no work is needed to be done to locate them.

Regardless of format, methods of colour analysis can be implemented programmatically. This is because colour is intrinsic to all methods of storing the properties of an image. It is nearly impossible to describe a specific object without discussing some aspect of it's colour, even it that colour is greyscale.

Shape analysis is more complex for raster images. Unlike vector images, shape boundaries are not clearly defined. The global boundary can normally found through edge detection. The global boundary is defined here as a single object composed of every boundary in the image. Detecting the shapes requires the specific boundaries to be found which requires further analysis. The global boundary is analysed to find points of boundary intersection, and the curvature between those points. Group of curves must examined to find combinations that produce shapes.

Shape and colour analysis allows for more linguistic method for locating objects. In Where's Wally puzzles, this means no previous image of Wally is needed. All that is required is a basic description, such as "red and white stripes" or "black glasses". This allows users to extend the solution past Wally, to other characters, who have not previously been seen. This form of analysis can lead to false positives, because a single description could encompass several different characters or objects. To prevent this, many different types of analysis should be combined. For example, a match for "red and white jumpers" that also has a nearby match for "skin colour" would produce a result that has more likelihood of being correct.

An example of the usefulness of shape and colour analysis, beyond Where's Wally, could be found in augmented reality (AR) technology. Google is currently developing this technology with the Google Glass device[12].

A common experience is the misplacement of keys. Users with access to AR devices could use colour and shape analysis to enhance their own searching (i.e. if they are visible, but in a cluttered area). As keys, with some exceptions, have a few well defined shapes and possible colour schemes; the device would not have to store what the specific keys look like in advance. Assuming that parallelism is available in such devices, this technique could potentially provide real time analysis of the scene, significantly helping the user.

2.3.1 Feature Analysis

Some of the most reliable computer vision algorithms (such as SIFT[2]) were developed while considering the neuroscience of human vision. Tanaka[13] and Perrett and Oram[14] studied this in detail. They found that human vision identifies objects with features that are invariant to brightness, scale and position. Humans are able to recognise the same objects under differing light levels, at different distances and in different positions in a room. These results have been used as a basis for feature analysis.

This technique finds ‘features’, which are points in an image that are scale, rotation and illumination invariant. These features are most immediately useful when compared with the features of another image. The features from a solo image Wally can be used to locate the same Wally in a group image.

The image that is being searched is often referred to as the Scene. Images that are being searched for in the Scene are known as Objects¹. The keypoints of Objects are known properties, and can be searched against the unknown keypoints of the scene.

This method requires an existing image to find Wally, restricting the flexibility of the search. When correctly implemented, this method is very reliable, as a high number of nearly unique keypoints must be matched to obtain a match. If Wally is not obscured, results found can be assigned high confidence. Normally Wally is obscured, so this method should be combined with other techniques. Combination helps to provide more flexible and reliable solution.

Feature analysis is useful in the manufacturing industry. Mass manufactured products need to be checked for defects. Due to the regularity of most merchandise produced this way, this is ideal for feature analysis. If products in the Scene do not match the features of the list of Objects, then the product can be determined to be faulty. This can simply be parallelised; multiple production streams can be scanned simultaneously. Using a centralised system could be cheaper than implementing per-line machines. This is because it could be integrated with existing systems, and help to centralise maintenance.

2.4 Computer Vision Libraries

Since the advent of computer vision, many libraries have been developed to implement and group computer vision techniques. One of the most extensive is OpenCV.

OpenCV[15] is an open source library used for implementing a wide range of computer vision techniques. Using in-built functions, users have immediate and simplified access to complicated algorithms. OpenCV is available for three major operating systems, Linux, Windows and Mac OS. The library is aimed at developing real-time solutions to computer vision problems[16]. The online documentation for OpenCV is extensive, including tutorials for common topics such as image recognition, ma-

¹Scene and Object are capitalised here to avoid confusion with the more general term of object

chine learning and image processing. These properties make OpenCV ideal for implementing computer vision on a wide scale. OpenCV has some drawbacks. It makes use of bespoke classes for dealing with arrays, called Mat. The Mat class helps to minimise the memory usage of programs using large arrays. Direct pixel access to Mat classes is not accessible in the normally expected C++ fashion. A template function, `Mat::at<type>(x,y)`, provides the method of access. This could confuse new users.

Another computer vision library is libCVD[17], based in Cambridge University. This is a versatile library, designed for speed and portability. It is often used to access streams of video data. Unlike OpenCV, it stores in pixels in readily accessible STL vectors. This is useful, because it makes accessing pixels more intuitive. However, libCVD does not have the range of implemented computer vision techniques found in OpenCV. It lacks the level of documentation that OpenCV offers. LibCVD can be used in combination with OpenCV[18]. This allows the user access to the breadth of functions available in OpenCV, as well as the speed of input that libCVD brings.

OpenCV will be the sole library used to implement the Where’s Wally? solver. Using multiple libraries complicates the task, where a key aim is to produce a simple everyday implementation. OpenCV has many attractive properties for non-expert users, when compared with other libraries. LibCVD, for example, does not have the documentation or community to help with developmental issues.

2.5 Parallel Programming

When implementing parallelism, it is desirable to speed up a program and use an efficient number of cores. A useful definition is speedup; the scale to which a parallelised version of a program is faster than an efficient serial implementation.

$$\text{Speedup}(P) := \frac{T_{\text{serial}}}{T_{\text{parallel}}(P)} \quad (2.1)$$

Equation 2.1 describes the speedup of a system. Here P is the number of computing units (threads, processors etc.), T_{serial} is the time the serial program takes and T_{parallel} is the parallel time. For most systems, the maximum speedup possible is equivalent to P , known as linear speedup.

Most implementations do not achieve linear speedup, especially for large numbers of computing units. A program that solves a problem of fixed size with near-linear speedup is said to experience ‘strong scaling’. In general, scaling will be limited being strong long before this happens, due to Amdahl’s law, seen in equation 2.3.

$$T_{\text{parallel}}(P) = T_{\text{serial}} \left(\alpha - \frac{1 - \alpha}{P} \right) \quad (2.2)$$

Here, α is the portion of the code that is purely serial. Inserting this into the previous definition of speedup in equation 2.1, we see Amdahl’s Law emerge.

$$\text{Speedup}(P) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(P)} = \frac{T_{\text{serial}}}{T_{\text{serial}} \left(\alpha - \frac{1 - \alpha}{P} \right)} = \frac{1}{\alpha + \frac{1 - \alpha}{P}} \quad (2.3)$$

As P tends to large numbers the speedup approaches the constant value of $1/\alpha$.

Parallel programs instead often aim for to achieve ‘weak scaling’, described by Gustafson’s law. This is the case if the problem size scales with the number of cores for a fixed amount of work per computing unit.

$$T(P, N) = T_{\text{serial}} + T_{\text{parallel},N} = \alpha + \frac{N(1 - \alpha)}{P}$$

$$T(1, N) = T_{\text{serial}} + NT_{\text{parallel},N} = \alpha + N(1 - \alpha)$$

Here, P is the number of computing units, and N is the size of the problem. Defining $T(P, N)$ as the time for the program to complete the purely serial sections. T_{serial} is the time it takes for each of the P processors to complete one task of size $1/N$, $T_{\text{parallel},1/N}$. The time for 1 processor to do an equivalent amount of work is defined in $T(1, N)$. This is the sum of the serial time T_{serial} and every bit of work the processors would do, i.e. $NT_{\text{parallel},N}$.

$$\text{Speedup}(P, N) = \frac{T(1, N)}{T(P, N)} = \frac{\alpha + N(1 - \alpha)}{\alpha + \frac{N(1 - \alpha)}{P}}$$

$$\text{Speedup}(P, \beta P) = \frac{\alpha + \beta P(1 - \alpha)}{\alpha + \beta(1 - \alpha)} \propto P \quad (2.4)$$

Here, β is some scaling constant. It is clear that Gustafson’s Law produces better scaling, as long as N scales with P .

Everyday users have most regular access to computer vision techniques through mobile devices, such as phones, laptops and ipads. These devices rarely have more than 4 cores. The most cores that can be expected is within a home computer, having no more than 8 cores Any parallelism applied should not require the typically large number of cores.

2.6 Parallel Computer Vision

Computer Vision poses an interesting challenge for parallelisation. In some regards, computer vision is a typical data parallel problem. Data parallelism uses the available computing units to act on subsets of the total data. Such program simply needs to act as efficiently as possible on multi-dimensional arrays (normally 2 physical dimensions each with 3 colour dimensions). This is a common form of parallelism, implemented in applications such as parallel Fourier transforms, simulation of crystalline matter and database analysis. Image processing, a subset of computer vision, normally falls under this category.

More complicated elements of computer vision fit task parallelism better. This is because they require several independent tasks to be completed.

Downton[19] discusses the use of pipeline processing farm in computer vision. This is similar to a task farm, but has a continuous flow of data to process. By parallelising independent tasks, the latency of image analysis can be reduced. This would reduce the delay between the image being input, and the result being output. This paper was written in

1994 and does not take into account modern technology when discussing potential uses. The advent of multi-core camera phones broadens potential implementations beyond the encoding algorithms and handwriting recognition discussed by Downton.

A current trend in parallel computing is to use accelerators to improve performance. Graphical Processing Units (GPUs), are the most common choice, due to their good performance/price ratio. Fung[20] implements the GPU based acceleration for several computer vision techniques including feature detection. The features are detected using Harris detectors, which are often not used in favour of SIFT-like keypoints. The paper does not discuss the details or the benefits of implementing parallel Harris detectors.

Parallel implementations exist for many algorithms, such as SIFT and Speeded Up Robust Features (SURF), discussed further in section 3.5.2. Yimin Zhang[21] implements two forms of parallelism for SIFT, showing large increases in the amount of frames that can be calculated per second. At 640x480 pixels, the image size used to obtain these speedups is small in comparison to an average Where's Wally? puzzle. A full Where's Wally? image is typically larger than 2000x1000 pixels, which has 6 times the area. Current digital cameras, which might be used with everyday computer vision, typically offer images that are an order of magnitude larger. Digital cameras frequently list their resolutions in the 10s of mega-pixels, which is a 32 times increase in pixels. Despite speedups, the size of these images might inhibit analysis at a useful speed.

Expanding on previous work, Zhang et. al. discuss in depth the effects that limit scalability [22]. The paper shows the purely serial portion of the code takes up less than 2% of the runtime. This shows that reduced scalability is due to more complicated factors, which should be avoided by novice users.

Nan Zhang presents the multi-core implementation of parallel SURF[23]. Within, Zhang shows that the multi-core implementations can have speed comparable to GPU implementations. Low-end computers (and by extension, cheap mobile devices) are suggested to lack the quality of GPU that can implement high speed algorithms. This implies that CPU based implementations are preferable for widespread usage.

The existing parallel implementations of SIFT/SURF algorithms are not directly available. This adds complexity to the project, which aims to maintain simplicity to ensure a wide development base.

2.7 Shared Memory Parallelism

This project will implement parallelism through the shared memory multiprocessing API known as OpenMP.

Shared memory is a region of memory that can be accessed equally by several processors. Normal examples of this are in multi-core systems, which have a shared L2 or L3 cache. Having shared memory allows data to be quickly shared amongst processors. This removes redundant and temporally expensive copying of data from main memory, and reduces the overall cache usage. Using less of the available cache means that a larger amount of other values can be stored, also reducing calls to main

memory. In a similar way, messages can also be passed between processors at high speeds. Communication through a local cache is considerably faster than typical messaging systems. Most recently manufactured computers are multi-core, so it is likely that everyday devices will be able to benefit from shared memory parallelism.

OpenMP implements shared memory multiprocessing through the use of directives and routines. A directive is a compiler flag that informs the compiler that the user intends for a specific behaviour to occur. These are simple to include into a program, often requiring little to no changes to a serial program to parallelise it. Listing 1 shows the simplicity of implementing parallelism.

Listing 1: 'A simple loop parallelised with OpenMP'

```
// a simple serial for loop
int x[4];
for(int i=0; i<4; i++) {
    x[i] = i;
}

// and again parallelised with OpenMP
int y[4];
#pragma omp parallel for default(none) shared(y)
for(int i=0; i<4; i++) {
    y[i] = i;
}
```

There are some issues that come with the use of shared memory multiprocessing. The main one is that scaling is often poor. This is due to the fact that there is a fixed amount of processors attached to any block of shared memory. Once the system is scaled beyond this amount, the program starts to become dominated by the speed of message between the two memory systems. Furthermore, the speed at which the CPU can write to the memory is limited. Increasing the number of processors attempting to write to memory through the same CPU causes a bottleneck to occur.

Another popular form of parallelism is through message passing. This is passing messages between processors. In this way data can be shared between processors, allowing for processors to interact. Unlike shared memory parallelism, message passing protocols typically do not rely on shared memory. Messages are instead sent over the bus, meaning that the processors being used can be in different nodes.

One of the most commonly implemented message passing standards is the Message Passing Interface (MPI). MPI scales to very large numbers of processors. However, converting existing code for use in MPI is time consuming, often considerably increasing the maintenance requirements for the code. Listing 2 shows one potential way to parallelise the same loop from listing 1 using MPI.

Listing 2: 'A simple loop parallelised with MPI'

```
// a simple serial for loop
int x[4];
```

```

for(int i=0; i<4; i++) {
    x[i] = i;
}

// and again parallelised with MPI
int y[4];
int rank;
MPI_Status status;
MPI_Comm_rank(&rank);
if(rank == 0) {
    y[0] = 0;
    MPI_Recv(&y[1], 1, MPI_INT,0,0,MPI_COMM_WORLD, &status);
    MPI_Recv(&y[2], 1, MPI_INT,0,0,MPI_COMM_WORLD, &status);
    MPI_Recv(&y[3], 1, MPI_INT,0,0,MPI_COMM_WORLD, &status);
} else {
    MPI_Send(rank,1, MPI_INT,0,0, MPI_COMM_WORLD);
}

```

MPI requires experienced developers to produce efficient code. Untrained developers may have difficulty implementing an appropriate model of parallelism for a given problem. Users of a parallel computer vision system may have to produce custom functions as they go. The complexity of MPI restricts the amount of users who could generate new definitions. This opposes the idea of an everyday use of parallel computer vision. Expertise would be required to tailor parallel computer vision to each new problem.

OpenMP is a good fit for implementing everyday parallel computer vision. Existing code requires only a small amount of modification, and basic parallelism does not require much expertise in parallel techniques. The scalability issues associated with shared memory parallelism are unlikely to be prominent in everyday systems.

3 Algorithms

This section discusses the algorithms used to implement object recognition. These are broken up into three sections

- Colour Analysis
 - Region detection
 - Line Width Estimation
- Shape Analysis
- Feature Recognition
-

They require more in-depth discussion than the other topics in colour analysis. Special focus is given to the algorithms that will be used for solving Where's Wally puzzles.

3.1 Colour Analysis

The functionality of analysing the colours of an image is critical for object recognition. Colour extraction is one of the primary ways of analysing the information contained by colour. Manipulation of pixels, such as blurring or sharpening an image, is a useful technique.

3.1.1 Colour Extraction

Extracting specific colours is one of the most important techniques in colour analysis. In raster images, extracting shades of the primary colours (red, green and blue) is a trivial task. Colours are stored as combinations of these colours, so extracting the specific values is simple.

For non-primary colours, a technique is needed to clearly describe them. A commonly used notation is the hexadecimal format, also known as a hex triplet. Values are stored in the form '#RRGGBB', where RR, GG and BB are the hexadecimal values for the red, green and blue components of a pixel. Examples of this can be seen in table 3.1.


Colour	Common Name	Hexadecimal
	White	#FFFFFF
	Black	#000000
	Crimson	#DC143C
	Sea Green	#2E8B57
	Orchid	#DA70D6

Table 3.1: An example of common colours with hexadecimal outputs

By finding the regions of the image that satisfy the red, green and blue colour values described in a hexadecimal value, specific colours can be located within an image.

This extends to searching for ranges of colours. This is useful, because images are regularly not in blocks of a single colour. For example,

gradients (one colour gradually shifting into another) are commonplace in many images. Attempting to highlight gradient with a single colour would reveal a small subsection of the desired area.

The ability to search ranges of colours enables the user to search more generally. If the precise colours used changes between images, using a colour range extends the generality of the function it is used in.

3.1.2 Blur

Another useful tool in colour analysis is blurring. Blurring allows the merging of nearby colours, or to help mitigate the effect of compression artefacts. Compression artefacts are visual distortions of the image, due to the intentional loss of data.

Two common methods of blurring are Median and Gaussian blurring. A median blur[24] causes a given pixel to take the median value of its neighbouring pixels. The 'size' of the blur describes the radius within which the median is calculated. Median blurs are often used for reducing the noise in an image, such as a photo taken in low light.

The Gaussian blur uses the Gaussian function (equation 3.1) to develop a weighted average of neighbouring pixels.

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (3.1)$$

This produces an image that appears to be blurred more smoothly than with the median blur, see figure 3.1. This is useful for Laplacian edge detection schemes, which is sensitive to noise.



(a) An image with no blur



(b) 4 pixel blur



(c) 8 Pixel blur



(d) 16 pixel blur



(e) Horizontal blur



(f) Vertical blur

Figure 3.1: Gaussian blur of an image

For this project, blurring will be done with Gaussian blurs. Features with fine detail, as regularly occurs in Where's Wally? puzzles, can lose important information with the median blur. For example, a line that is a single pixel wide could be completely removed by a median blur. In Where's Wally puzzles, a fine detail could be critical to locating Wally.

The Gaussian method avoids this, as a pixel in strong contrast with it's neighbour will maintain a level of contrast. As seen in section 4.2, Gaussian blurs are also useful for blending nearby masks. This is because Gaussian blurs will rapidly 'spread' binary values that are grouped together.

Blurring is not a trivially parallel task. For each pixel that is to be blurred, a large group of neighbouring pixels are required. Pixels on the border can often develop visual errors known as artefacts. Decomposing the image into subimages could produce artefacts on interior borders. To avoid this, each subimage would need a halo of data that it could read but not write to. The size of the halo would be dependent on the size of the blur being performed.

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.36	0.00	0.00
0.00	0.36	1.00	0.36	0.00
0.00	0.00	0.36	0.00	0.00
0.00	0.00	0.00	0.00	0.00

a An image to be sharpened

0.00	0.00	0.13	0.00	0.00
0.00	0.11	0.30	0.11	0.00
0.13	0.30	0.62	0.30	0.13
0.00	0.11	0.30	0.11	0.00
0.00	0.00	0.13	0.00	0.00

c The blurred image

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	1.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00

b An ideally sharpened image

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.17	0.00	0.00
0.00	0.17	1.00	0.17	0.00
0.00	0.00	0.17	0.00	0.00
0.00	0.00	0.00	0.00	0.00

d The sharpened image

Table 3.2: The process of sharpening an image. The arrays represent images in the process of sharpening. The sharpened image is obtained by subtracting the blurred array from the original with equal weighting. The result was then rescaled

3.1.3 Sharpen

The sharpen technique is another useful tool for colour analysis. Sharpen is intended to allow an image to be made more focused[25]. This is very useful for when details in an resized image have been blurred or compressed.

Sharpening an image is, in many ways, the opposite of blurring an image. Noting this, the most common method of sharpening, known as "unsharp mask" creates a blurred copy of the image, and remove it from the original using a weighting. Table 3.2 shows this process numerically. With an appropriate choice of weighting, the ideally sharp image can be obtained using the unsharp mask.

The unsharp mask method is sufficiently simple to implement for any developer.

Sharpening, like blurring, requires some message passing when parallelised. Though subtracting the weighted blur can be done independently, blurring subimages will lead to artefacts at the borders on the main image.

3.2 Region Detection

Region detection is a tool that allows the discrete labelling of connected pixels in a binary image. To the human eye, the connected regions of a binary mask are obvious. However, the masks does not explicitly contain information about the connectivity of it's pixels. Thus the tasks is not simple computationally.

3.2.1 Naive Solution

A computationally naive way to do detect regions is shown in Figure 3.2. Each non-zero pixel is assigned a unique integer value. Each pixel in the image is assigned the maximum value of itself and it's four nearest neighbours. This is repeated until the image is stable and no pixel

changes value. This method takes the maximum value of neighbouring pixels, and ignores non-zero pixels, so maximal values can not be spread outside of a region's boundary.

Within a region, it is evident that every pixel will have the value of the maximum pixel within that region. As each pixel has a unique value, it follows that each regional maximum must also be unique. The number of regions can be found by counting the unique values in the matrix. Similarly, properties of a region can be calculated by analysing pixels that have the same value as each other. For example, the mean position of the pixels within the region, the size of the region, and the bounding box around the region can be calculated this way.

This method is suitable for shared memory parallelism, but is not efficient. To determine if a pixel should have its value changed, 4 other pixels must be read. This must be done an indeterminate number of times, as stability is dependent on the arrangement of the mask.

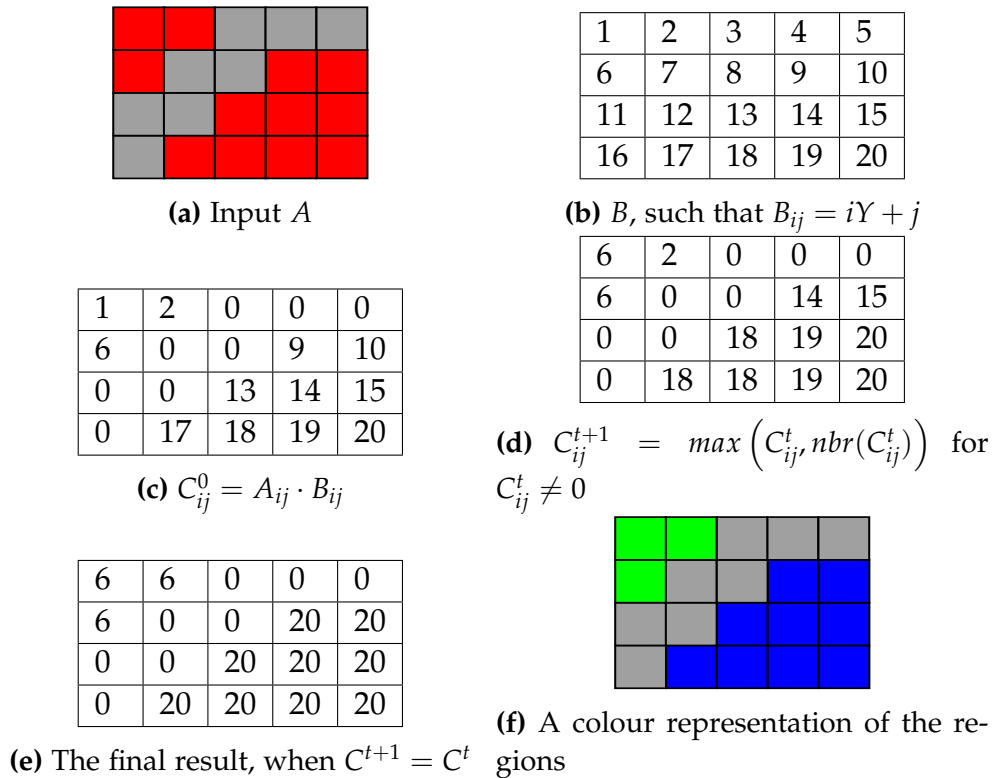


Figure 3.2: Steps in a naive region detection algorithm

3.2.2 Flood Fill

Flood fill[26] is an algorithm commonly used in 'paint' programs for filling an indicated region with colour. The algorithm can be implemented in several ways. The simplest to understand is the recursive version.

The recursive algorithm is initialised with a starting pixel. Pseudo code can be seen in listing 3.

Listing 3: 'Pseudo code for the flood fill algorithm'

```
void flood_fill(int mask[][][], int x, int y, int region_array[][]
    if (is_zero(mask[x][y] || region_array[x][y] == region_number)
        return ;
```

```

    } else {
        region_array[x][y] = region_number;
        flood_fill(mask, x+1, y, region_array, region_number);
        flood_fill(mask, x-1, y, region_array, region_number);
        flood_fill(mask, x, y+1, region_array, region_number);
        flood_fill(mask, x, y-1, region_array, region_number);
        return;
    }
}

```

```

int mask[10][10];
int region_array[10][10];
flood_fill(mask, 0,0, region, 1);

```

The flood fill algorithm can also be implemented in a queue based system (last in, first out), which avoids recursion. A graphical example of the flood fill algorithm can be seen in figure 3.3.

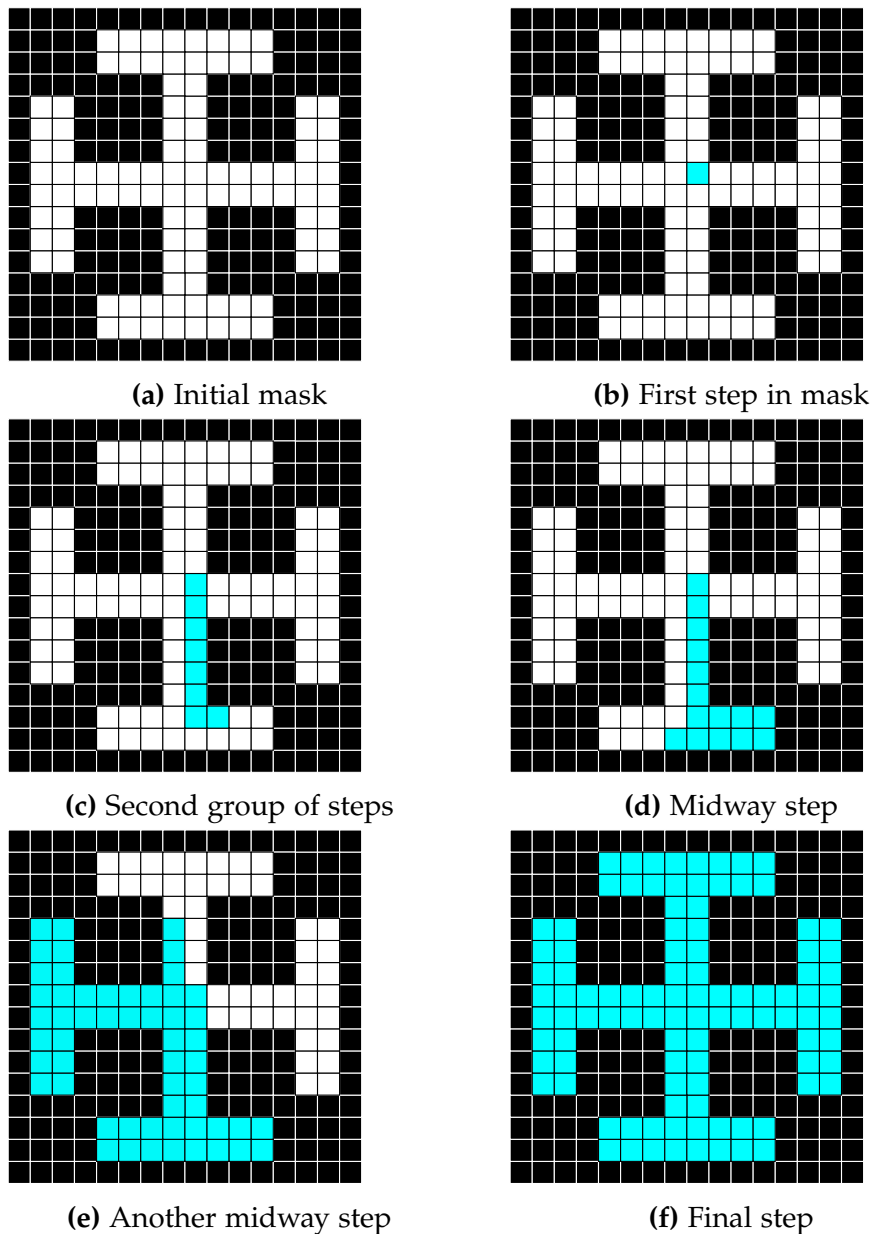


Figure 3.3: Various steps in the flood fill algorithm

The flood fill algorithm is useful when a hint is available for a good starting point. When trying to detect all the regions in a mask, the starting point needs to be automatically calculated.

Flood fill can be parallelised, but needs halo data to be passed. When the image is decomposed, regions could be split along a subimage border. This means that halo data must be analysed to match regions that are equivalent.

3.2.3 Connected Component Labelling

Lifeng [27] describes the method of connected-component labelling.

The algorithm, seen graphically in figure 3.4, works as follows;

Listing 4: Pseudo code for the Connected Component Labelling algorithm

```

int region_count = 0;

for(int i=0; i<rows; i++) {
    for(int j=0; j<cols; j++) {
        int north, west;
        if(i==0) { north = 0; }
        else { north = array[i-1][j]; }
        if(j==0) { west = 0;}
        else { west = array[i][j-1]; }

        if(west > 0 \&\& north > 0 \&\& west != north) {
            list_equivalence(west, north);
        }

        if(north > 0) { array[i][j] = north; }
        else if (west > 0) { array[i][j] = west; }
        else { array[i][j] = region_count; region_count++; }
    }

    merge_equivalent_regions ();
}

```

A pixel $p_{i,j}$ that has zero-valued or non-existent neighbours $p_{i-1,j}$ and $p_{i,j-1}$ is assigned a new temporary label. Otherwise, if the upper pixel $p_{i,j-1}$ is non-zero, it has a label (thanks to the ordering of the algorithm). The pixel $p_{i,j}$ is then assigned with the label of $p_{i,j-1}$. If the label of $p_{i,j}$ has not been set, then it gets the label of the non-zero left pixel $p_{i-1,j}$. If $p_{i-1,j} = p_{i,j-1}$ but they do not have the same label, then a label equivalence is established. Once the matrix has been traversed, equivalent labels are merged, and all regions have been detected.

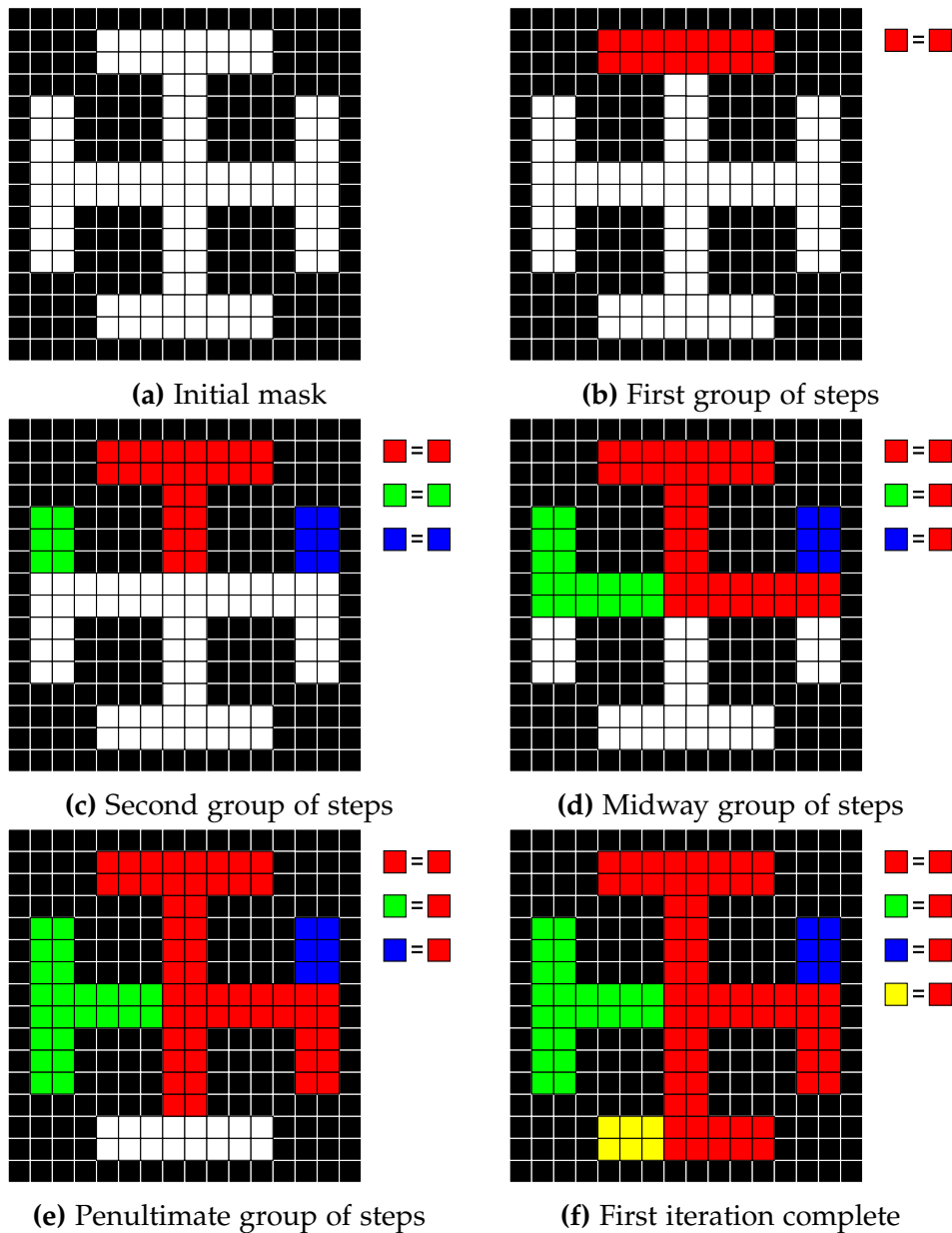


Figure 3.4: The steps in the first iteration of the connected component labelling algorithm

This method is both simple and memory efficient enough to be suitable for parallelism in this project. The algorithm maintains correctness on decomposed images, as long as halo data is passed between the processors. To find label equivalence across threads, only a single row or column is needed to be read. Between threads, the process is one-directional; data is only required to travel north and west of the current thread. This can be seen graphically in figure 3.5. In the case of shared-memory parallelism, accessing the necessary data is a minimal overhead. Critically, this process is largely the same as the serial version of the code. This provides a simple platform to enable parallelism.

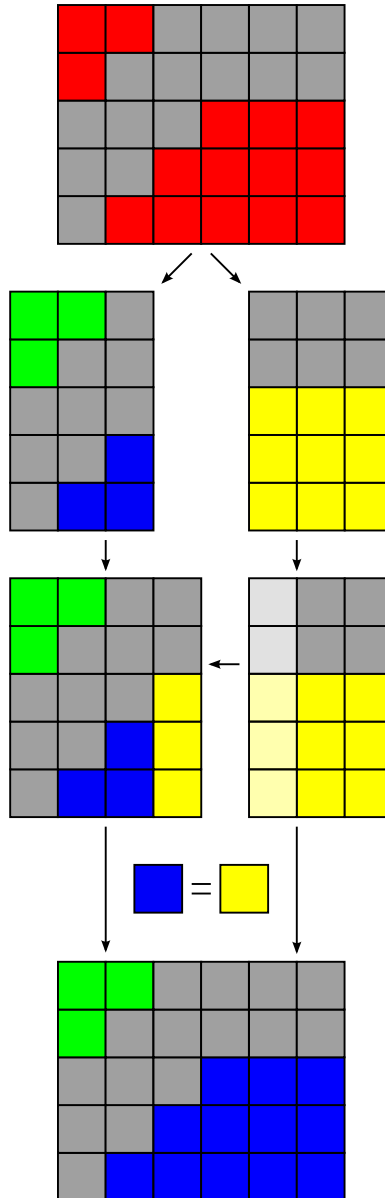


Figure 3.5: A diagram explaining parallel region detection. The initial image is split into two sections, and each region inside the subimage is given a unique id. One thread sends halo data to the other, which then calculates what threads are the equivalent. The main image is then put back together, with the parallel region equivalences applied.

Connected component labelling will be used for region detection in the Where's Wally? solver. It maintains simplicity under parallelism when compared to the flood fill algorithm. It is considerably more efficient than the naive solution. The naive solution requires 4 elements of data for every pixel analysed, and each pixel must be analysed until stability is reached. In a worst case scenario, this is an $O(n^2)$ problem. Connected component labelling is $O(n)$ for all cases.

3.3 Line Width Estimation

In cartoon images, characters and objects are regularly bounded by black lines. Within a given image style, these lines can be taken as a reference point for the scale of the image. For example, take one image with lines

that are 4 pixels wide and another with lines that are 2 pixels wide. It can be inferred that the first image is twice the scale of the second. This can be used to put an upper limit on how large a match between an Object and a Scene can be. Estimating the line width can be done using a combination of a few techniques.

For images with no intersecting lines, only a few steps need be followed.

- Produce a mask of all the black in the image.
- Count the distance to the nearest zero-valued pixel (called zero-distance hereafter). This can be done with the OpenCV `distanceTransform` function.
- Find the maximum zero-distance in the image.

Black lines are chosen here, due to their regularity as a boundary. Other colours can be similarly implemented.

In the example image, figure 3.6(a), a 7 pixel wide line has a maximum zero-distance of 4. For an 8 pixel wide line, it follows that the zero-distance would be 4. The maximum zero-distance can resolve line width of simple images to within 1 pixel.

For images with intersecting lines, this method will fail to produce correct results. Figure 3.6(b) shows a situation where the method fails. Two orthogonal 3 pixel wide lines cross over each other, creating a maximum zero-distance of 4. This would indicate that the line width of this image is 7 or 8, which is wrong by nearly a factor of 3. Images with large regions of black shading would cause this method to fail with larger errors.

A more complicated method, described in section 3.3.1, relies on two statistical properties of the image.

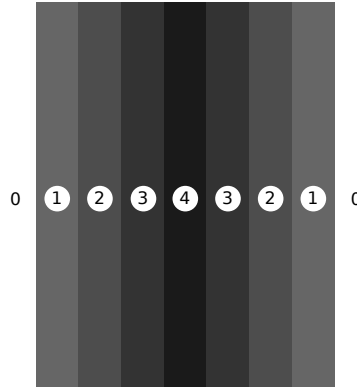
3.3.1 Average Distance to a Zero Valued Pixel

Inspecting figure 3.6(a), it is clear that the average distance is dependent upon something approaching a sum of incremental integers. Lines with even and odd widths will differ slightly; even values have two maximum zero-distances, odd values only have one. This can be seen in equations 3.2, where `PixelDistances(n)` lists the zero-distances in an n pixel wide line.

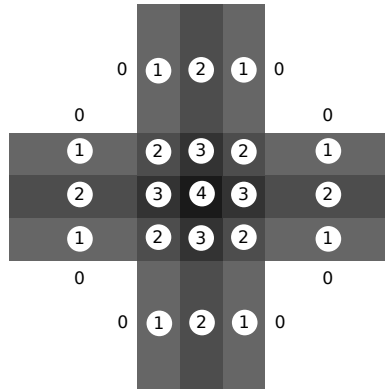
$$\begin{aligned} \text{PixelDistances}(4) &= \{1, 2, 2, 1\} \\ \text{PixelDistances}(5) &= \{1, 2, 3, 2, 1\} \\ \text{PixelDistances}(6) &= \{1, 2, 3, 3, 2, 1\} \end{aligned} \quad (3.2)$$

For a general N -pixel wide line, the sum of all values is in the range $[0, N/2]$ on one side, and $[N/2, 0]$ on the other. This has the mathematical form of a geometric series. This particular geometric series can be conveniently expressed as simple formula, described in equation 3.3.

$$\text{Sum}(N) := \sum_{i=0}^N i = \frac{N(N+1)}{2} \quad (3.3)$$



(a) A representation of a line that is seven pixels wide, broken down into reach region with different zero-distances



(b) Two intersecting lines, that are 3 pixels wide. The point of intersection produces a maximum zero-distance of 4

Figure 3.6: Two examples of lines with their zero-distances calculated.

Even valued widths are considered first, as this is the simplest case to analyse. This can be calculated as twice the sum of integers in the range $[0, N/2]$, as seen in equation 3.4.

$$\begin{aligned}
 \text{EvenSum}(N) &:= 2 \text{Sum}(N/2) \\
 &= 2 \sum_{i=0}^{N/2} i \\
 &= 2 \frac{(N/2)(N/2 + 1)}{2} \\
 &= N \left(\frac{N}{4} + \frac{1}{2} \right) \\
 &= \frac{N^2 + 2N}{4} \tag{3.4}
 \end{aligned}$$

For an odd-valued N , the sum is in the range $[0, (N + 1)/2]$ and $[(N - 1)/2, 0]$. This can be reduced to twice the sum of integers in the range

$[0, (N - 1)/2]$ plus $(N + 1)/2$, shown in equation 3.5.

$$\begin{aligned}
\text{OddSum}(N) &:= \frac{N + 1}{2} + 2 \text{Sum}((N - 1)/2) \\
&= \frac{N + 1}{2} + 2 \sum_{i=0}^{(N-1)/2} i \\
&= \frac{N + 1}{2} + 2 \frac{((N - 1)/2)((N - 1)/2 + 1)}{2} \\
&= \frac{N + 1}{2} + \frac{N - 1}{2} \frac{N + 1}{2} \\
&= \frac{N + 1}{2} \frac{N + 1}{2} \\
&= \frac{N^2 + 2N + 1}{4}
\end{aligned} \tag{3.5}$$

These values are very close, differing by only $\frac{1}{4}$ of a pixel. When these formulas are used to calculate the average zero-distance, this difference further decreases. Equation 3.6 demonstrates this. The estimation formula loses reliability for $N < 1$. As pixels are positioned on a discrete grid, the only possible value for $N < 1$ is zero, which represents no line.

$$\begin{aligned}
\text{AvgDist}(N) &:= \begin{cases} \frac{\text{EvenSum}(N)}{N} = \frac{1}{4} (N + 2) & \text{if } N \text{ even} \\ \frac{\text{OddSum}(N)}{N} = \frac{1}{4} (N + 2 + \frac{1}{4N}) & \text{if } N \text{ odd} \end{cases} \\
&= \frac{N + 2}{4} + O(N^{-1})
\end{aligned} \tag{3.6}$$

For large values of N , the average distance approaches the EvenSum formula. This can be seen in figure 3.7. The background shading represents the values bounded by the values of OddSum and EvenSum. The figure also shows the maximum error for any given value of N is at most a quarter of a pixel.

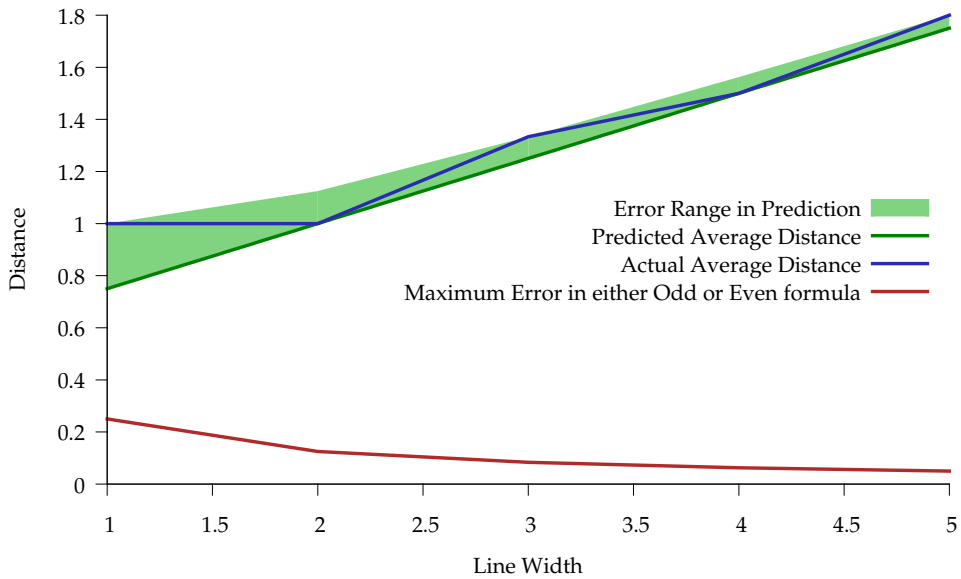


Figure 3.7: Graph showing the estimation of the average distance to a zero valued pixel using the Odd and Even formulae.

Equation 3.6 shows that there is an approximate linear relationship between the average zero-distance and the width of the line drawn. It is possible to invert this equation. This produces a formula, equation 3.7, to calculate line width from the average zero-distance.

$$\text{LineWidth}(avg) := \text{AvgDist}^{-1}(avg) = \frac{avg - 0.5}{0.25} \quad (3.7)$$

Using this equation, the line width can be approximated using an easily calculable property of the image; the average distance to a zero-valued pixel. The maximum error in equation 3.6 was a quarter of a pixel. Equation 3.7 accordingly has a maximum error of 1 pixel (at $N = 1$). The formula has a minimum error of 0 for all even valued widths. An unusual effect of this is that the method cannot determine a difference between a line of width 1 pixel and a line of width 2. This means that this method can be used only as an upper bound on the scaling between two images.

3.3.2 Standard Deviation in Distance to Zero Valued Pixels

The line width can be deduced from another calculable property of the system; the standard deviation. Standard deviation, here, is defined to be the normalised deviance of all pixels from the average position of each pixel, x_i .

$$\text{StdDev}(N) := \sqrt{\frac{\sum (x_i - \text{AvgDist}(N))^2}{N}}$$

The sum, here called Deviance within the square root can be simplified.

$$\text{Deviance}(N) := \sum_{i=0}^N (x_i - \text{AvgDist}(N))^2 = \begin{cases} \text{DevEven}(N) & \text{if } N \text{ is even} \\ \text{DevOdd}(N) & \text{if } N \text{ is odd} \end{cases}$$

As before, x_i will take values from $[0, \dots, N, \dots, 0]$, and $\text{AvgDist}(N)$ is defined in equation 3.6. The method for finding the standard deviation for even valued widths will be shown, odd values follow from above. The even equation is written as

$$\begin{aligned} \text{DevEven}(N) &= 2 \sum_{i=0}^{N/2} \left(i - \frac{N+2}{4} \right)^2 \\ &= 2 \sum_{i=0}^{N/2} \left(i^2 - 2i \frac{N+2}{4} + \frac{(N+2)(N+2)}{16} \right) \\ &= 2 \left(\sum_{i=0}^{N/2} (i^2) - 2 \frac{N+2}{4} \sum_{i=0}^{N/2} (i) + \sum_{i=0}^{N/2} \left(\frac{(N+2)(N+2)}{16} \right) \right) \\ &= 2 \left(\sum_{i=0}^{N/2} (i^2) - \frac{N+2}{4} \frac{N(N/2+1)}{2} + \frac{N}{2} \frac{(N+2)(N+2)}{16} \right) \end{aligned} \quad (3.8)$$

The sum in equation 3.8 is, as with equation 3.3, easily expanded using geometric identities. In this case, the formula is described in equation 3.9.

$$\sum_{i=0}^N i^2 = \frac{1}{6} N(N+1)(2N+1) \quad (3.9)$$

Continuing the expansion of $\text{DevEven}(N)$;

$$\begin{aligned}\text{DevEven}(N) &= \frac{1}{3} \frac{N}{2} (N/2 + 1)(N + 1) - \frac{N(N + 2)(N + 2)}{4} + \frac{N(N + 2)(N + 2)}{16} \\ &= \frac{1}{48} (N^3 - 4N)\end{aligned}\quad (3.10)$$

Using this formula for DevEven , the even width standard deviation becomes

$$\begin{aligned}\text{StdDevEven} &= \sqrt{\frac{\text{DevEven}(N)}{N}} \\ &= \sqrt{\frac{N^3 - 4N}{48N}} \\ &= \frac{N}{4\sqrt{3}} + O(\sqrt{N})\end{aligned}\quad (3.11)$$

Using similar calculations for odd widths, StdDevOdd has a value of

$$\begin{aligned}\text{StdDevOdd} &= \sqrt{\frac{\text{DevOdd}(N)}{N}} \\ &= \sqrt{\frac{N^2}{16N} + \frac{N^2 - 3N + 2}{48N}} \\ &= \frac{N}{4\sqrt{3}} + O(\sqrt{N})\end{aligned}\quad (3.12)$$

As these functions are equivalent (up to $O(\sqrt{N})$), the Standard Deviation approximation is defined as

$$\text{StdDev}(N) := \frac{N}{4\sqrt{3}}\quad (3.13)$$

Figure 3.8 shows the comparison of these formulae with actual data. The data and approximate standard deviation are well bounded inside the Odd and Even standard deviations. The maximum error that can be expected from the approximation is less than 0.3 pixels. This is at the boundary case of 2 width pixels. At all other points, the error is a small fraction of the actual value.

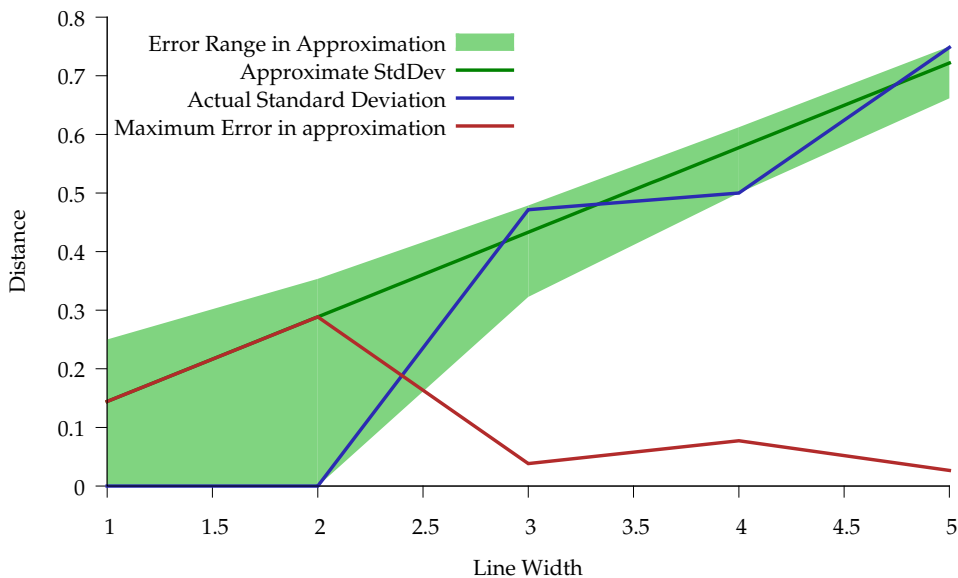


Figure 3.8: Graph showing the relationship between line width and the Standard Deviation of pixel distance to zero valued pixels.

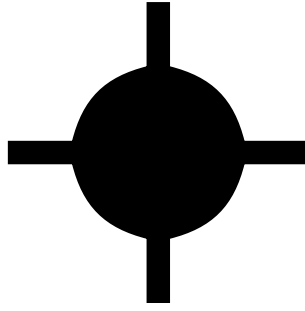
Again, as a linear formula has been produced, the relationship can be reversed, to get line width from standard deviation.

$$\text{LineWidth}(\text{stddev}) := \text{StdDev}^{-1}(\text{stddev}) = 4\sqrt{3} \cdot \text{stddev} \quad (3.14)$$

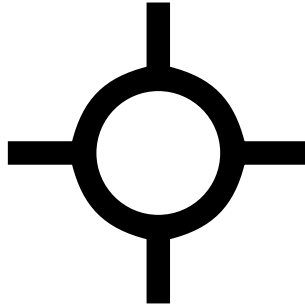
Equation 3.14 can not give an exact estimation of line widths; it will overestimate odd widths and underestimate even widths. For the correct standard deviation, it will give the correct line width to within ± 1 pixel, for all values. Thus the standard deviation method can put an upper bound on the scaling between two images.

3.3.3 Combining the Methods

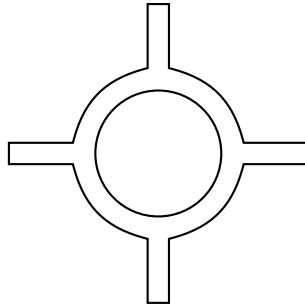
Each of these methods is flawed when introduced into an image with overlapping lines, as in figure 3.6(b). This can be compensated for by successively removing the pixels with the largest distance from the map. At some point during this removal, it is expected that a "sane" mask will be produced. A sane mask is one that displays only the boundary lines of an image. For example, a filled in square would be reduced to the 4 border lines that define it's shape. A visual example can be seen in figure 3.9. The sane mask should produce the most correct line width estimations from either formula. Determining which mask is the sane one is not immediately obvious, but can be deduced using a combination of both methods.



(a) The original image, with an equation distorting solid block of black in the center



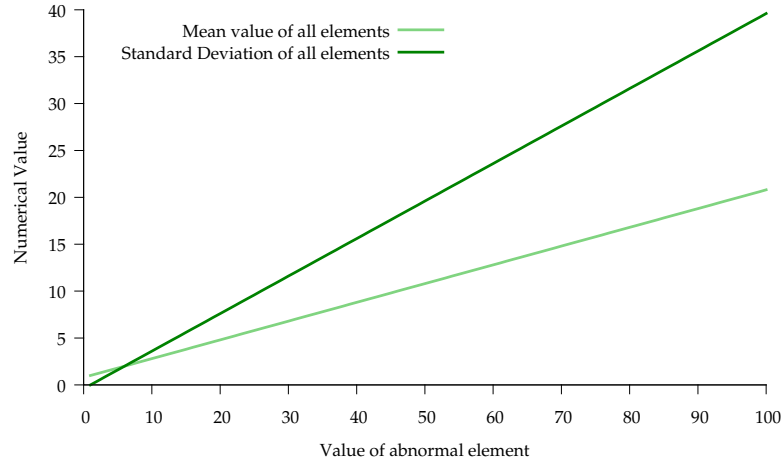
(b) The same mask, replacing solid circle in the center with it's border



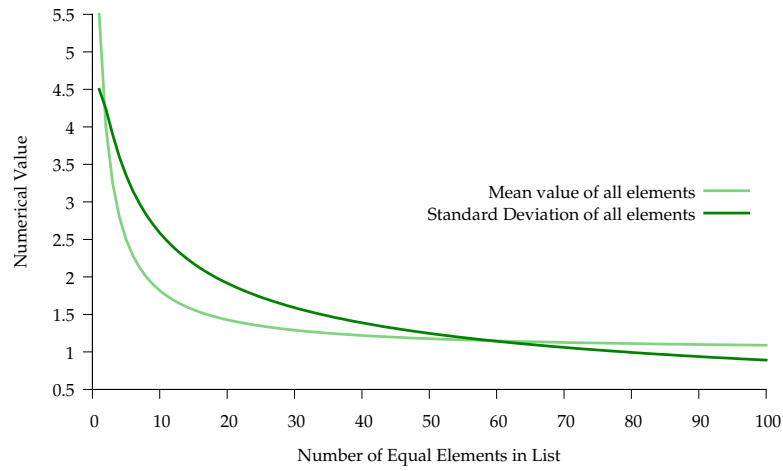
(c) The image after too many maxima have been removed, which would also distort the equation

Figure 3.9: A representation of the removal of the highest zero-distance pixels, to reveal a sane mask

Each method reacts differently to the removal of the largest zero-distance pixels. This can be seen in figure 3.10, the mean and the standard deviation have very different responses. These differences allow a combination of the two methods; only when they agree can a mask be described as sane.



(a) Changing the value of a single abnormal element in a list



(b) Changing the ratio of normal data to abnormal data

Figure 3.10: The response of mean and standard deviation to various types of erroneous data

It is unlikely a perfectly sane mask will be located. In this case, the mask with the minimal difference between the two estimations of line width will be the sane mask. The average of the two methods will be the line width estimation used. The line width is reliable to the nearest integer, rounded up, except in the case that the line width is 1 pixel. This is because a 1 pixel width line is indistinguishable from a 2 pixel width line, using the mean and standard deviation. To demonstrate this, we define Z_n as the list of zero distances of each pixel in the slice of an n wide pixel.

$$Z_1 = \{1\}, \text{Mean}(Z_1) = 1, \text{StdDev}(Z_1) = 0$$

$$Z_2 = \{1,1\}, \text{Mean}(Z_2) = 1, \text{StdDev}(Z_2) = 0$$

This being the case, a 1 pixel line may be interpreted to be a 3 pixel line. Using this method, a maximum line width can be estimated. Comparing these line widths, the scaling between two cartoon images can be approximated.

3.3.4 Testing

This method was tested with an image with several thickness. The base image contains vertical lines that are increasingly further apart from each other, with lines connecting each point. This can be seen in figure 3.11. There are two types of test, solid lines, which have clearly defined boundaries, and aliased lines, which are more realistic. The results of using this method of each of these images is listed in table 3.3. The line widths are almost all correctly predicted to within 1 pixel, with the exception of 1 pixel on a solid line. As discussed earlier, this has been anticipated.

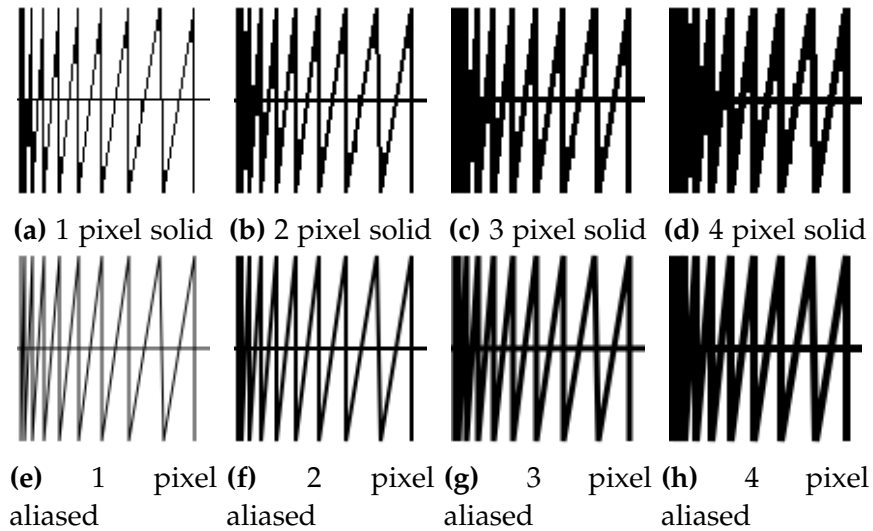


Figure 3.11: Images used to test `estimate_black_line_thickness`. They are designed to stress estimation, with lines that overlap regularly. The top row are solid pixels, with clear boundaries. The bottom row show aliased lines, which are more realistic.

Line Width	Solid Line Predicted Width	Aliased Line Predicted Width
1	2.42688	1.31487
2	2.74886	2.75531
3	3.05320	2.87646
4	4.21705	4.33824

Table 3.3: A comparison of the line width analysis technique on images from 3.11 All but one of the values are within 1 pixel of the actual value. The anomalous 1 pixel result can be mistaken as a 2 pixel width. The technique cannot guarantee distinction between 1 and 2 pixels.

3.3.5 Parallelism

Analysing the line width is a parallelisable task. Calculating the zero-distance of each pixel in the map needs to be done in serial. This is because bisecting a line could make a large change to the standard deviation, if there aren't many lines. The bisection can't easily be avoided without knowing the width of the lines, so using halo data is not viable. However, splitting the mask after the zero-distance has been calculated is parallelisable. The average can be calculated with a reduction operation, as can the summation of the standard deviation. The overhead of using

OpenMP for small images is likely to reduce usefulness of parallelism. Large images may benefit from this, as OpenMP is initialised only once for a larger proportion of data.

3.4 Shape Analysis

Shape analysis is a computer vision technique used to analyse shapes contained in a binary mask.

Suzuki[28] discusses two closely related techniques for border following. This is the method of 'following' edges that are defined in a binary image. By following these edges and recording the path taken, all shapes defined within the mask can be located. As the recorded path contains a large number of points, it may be preferable to simplify the shape before analysis.

The Ramer-Douglas-Peucker algorithm[29][30] can be used to approximate curves. This algorithm removes points on the curve that don't sufficiently alter the shape of the line. The algorithm, which is recursive, can be seen below

1. Begin with first and last points on a curve, A and B .
2. Locate the point, C , that is furthest from the the line that joins the two points.
3. If it is greater than ϵ away from the line, then it contributes importantly to the shape of the line, and is kept. Otherwise the point does not contribute, and can be removed.
4. The curve AB is now split into two curves, AC and BC . Repeat steps 1-4 on these curves until all curves have been approximated

This function is computationally expensive, as it is recursive and requires large numbers of perpendicular distance calculations for each point calculation. The complexity of this algorithm is, in worst case scenarios, $O(n^2)$ [31]. For large images, which would contain large numbers of complex shapes, this could dramatically increase the runtime.

OpenCV has an inbuilt function, `findContours`, which implements Suzuki's border following. It has an implementation of the Ramer-Douglas-Peucker algorithm, `approxPolyDP`.

3.5 Feature Detection

There exists a large number of algorithms for locating features[2][32][33][34]. Two of the most prominent algorithms, SIFT and SURF, will be analysed in detail. These were considered more thoroughly than the others, because OpenCV provides implementations that are ready to use.

3.5.1 Scale-Invariant Feature Transform

Scale Invariant Feature Transform (SIFT)[2], as discussed in section 2.3.1, is an algorithm to detect keypoints in an image. These keypoints are chosen as to be invariant under changes in scale, translation and rotation. They are designed to be partially invariant to affine changes and varying illumination.

Each keypoint has a corresponding descriptor, which is a feature vector. A feature vector attempts to describe an area around the keypoint, helping to create a unique identifier.

Direct keypoint matching provides a very reliable method of finding a given Object within a larger Scene image. Some of the techniques required to produce reliable results require unexpected amounts of memory. To recreate the image in scale space, four versions of the image must be produced, one of which is scaled to be twice the size of the original.

This is not a problem when analysing a single image, or analysing multiple images linearly. If the algorithm is used for analysing multiple images concurrently, memory constraints could limit its efficiency. This can be seen with Zhang's parallel implementation of SIFT [21], where the scale-space creation is one of the areas the program spends most time on. This paper shows reasonable parallel efficiency, with results up to 32 cores, far beyond the goal of this course. The paper only solves 5 images at a time, which gives a 7 times speed-up over an optimised version of SIFT. This number of images is not large enough to show the true limits of shared memory parallelism.

3.5.2 Speeded Up Robust Features

Speeded-Up Robust Features (SURF), developed by Bay et. al.[32], offers a similarly robust solution, but with greater efficiency. This algorithm replaces Laplacian of Gaussian filters used in SIFT with box filters, which calculate in constant time, once an integral image has been produced. It has greater potential for parallelism than SIFT; versions of the image for the scale space that need to be calculated are independent of the previous level and can be done in parallel.

In some cases, SURF can have less reliable results than SIFT. As such this project will use SIFT.

SIFT and SURF both lack the direct means of actually matching keypoints. It is left to the user to make use of the invariance provided with each keypoint.

3.5.3 Matching: Brute Force Matcher

Brute force matching is the standard 'naive' implementation of keypoint matching. It matches Scene keypoints by comparing the corresponding Scene descriptor with all Object descriptors. The brute force matcher then returns the keypoint with the best match.

This is a reliable, if slow method of detecting matching keypoints. It is also deterministic, which is useful for getting repeat results.

3.5.4 Matching: FLANN

The Fast Library for Approximate Nearest Neighbours[35] (FLANN) matcher is a matcher that outperforms the brute force matcher for large images. The library automatically chooses an approximate nearest neighbour algorithm depending on the nature of the dataset. All of the algorithms share one thing in common; they do not guarantee optimum matches

between keypoints. Instead, they provide a large (Lowe claims order of magnitude) speedups, by selecting matches randomly.

Often this is done with some variation on an approximate kd-tree. The k-d (as in 3D) tree allows the user to partition a high dimensional problem, and enables efficient searching. Approximating this can provide large speedup.

The downside to this is that results are not deterministic. This can be an issue if properties of the system are sensitive to the parameters of given results.

For direct comparison of descriptors, the brute force matcher will be chosen. This is because this project will be tested for correctness, and noisy results could inhibit measurement. Both matchers are available through OpenCV, so future users would be able to choose the FLANN matcher, if determinism wasn't important.

3.5.5 Matching: Relative Keypoint Distances

A potentially more flexible method of detecting Wally can be found in relative keypoint distances. This records the structure of an object in a matrix containing the relative distances between all keypoints. This matrix could be stored within the program, for comparison with a similar matrix generated from the Sene's keypoints.

Well matched keypoints correspond to rows of each matrix 'matching'. Matched rows contain the same elements in the same order. A group of keypoints matched this way would strongly imply that the Object was present in the Scene.

Comparing the two rows from different matrices was an interesting problem. Matching will occur between matrices of different size. A good match here would imply that a row of a small matrix is a subset of a row from the large matrix. Extraneous keypoints could appear inside an otherwise ideally matched row. Rows could also match well, without matching perfectly. Thus some method was desired for comparing the rows without requiring that one be a contiguous subset of the other. This is possible through the naive method of creating a list of all ways a row can match. There exists a better solution.

The problem appears similar to locating the ordered intersection of two unordered multisets {groups of numbers with non-unique elements}. We act on the order that each element appears in the intersection, as the value itself does not hold any particular meaning. This means that our unordered sets can now be described as ordered.

$$\begin{aligned}
 A = \{0, 1, 2\}, B\{0, 1, 2\} &\rightarrow \{0, 1, 2\} \\
 A = \{0, 2, 1\}, B\{0, 1, 2\} &\rightarrow \{0, 2\}, \{0, 1\} \\
 A = \{2, 1, 0\}, B\{0, 1, 2\} &\rightarrow \{0\}, \{1\}, \{2\} \\
 A = \{0, 1, 2\}, B\{0, 1, 2, 3, 4\} &\rightarrow \{0, 1, 2\} \\
 A = \{0, 1, 2\}, B\{0, 1, 2, 1, 0\} &\rightarrow \{0, 1, 2\} \\
 A = \{0, 1, 2\}, B\{0, 2, 1, 2\} &\rightarrow \{0, 1, 2\}
 \end{aligned}$$

For this problem, we only need to know the size of the largest ordered intersection, referred to here out as the matching number. Worth noting is that the values not held within the Object matrix have no effect on the matching number and can be removed.

Finding the largest ordered intersection is equivalent to determining the disorder in a set. Consider a perfectly ordered set of unique numbers, which is the product of removing extraneous data from the Scene matrix. We define $M(A \hat{=} B)$ as the matching number of two sets A and B .

$$A := \{1, 2, 3, 4\} \quad B := \{1, 2, 3, 4\}$$

$$M(A \hat{=} B) = 4 = \text{Size}(A \cap B)$$

The largest matching number is clearly the size of the set. Introducing non-unique values, but maintaining the ordering, we can see that the matching number remains unchanged.

$$B = \{1, 2, 2, 3, 4, 4\}, M(A \hat{=} B) = 4 = \text{Size}(A \cap B) - \text{Size}(A \setminus B)$$

We permute any two elements of the set.

$$B = \{4, 2, 2, 3, 1, 4\}, M(A \hat{=} B) = 3 = \text{Size}(A \cap B) - \text{Size}(A \setminus B) - \text{Disorder}(B)$$

Multiple permutations increase the disorder, and decrease the matching number, as expected.

Algorithmically, this is as simple as counting the number of strictly ordered neighbours in the Scene row that also appear in the Object row. This means that determining how well matched two rows are is an $O(n)$ problem, where a naive solution could be $O(n!)$, which is exponential complexity.

4 Implementation

The Where's Wally? solver was implemented in C++. This was due to familiarity with the language, it's availability in OpenCV and the automated memory handling it provides.

The program was written in a modular fashion, with the understanding that it could be extended easily. One aspect of this involves implementing a blueprint `Search_Pattern` class, which contains basic information needed for any search pattern. By extending this, and defining it's virtual functions, new search patterns can easily be added to the program.

The program has three main sections, the framework, I/O and the search patterns. I/O parses command line input, and displays the requested forms of output, including results, graphical displays and timings. Graphical output was done using OpenCV's image display command, `imshow`. The framework section handles the flow of data, as well as the analysis of results before they are sent to I/O. Search patterns are the varying methods used to find Wally, and are called by the framework to analyse the input image.

Described within the following sections are the functions required to implement the search patterns, and the search patterns themselves.

4.1 Function Implementation

Before the patterns that locate Wally can be put together, some algorithms must be implemented. Listed below are these algorithms and the technicalities to their implementation.

Get Colour in Image A function was developed to extract a specific range of colours from an image. Two hexadecimal colours were required, as well as 6 colour ratio numbers. The ratio numbers made sure that each combination of colour was within a certain ratio. The first two numbers represent the ratio between red and green

$$R \geq rg \cdot G$$

$$G \geq gr \cdot R$$

Similar relationships exist for the other combinations of colours, using rb, br, bg and gb . In this way, it is possible to define colours generically. For example, blue colours can be defined as ones with high bg and br ratios. The extraction of a yellow image can be seen in figure 4.1.

This was designed to assist finding Wally's red stripes. Although his stripes are red, the precise shade cannot be known before hand. The ratio functionality allows this imprecision to be mitigated.

Parallelism was implemented by decomposing into vertical slices. To maximise potential parallelism, the images is sliced for each thread running in the program. This allow the function to make full use of any available threads.

Get Greyscale in Image A similar function was developed for finding greyscale in images. Technically, the colour extractor is equally

capable of doing finding grey in an image. This interface can be simplified for greyscale images.

This function defines a range of colours between two numbers, in the range of 0-255. It also defines a tolerance for colourfulness. This allows the location of nearly grey colours, which is useful for finding whites or blacks that have been blurred on a boundary.

Parallelism was implanted in the same way as the colour extractor.

Estimate Black Line Width As discussed in section 3.3, a line width algorithm has many uses. This was implemented, as mentioned, using OpenCV's `distanceTransform` function.

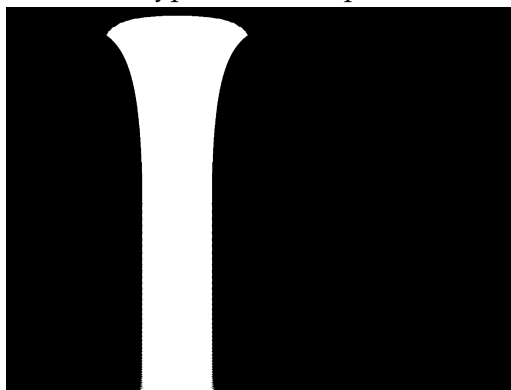
The estimation was parallelised through the decomposition of the image into slices. To maintain equivalent correctness with the serial case, calculating the zero-distance should be done within a critical section. However, calculating the zero-distance for a subsection, in general, will not affect the calculated distance by more than 1 pixel. As this function is intended to give resolution within a pixel, the decrease in accuracy is worth the increase in speed.

Find Regions in Image Using the Connected-Component Labelling algorithm from section 4, a region detection function was created. The region equivalence list made use of the map functions, found in C++'s standard template library. This is a simple way to create such a list, allowing array-like syntax with non-sequential keys.

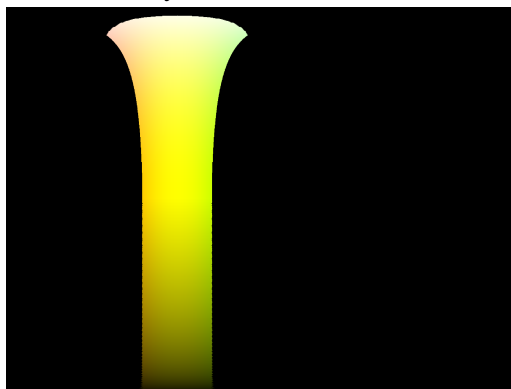
This function was parallelised by vertical decomposition. Boundary data was synchronised after the parallel region, as the memory was shared.



(a) A typical colour spectrum



(b) The mask produced by the function. White represents a pixel that was considered to be yellow.



(c) The mask overlaid on the original image, revealing "yellow" colours.

Figure 4.1: Isolating yellow from an image using `get_colour_in_image`

4.2 Pattern: Red and White Stripes

When trying to locate Wally, one of his most prevalent features is his red and white jumper. Few other elements of the puzzles use the colour scheme of red and white, and less use red and white stripes. Finding regions with red and white stripes is one of the most immediate ways to identify Wally. This can be done by following these steps, seen graphically in figure 4.2;

1. Create a mask that shows all white pixels in the image, figure 4.2(b)
2. Create a mask that shows all red pixels in the image, figure 4.2(c)
3. Blur or displace white mask in vertical direction, figure 4.2(d)

4. Blur or displace red mask in vertical direction, figure 4.2(e)
 5. Multiply binary masks together, producing an area that shows where the blurs overlap, figure 4.2(f)
 6. Create a binary version of multiplied mask, such that any non-zero value takes the 'on' value.
 7. Blur the binary mask in the vertical and horizontal directions, to merge nearby values.
- Optional. Repeat with horizontal blur and remove common elements from the vertical mask.

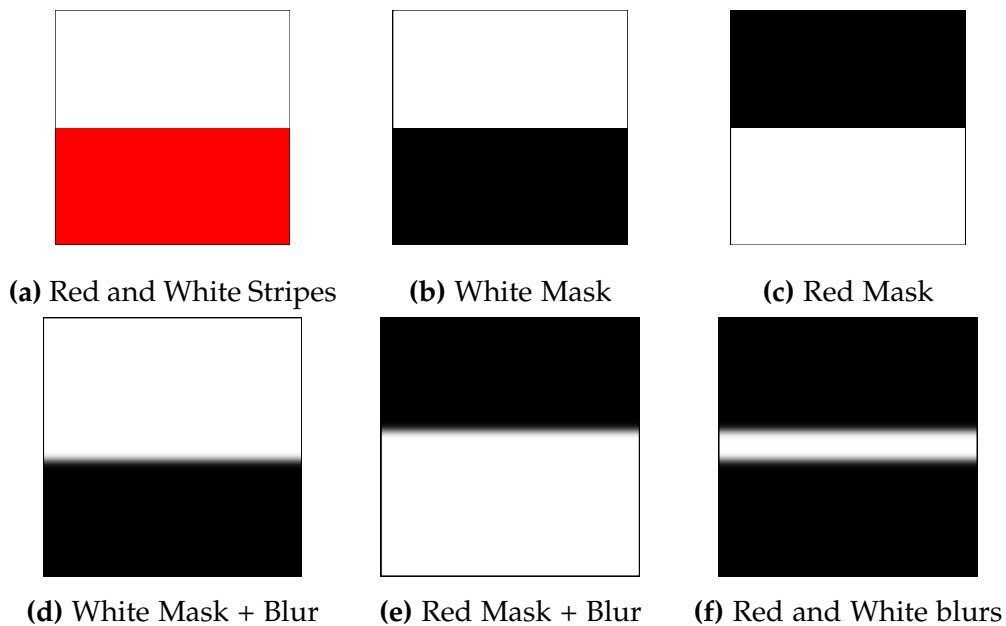


Figure 4.2: Example of finding red and white striped regions

Following the optional step allows the removal of non-horizontal stripes from the results.

Certainty, the likelihood that a given result is correct, is derived from the expected size of Wally. There is a ratio between the bounding box surrounding Wally, and the size of his jumper. Results that do not fit this ratio are removed from the results. Scale can be inferred from the width of the black lines in the image. The thickness of the lines implies a rough size to Wally. Certainties are taken from a Gaussian distribution centered about the estimated size of Wally.

4.2.1 Weaknesses

Finding Wally by his stripes is not wholly reliable. Although Wally is one of the few characters to wear red and white, he is not the only one. A good example of this is Wenda, figure 1.2(c), who wears a similar jumper. Furthermore, objects in the image regularly have a red and white motif, such as skirts, umbrellas and cakes. Without user intervention, it is hard to prioritise results based solely on this information. One method is to identify the largest regions of red and white stripes in the image as most likely to be Wally. This is often incorrect because of the presence of large red and white objects, see figure 4.3.



Figure 4.3: The Red and White Stripes pattern failing to find Wally. The blue ring indicates the top result (an umbrella). The green ring indicates Wally's actual location, which is not even ranked in the top 100 results.

One way to mitigate this is to remove 'horizontal' matches from the vertically blurred mask. Wally is normally found standing upright, and so his jumper normally is striped horizontally. Vertical stripes are unlikely to be from potential Wallys, so they are removed from the search. This involves repeating the same techniques required to find the regions originally, but with a horizontal blur. Pixels that match the horizontal blur can be removed from the original mask, preventing their inclusion in the region detection.

Good values for the blur used for merging nearby regions are dependent on the size of Wally's stripes. If the value is too small, Wally's jumper as a whole is never located. Too large and too many incorrect regions are included in the definition of Wally's jumper. The size of Wally's stripes are not a known property, so a guess must be made as to a good value. One way of doing this is by estimating the average line width of the image, and extrapolating the size of stripes from there. For low resolution images, however, the ratio of stripe size to line width varies wildly, making it hard to estimate correct values.

4.2.2 Testing

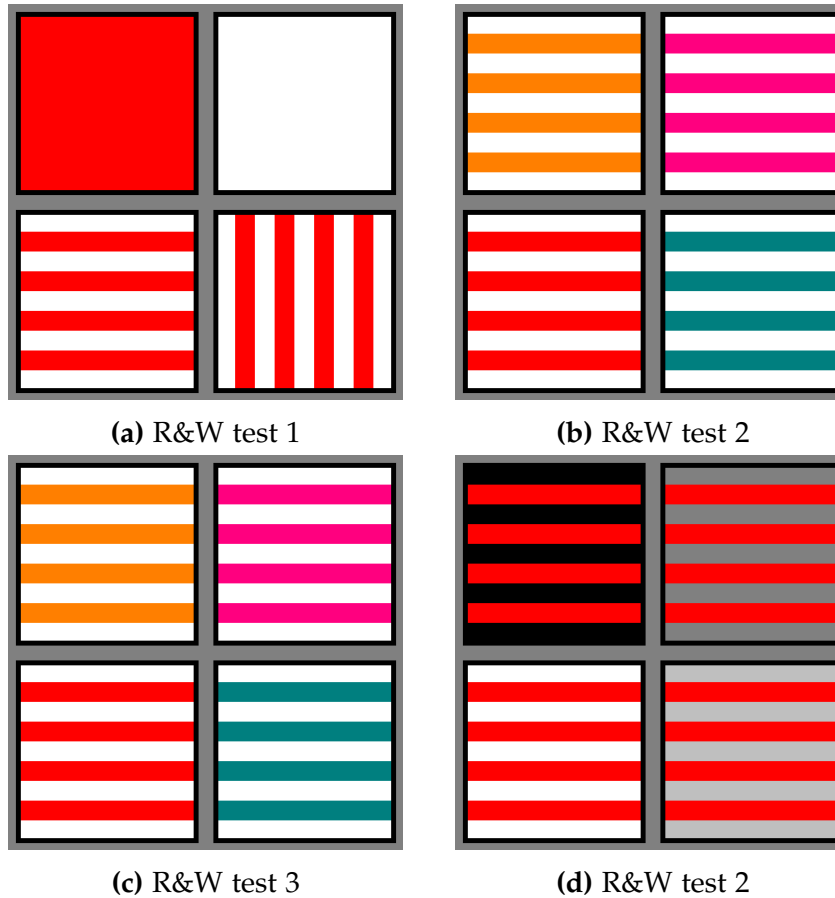


Figure 4.4: Test images used to check the red and white pattern

4.2.3 Parallelism

Finding red and white stripes should be very parallelisable. The image can be split into parallel subimages to find the red and white masks. The creation of the masks is a per-pixel operation, so requires no information from neighbours. This means that each parallel thread is entirely independent of its neighbours, reducing the need for synchronisation.

The vertical blurring only needs a small amount of information from any vertical neighbour, due to the vertical blurring. Choosing to decompose the image horizontally avoids the need for any halo data to be swapped. The optional stage in the algorithm can be done by switching the decomposition, or by simply ignoring the halo data. This is because the bulk of usable information should come from the vertically blurred mask.

As with the creation of the red and white masks, the matrix multiplication is per element. This again means that the problem can be decomposed into subimages to give large speedup.

4.3 Pattern: Blue Trousers

This pattern is much like the Red and White pattern discussed in section 4.2. It uses colour analysis to locate Wally's trousers, by searching for blue areas in an image.

This pattern shares the same weaknesses as the Red and White pattern. There is no guarantee that a correctly found pair of blue trousers will belong to Wally. This pattern is expected to be unreliable, as Wally's legs are regularly obscured.

Certainty will be defined in the same way as the Red and White pattern.

Any parallelism that occurs here is much the same as could occur in the use of the colour analysis functions.

4.4 Find Glasses

Finding Wally's glasses is done using shape analysis.

Making use of the `estimate_width_of_black_lines` function, the approximate size of Wally's glasses can be predicted. This is because the width of lines in a Wally image scale approximately with the rest of the size of the image. Using OpenCV's `findContour` and `approxPolyDP`, the contours in the image can be found and accessed. Contours that have an area that is much less or much greater than the predicted size of Wally's glasses can be discarded.

Approximately circular contours can be searched for by assuming that circular curves are formed like regular polygons. There is a well defined relationship for the interior angle between two points in a regular polygon. Thus we can define the irregularity of a curve; the average angular deviation from the angles of a regular polygon with the same number of points. Curves with high irregularity can be removed from the glasses candidate list. The list is populated with single circles. The next step is to find pairs of circles that are the same size, very close vertically, and within a specific range horizontally. That is, we look for pairs of circles that are arranged as glasses would be.

As noted in section 3.4, finding these contours can be an $O(n^2)$ problem. In large images, contours exist with high values of n that are very unlikely to be Wally's glasses. Furthermore, large images likely contain large numbers of contours. The circle matching step is also $O(n^2)$. It is desirable to keep these calculations as short as possible. Thus the image is decomposed into subimages that are twice as big as the prediction of Wally's glasses. This removes needless long range matching of circles pairs, and prevents overly large contours from becoming a problem.

As the program already decomposes the image into independent task, it is natural to parallelise this pattern.

The certainty will be defined by how much the two lenses of the glasses are similar. This is not a very efficient metric, Wally's eyes are expected to be slightly dissimilar.

4.5 Find Features

This pattern uses feature detection to locate Wally. As decided in section 3.5, the feature detection was done with the Scale-Invariant Feature Transform algorithm. Using these keypoints generated for a built-in list of Objects and the Scene, matches are produced. These matches are anal-

ysed, and good matches are used to create a homography. A homography is a matrix that defines a perspective change on a 2D surface.

$$\vec{b} = H \cdot \vec{a}$$

$$\begin{pmatrix} b_x \\ b_y \\ 1 \end{pmatrix} = \begin{pmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & H_{22} \end{pmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ 1 \end{pmatrix}$$

$$b_x = H_{00}a_x + H_{01}a_y + H_{02}$$

$$b_y = H_{10}a_x + H_{11}a_y + H_{12}$$

$$1 = H_{20}a_x + H_{21}a_y + H_{22}$$

By inspection, we can see that H_{03} and H_{12} represent the translational displacement. The elements H_{00} , H_{01} , H_{10} and H_{11} are the values for scale and rotation. The bottom row doesn't hold any information about the perspective change. We can infer that H_{22} should be equal to 1, and the values H_{20} and H_{21} should be 0.

Matches were made using the Brute Force Matcher supplied by OpenCV. The quality of matches was assessed by the relative distance between them. If the distance was within some multiple of the smallest relative distance, then the match was considered a good one.

An attempt was made to locate Wally using features without requiring a list of Objects. This was done by using the Relative Keypoint Distance technique discussed in section 3.5.5. The implementation of this was not flexible enough to provide a useful method of grouping keypoints.

The homography is an estimate of the perspective change between the Object and the Scene. The last values in the homography matrix can be used to develop the certainty of a match being Wally. A real perspective change will have $H_{22} = 1$ and $H_{20} = 0, H_{21} = 0$. Thus the certainty can be defined as something of the form

$$\text{Certainty}(H) = 1 - (H_{22} * (H_{20}^2 + H_{21}^2))^2$$

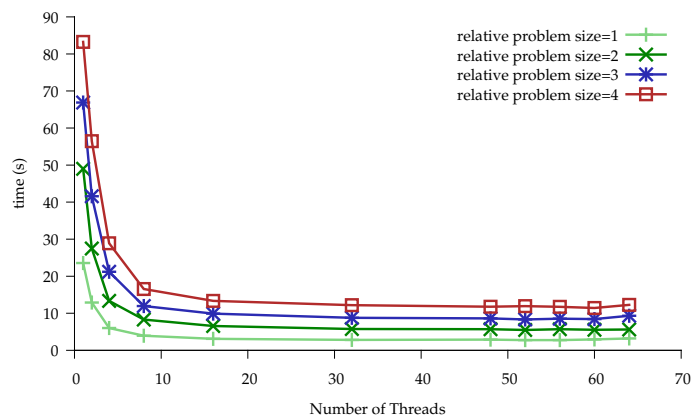
5 Results

For this project, it is important to obtain results for reliability as well as speedup. Functions that directly attempt to find Wally had their reliability tested for a variety of images. All tests for parallelism are conducted for various data sizes as well as core count. All timing results were measured on the Morar cluster.

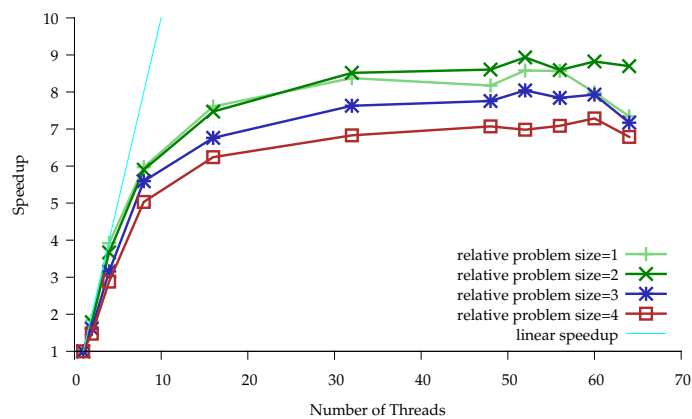
Reliability results are measured using scans from two Where's Wally? puzzle books; "The Fantastic Journey" and "The Wonder Book". The scans were scanned at the largest resolution available. Technical difficulties with highly uncompressed scans meant that the high resolution images were highly compressed, resulting in compression artefacts.

5.1 Extracting Greyscale from Image

The line width technique experienced good speedup to 8 cores, which is the maximum a home computer is expected to have. Figure 5.1 shows the results. The algorithm was repeated 100 times on a 2000x2000 pixel test image. The greyscale extractor is extremely similar to the colour extractor, so the similar speedup behaviour can be expected.



(a) Wall time



(b) Speedup

Figure 5.1: Time and speedup of extracting greyscale from an image 100 times

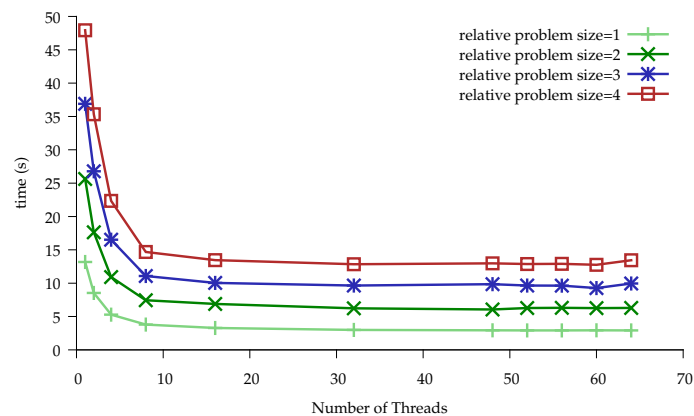
The greyscale analyser shows a speedup that fits well to Amdahl's laws. The solution time increases approximately linearly with problem size. The speedup is slightly higher for smaller images. This is because

it hasn't completely filled the available cache. This is inferred from the fact that problem sizes 1 and 2 share the same profile, but three and four have reduced speedup. This implies that somewhere between size 2 and 3, the cache is filled. The parallel slopes at the end of the graph are likely due to coincidental noise in the timing data.

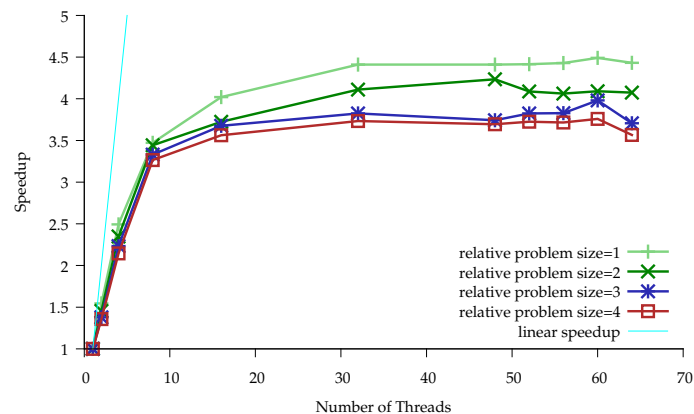
5.2 Line Width

The line width algorithm discussed in section 3.3 was implemented in parallel. The results can be seen in figure 5.2. The algorithm was repeated 100 times on variously sized sections of a 2000x2000 pixel test image.

The line width calculation has experiences a linear increase in wall time for an increase in work. However, the maximum speedup is larger for smaller problem sizes. This reflects the fact that there is more cache available for The speedup is efficient up to 8 cores, which is the largest number home computers can reasonably have. This speedup appears to fit the limits imposed by Amdahl's law. This is to be expected, as the zero-distance calculation is strictly linear.



(a) Wall time

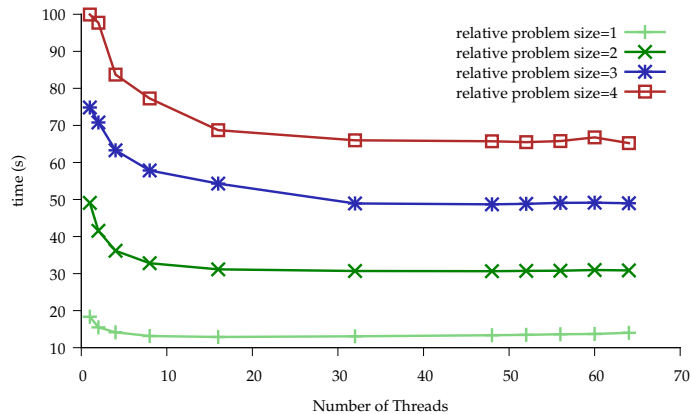


(b) Speedup

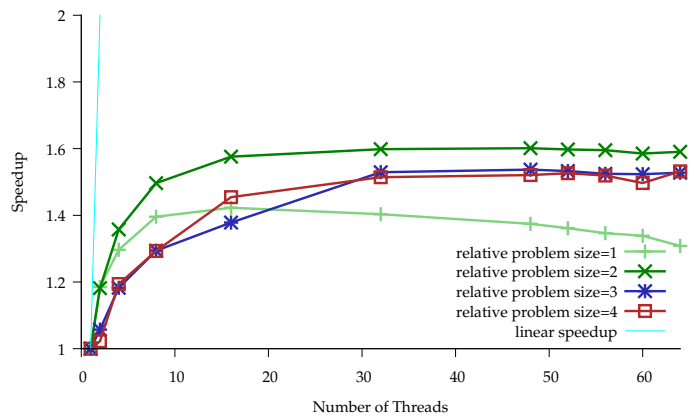
Figure 5.2: Speedup of an image's linewidth being estimated

5.3 Region Detection

The parallel implementation of region detection has poor speedup. Figure 5.3 shows that the maximum speedup achieved is less than 2. This is due to the relatively low level of parallelism in this function. The regions boundaries must be calculated in serial, and creating a full region equivalence map was done in serial. Better speedup could be obtained from utilising more parallelism in this section.



(a) Wall time



(b) Speedup

Figure 5.3: Speedup of an image's regions being detected

5.4 Find Features

Finding Features requires given Objects to search a Scene with. Object images were taken from "The Fantastic Journey", a set of Wally puzzles. The pattern is correctly able to recognise Wally in some situations, as seen in figure 5.4. Despite the apparent strictness of the algorithm, strange false positives can be produced, see figure 5.5. This is likely due to the low amount of keypoints that can be found in the relatively low resolution Object images. These false positives do not resemble Wally, and often highlight areas that do not contain people, obfuscating the results.



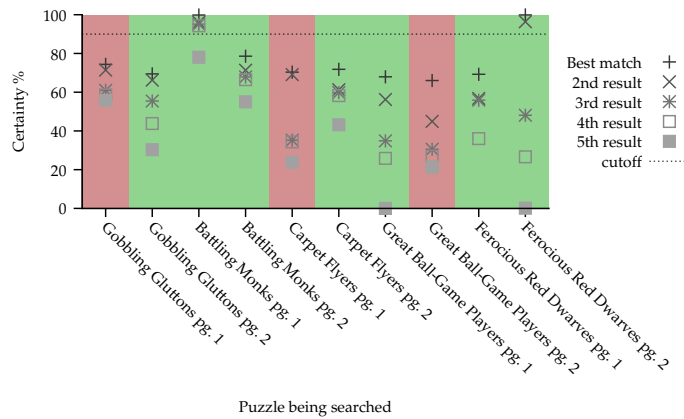
Figure 5.4: An example of correct output from the Find Features pattern



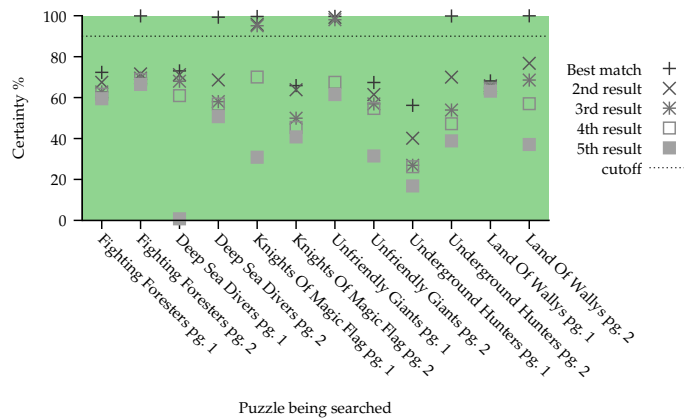
Figure 5.5: An example of the output of the Find Features pattern

5.4.1 Reliability

Figure 5.6 shows the results from this pattern, when using Object images that are known to be in the Scene.



(a) Results from the first half of "The Fantastic Journey" book



(b) Results from the second half of "The Fantastic Journey" book

Figure 5.6: The reliability of the Find Features Stripes pattern. Results are taken from "The Fantastic Journey", and each Object that is scanned for is taken from this book. A green background means that the match correctly discerned whether or not Wally was present. A red background means the opposite.

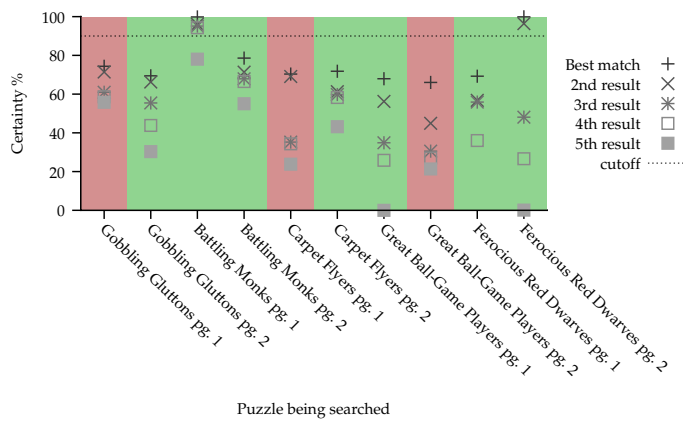
The pattern produces results that are correct for 87.5% of the pages in The Wonder Book. Wally only appears in half of those pages, so the algorithm gave 25% false negatives. The false negatives likely appear due to a number of factors. A major one is that some of the Object images are very low resolution, figure 5.7. These images will produce only a small number of matches. In an attempt to make the pattern as generic as possible, the backgrounds of each Object were removed. SIFT attempts to use boundaries to locate keypoints. In low resolution images, the useful keypoints may not exist on the interior of a Wally Object. Thus by removing the background, the boundary keypoints are lost and the Scene and Object cannot be usefully compared.

The correctness cut-off is very distinct. Values that are described as over 90% certain are always seen to show Wally correctly. This implies that the certainty algorithm could be made more strict.

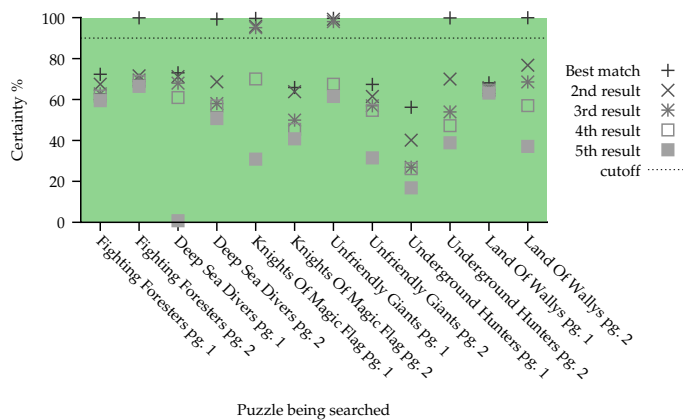
Figure 5.8 shows the set of Object images derived from "The Fantastic Journey", but in Scenes from "The Wonder Book". In this way, it was possible to test how well the pattern can be extended to arbitrary cases.



Figure 5.7: A comparison of Wally Resolutions from the same book. There is a large difference, the face in the second image is almost completely blurred.



(a) Results from the first half of "The Wonder Book" book

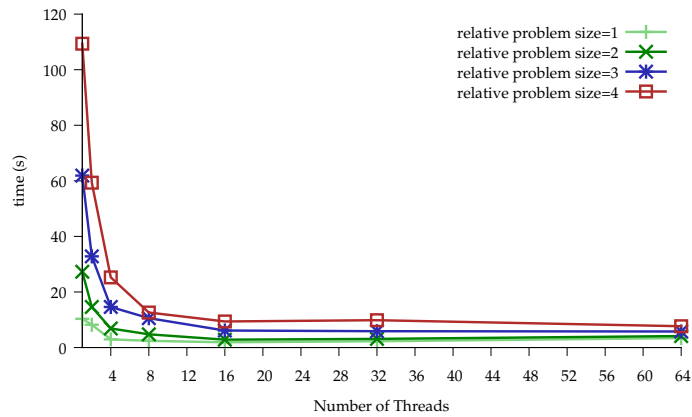


(b) Results from the second half of "The Wonder Book" book

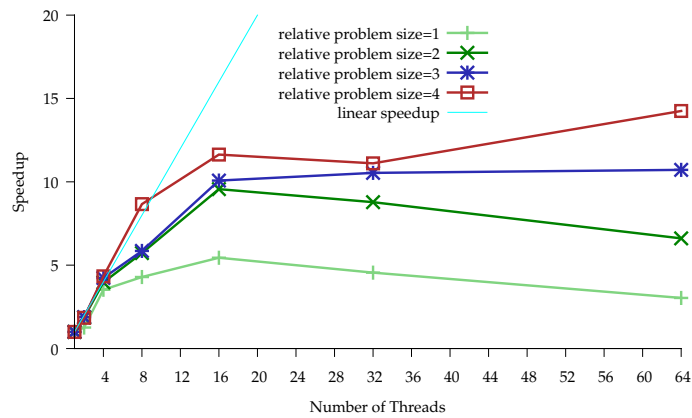
Figure 5.8: The reliability of the Find Features pattern with unknown Objects. These results were taken from the solutions of puzzles found in "The Wonder Book". This compares known Wally Objects from "The Fantastic Journey" and attempts to apply them to unknown objects.

5.4.2 Speedup

The speedup was measured using variously sized subsections of an image. The image was chosen such that a serial version correctly locates Wally. The results are shown in figure 5.9. Peak speedup is obtained at 16 cores. Behaviour after this point is dependent on the size of the problem. Large problems seem to continue, albeit at a markedly lesser efficiency. Reducing the problem size decreases the scaling, meaning for smaller images it is more better to use less cores. This is likely due to the increasing amount of communications and cache synchronisations caused by critical regions.



(a) Wall time



(b) Speedup

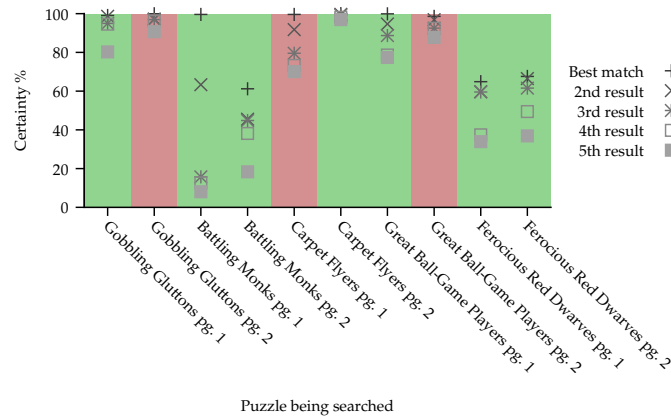
Figure 5.9: Time and speedup of the Find Features pattern

5.5 Red And White Stripes

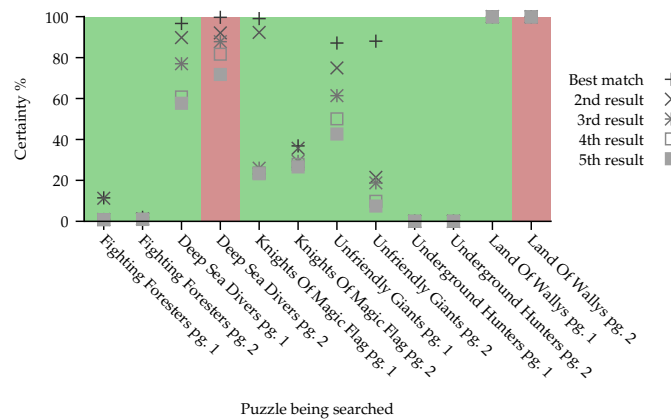
The results for the Red and White Stripes pattern are presented below. The pattern requires some tweaking to optimally find Wally. This makes producing a generic solver difficult. The colours between puzzle books changes slightly. Thus focusing in on the red found in Wally's stripes in one puzzle might lock them out in another. This can be seen in section 5.5.1, where the reliability is non-existent.

5.5.1 Reliability

When tuned to find Wally in a specific set of images, this pattern works well, as seen in figure 5.10. In images that contain low amounts of red and white, this pattern is capable of reliably finding Wally's lost hats. These are small red and white hats that are hidden through the puzzle. Conversely, images with large amounts of red and white stripes, such as the Deep-Sea Divers puzzle, complicate the problem considerably.



(a) Results from the first half of "The Fantastic Journey" book



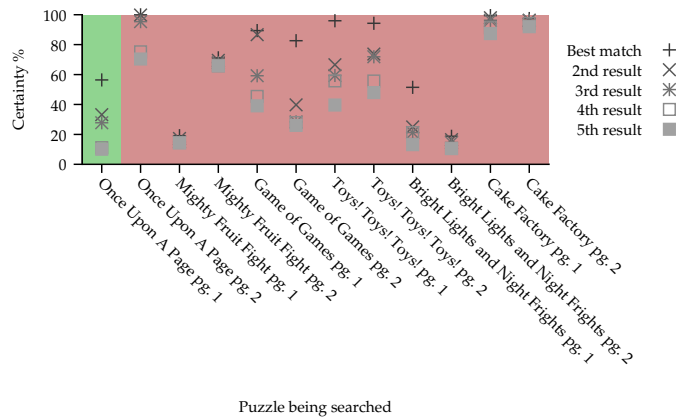
(b) Results from the second half of "The Fantastic Journey" book

Figure 5.10: The reliability of the Red and White Stripes pattern. Results are taken from "The Fantastic Journey", and the program was tweaked to find Wally more often. A green background means that the match correctly discerned whether or not Wally was present. A red background means the opposite.

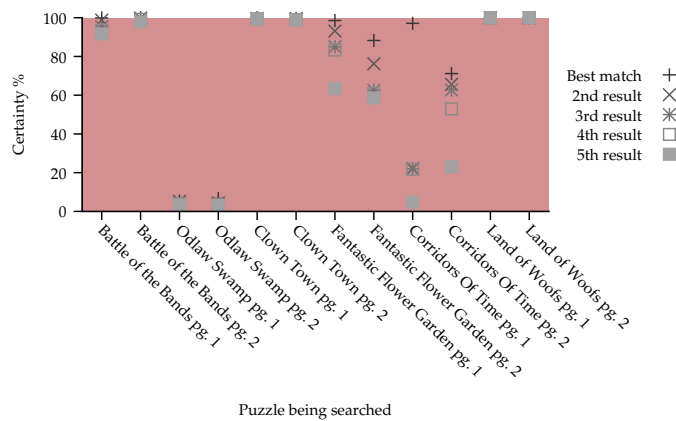
When the settings that are optimised for "The Fantastic Journey" are used to locate Wally in "The Wonder Book", any reliability is lost. This can be seen in figure 5.11. As can be seen in both figures, the indicated certainty has no bearing on whether or not Wally can be found in the image. The "Underground Hunters" in figure 5.10 shows the particular weakness of this certainty. Despite correctly highlighting Wally, the results are never more than 1% certain.

5.5.2 Speedup

The Red and White pattern achieves peak speedup at 8 cores. The speedup is not efficient, around 25% at peak. This is due to the prevalence of strictly linear code, such as the Gaussian blurs.



(a) Results from the first half of "The Wonder Book" book



(b) Results from the second half of "The Wonder Book" book

Figure 5.11: The reliability of the Red and White Stripes pattern. Results are taken from "The Wonder Book", and the optimal settings for "The Fantastic Journey" are kept.

The first problem size decreases in speedup because the parallel work done is outweighed by the communications needed to pass the data.

5.6 Find Glasses

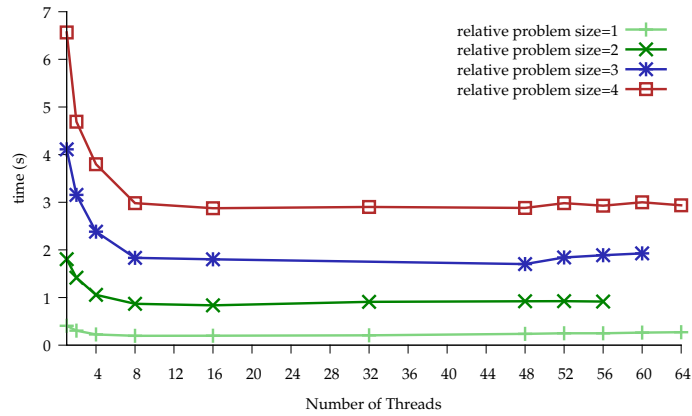
This method could not find Wally correctly. This was due to the large amount of paired circles in every image. These would often produce a higher certainty than Wally's glasses. A reliability chart has not been presented because no information is contained within.

5.6.1 Speedup

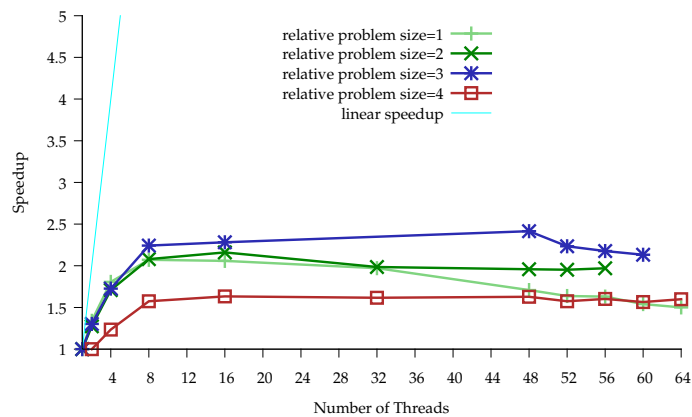
The Find Glasses pattern finds peak speedup at 32 cores. The speedup increases slightly for larger problem sets. The end of the data is noisy, but this is likely due to problems in the timing system of Morar.

5.7 Blue Trousers

As expected, this method has no ability to directly find Wally. Again, no reliability chart will be presented.



(a) Wall time



(b) Speedup

Figure 5.12: Time and speedup of the Red and White stripes pattern

5.8 Speedup

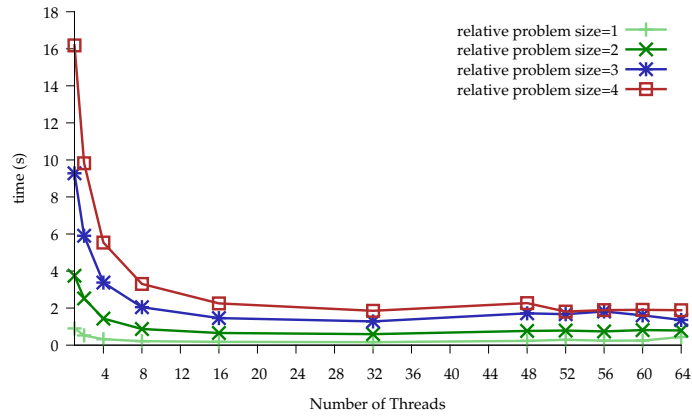
The speedup, seen in figure 5.14, is similar to that of the Red and White Stripes pattern. This is to be expected, as they are almost the same function.

5.9 Using All Techniques

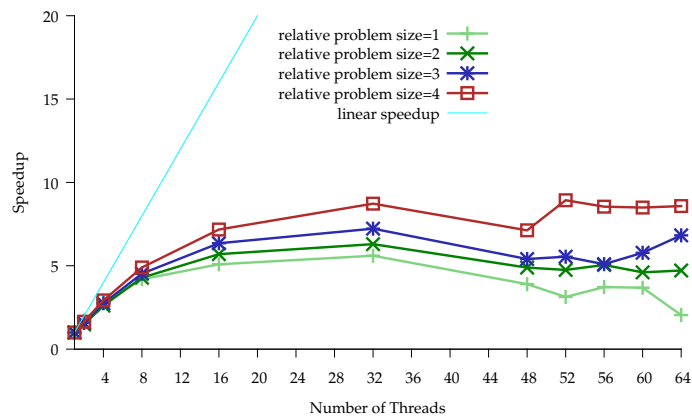
There will be no reliability analysis for this section. This is because the certainties from each pattern do not translate to each other. Thus, the reliability plots from before are full of noisy points that say nothing about the information.

5.9.1 Speedup

The program experiences super linear speedup at 4 and 8 cores, figure 5.15. This is likely due to timing errors in Morar, as the previous test of the Patterns do not show super linear behaviour. However, it is clear that there is good speedup at 16 cores, which is beyond the scope of regular computers. The speedup increases when the problem size is increased, which means that it is efficient to run large images. Worth noting, is that the running time is regularly dominated by a single pattern, either Find Features or Find Glasses. This means that more patterns could be used, or that a weighting of threads could improve the run time.



(a) Wall time

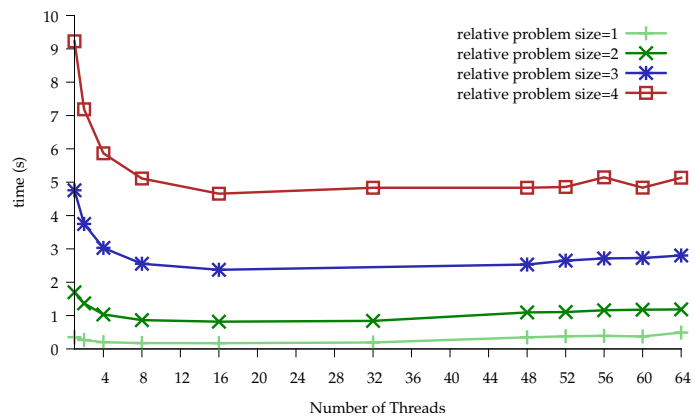


(b) Speedup

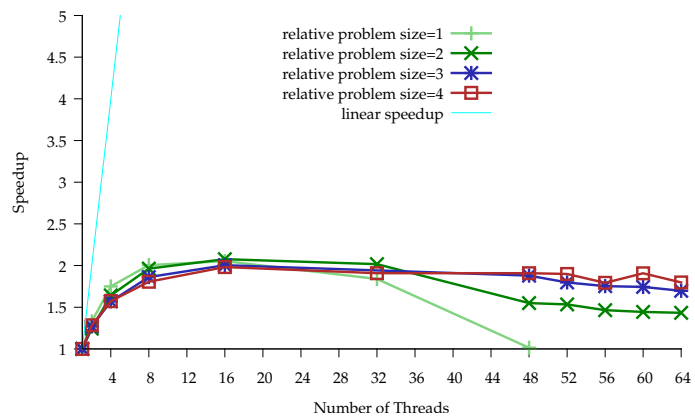
Figure 5.13: Speedup of the Find Glasses Pattern

5.10 Genericism

The solution was found to be capable of genericism. Implementing a "Yellow and Black" pattern to find Odlaw simply required copying the Red and White pattern, and changing the relevant ratios. The result can be seen in figure 5.16.

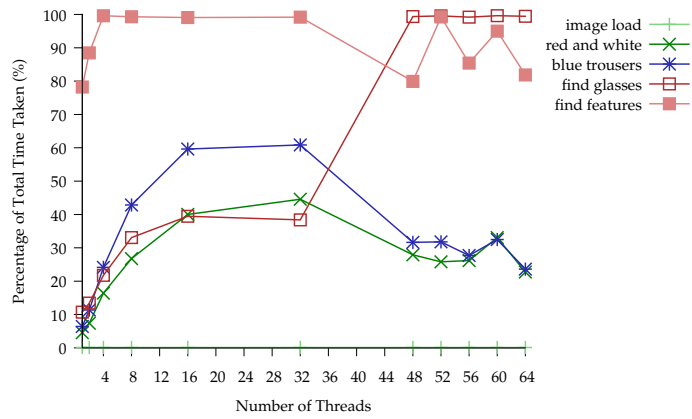


(a) Wall time

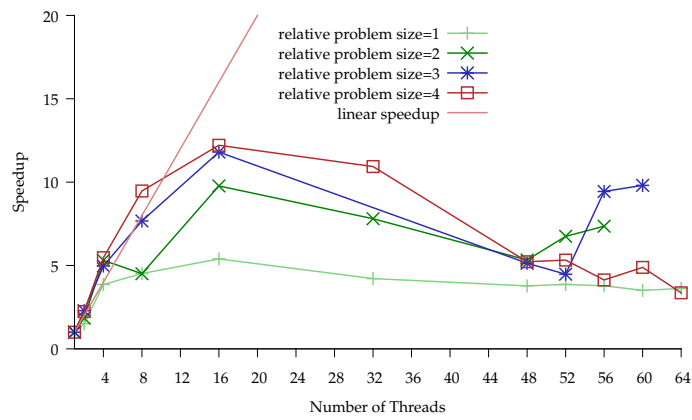


(b) Speedup

Figure 5.14: Speedup of the Blue Trousers Pattern



(a) Wall time



(b) Speedup

Figure 5.15: Speedup of entire program



(a) The Odlaw Character



(b) Odlaw as a result

Figure 5.16: Extending the program to find Odlaw

6 Conclusion and Evaluation

The goal of this report was to test the development of parallel computer vision for everyday use. Object recognition was selected as an appropriate topic within parallel computer vision for this. A Where's Wally? puzzle solver was created using readily available techniques and libraries, such as OpenCV. This was parallelised using OpenMP, allowing for an unintrusive inclusion of parallel techniques. The solver aimed to be generic, reliable and fast.

As seen in section 5, using parallel computer vision can improve the speed at which a complete solution can be found. In a limited time frame, it can also help to increase the accuracy of a solution by combining multiple techniques.

When tailored to a specific group of puzzles, the solutions found by the program can be very reliable. The feature recognition and colour analysis techniques were particularly reliable. The shape analysis performed could not specify clearly enough what a valid pair of glasses were, and so did not contribute considerably. Similarly, finding Wally by his blue trousers was not useful, as Wally's trousers are normally obscured.

When the program was not optimised for a set of puzzles, the reliability suffered. This prevented the program from being fully generic in it's goal, as it could not always reliably find Wally.

The program developed was able to find characters other than Wally, with only slight changes to the patterns. Finding Odlaw, who is essentially Wally with a different colour palette, was done through the same functionality that provided the Red and White pattern.

Future development in this area should consider the definition of the certainty of a result with care. Most of the patterns developed suffer from too many false-positives. Reducing these through a more discerning definition of certainty could help to increase the reliability of the program.

The program has potential as an example learning tool for parallel programming. Many of the parallel techniques used within can be understood by any developer who understands computer vision. The inclusion of OpenCV enables most developers to learn some computer vision through their documentation and tutorials.

If the results could be broken down visually, this program could also serve as an outreach device for HPC. Where's Wally? has a place in the popular culture, helping to give light hearted relevance to the discussions.

7 Evaluation

This project suffered from having a scope that was too large. Object recognition, even as a subset of computer vision, is too complex a topic to produce high quality code that encompasses all subtopics. A lack of experience working in projects meant that the disparity between workload and time was not recognised. This highlights the importance of following a work plan and overestimating completion time.

The project was designed as a test-driven development. After the implementation of test driven I/O, the project degenerated into a code-like-hell project.

This project was begun with little knowledge of computer vision techniques. Through the repeated use and analysis of results, a broader appreciation of the discipline has been gained.

Glossary

A list of terms used within

Object A known image, being searched for inside a Scene, feature analysis term

Scene An image that may or may not contain an Object, feature analysis term

OpenCV A computer vision library

libCVD A computer vision library

SIFT Scale-Invariant feature transform; A feature recognition algorithm

SURF Speeded Up Robust Features; A feature recognition algorithm based on SIFT

Extra Reading

The OpenCV documentation is extremely helpful. Of particular use, is the OpenCV cheatsheet[36].

References

- [1] D. Hofer and W. Perrig, "Subconscious image recognition[.]," *Zeitschrift für experimentelle und angewandte Psychologie*, vol. 37, no. 4, p. 580, 1990.
- [2] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2, pp. 1150–1157, Ieee, 1999.
- [3] UK Missing Persons Bureau, "Missing persons data and analysis 2010-2011." missingpersons.police.uk/en/resources/missing-persons-data-and-analysis-2010-2011, August 2013.
- [4] The Children's Society, "Still running 3 full report," 2011. makerunawayssafe.org.uk/sites/default/files/tcs/u24/Still-Running-3_Full-Report_FINAL.pdf.
- [5] Home Office, "National dna database annual report," 2012. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/200407/NDNAD_Annual_Report_2011-12.pdf.
- [6] J. L. Nielsen, "Scientific sampling effects: Electrofishing california's endangered fish populations," *Fisheries*, vol. 23, no. 12, pp. 6–12, 1998.
- [7] Y. Sun and R. Fisher, "Object-based visual attention for computer vision," *Artificial Intelligence*, vol. 146, no. 1, pp. 77–123, 2003.
- [8] J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber, "An integrated, modular framework for computer vision and cognitive robotics research (icvision)," in *Biologically Inspired Cognitive Architectures 2012*, pp. 205–210, Springer, 2013.

- [9] Microsoft, "Kinect for xbox 360," august 2013. <http://www.xbox.com/en-GB/KINECT>.
- [10] V. M. Patel, R. Maleh, A. C. Gilbert, and R. Chellappa, "Gradient-based image recovery methods from incomplete fourier measurements," *Image Processing, IEEE Transactions on*, vol. 21, no. 1, pp. 94–105, 2012.
- [11] L. G. Roberts, "Machine perception of three-dimensional solids," tech. rep., DTIC Document, 1963.
- [12] Google, "Google glass," august 2013. <http://www.google.com/glass/start/>.
- [13] K. Tanaka, "Mechanisms of visual object recognition: monkey and human studies," *Current opinion in neurobiology*, vol. 7, no. 4, pp. 523–529, 1997.
- [14] D. I. Perrett and M. W. Oram, "Visual recognition based on temporal cortex cells: Viewer-centred processing of pattern configuration," *Zeitschrift fur Naturforschung C-Journal of Biosciences*, vol. 53, no. 7, pp. 518–541, 1998.
- [15] OpenCV, "Opencv home website," august 2013. opencv.org.
- [16] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'reilly, 2008.
- [17] Edward Rosten, "libcvd homepage," august 2013. <http://www.edwardrosten.com/cvd/>.
- [18] V. Fragoso, S. Gauglitz, S. Zamora, J. Kleban, and M. Turk, "Translator: A mobile augmented reality translator," in *Applications of Computer Vision (WACV), 2011 IEEE Workshop on*, pp. 497–502, IEEE, 2011.
- [19] A. C. Downton, R. W. Tregidgo, and A. Cuhadar, "Top down structured parallelisation of embedded image processing applications," *IEE Proceedings-Vision, Image and Signal Processing*, vol. 141, no. 6, pp. 431–437, 1994.
- [20] J. Fung and S. Mann, "Openvidia: parallel gpu computer vision," in *Proceedings of the 13th annual ACM international conference on Multimedia*, pp. 849–852, ACM, 2005.
- [21] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "Sift implementation and optimization for multi-core systems," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, IEEE, 2008.
- [22] H. Feng, E. Li, Y. Chen, and Y. Zhang, "Parallelization and characterization of sift on multi-core systems," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 14–23, IEEE, 2008.
- [23] N. Zhang, "Computing optimised parallel speeded-up robust features (p-surf) on multi-core processors.," *International Journal of Parallel Programming*, vol. 38, no. 2, pp. 138 – 158, 2010.
- [24] D. Brownrigg, "The weighted median filter," *Communications of the ACM*, vol. 27, no. 8, pp. 807–818, 1984.

- [25] H. Malepati, *Digital media processing [electronic resource] : DSP algorithms using C / Hazarathaiah Malepati*. Burlington, Mass. : Newnes/Elsevier, [2010], 2010.
- [26] S. BURTSEV and Y. KUZMIN, "An efficient flood-filling algorithm," *COMPUTERS AND GRAPHICS*, vol. 17, no. 5, pp. 549 – 561, n.d.
- [27] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.
- [28] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, 1985.
- [29] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244–256, 1972.
- [30] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.
- [31] J. E. Hershberger and J. Snoeyink, *Speeding up the Douglas-Peucker line-simplification algorithm*. University of British Columbia, Department of Computer Science, 1992.
- [32] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision–ECCV 2006*, pp. 404–417, Springer, 2006.
- [33] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 10, pp. 1615–1630, 2005.
- [34] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 886–893, IEEE, 2005.
- [35] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration.," in *VISAPP (1)*, pp. 331–340, 2009.
- [36] OpenCV, "Opencv 2.4 cheat sheet (c++)." docs.opencv.org/trunk/opencv_cheatsheet.pdf, august 2013.