Finding Wally

Parallel Object Recognition

Cian Booth

MSc. Dissertation Report

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Computer vision is a field concerned with using computers to process and analyse images. It is an attempt to augment the human ability to percieve visual information. In doing so, it allows users to increase their own capabilities to accurately and quickly extract information from an image.

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 basic information received from the eyes, deducing any objects the information describes. For the average human, seeing the local environment and immediately recognising objects within is expected. The act of object recognition is a subconscious one, and the complexity associated with it is not known. The level of information contained in a single moment of vision is extremely high. A normal room contains hundreds or thousands of distinct shapes, all of which must be pieced together to form a cohesive object. Light levels typically vary from one position in a room to another, so an object may appear different on two sides. The human brain is still able to analyse this data extremely rapidly, and make decisions about objects in the image. This task is very intricate task and not simple to replicate.

Consider describing any single object. The adage "a picture is worth a thousand words" has a particular meaning here; fully describing the object would be an extensive task to say the least. Expressing this in a logically defined way increases the level of complexity; exactly how shiny is an apple, what precise shade of brown is the shoe, what is the exact shape of the knife? These are all questions, the answer to which is difficult to begin to define.

Unsurprisingly, implementing this on a computer is not simple. The normal description of a computer is a machine that exceeds at calculating mathematical operations at speed. Modern computers are binary devices, designed to store and act upon precise values. Human vision, as we understand it, does not directly map onto the operations that a computer excels at.

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

1.2 High Performance Computer Vision

One of the major blockades for widespread use of computer vision is the limiting relationships between speed, accuracy and generality. The human eye can detect most objects both quickly and accurately. Computer vision is not so advanced. Achieving real time speeds often comes at the cost of accuracy or generality. On the other hand, creating a fast or generalised system, if possible, comes at the cost of a greatly increased completion time.

For each way to describe an object, several different 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 do not create similarly dependable results. By combining these techniques together, the reliability and robustness of computational object recognition can be improved. Using all available techniques will considerably increase the time the vision program takes to complete. This can be mitigated by computing each techniques in parallel.

An intuitive method of parallelism for this type of problem is the task farm. This means that general tasks (in this case, each method of identifying an object) will be run concurrently. Task farming is useful in this case, because it allows the use of serial libraries and algorithms that would otherwise complicate or prevent parallelism. Computing results simultaneously, a task farm allows computer vision programs to produce results quickly, reliably and generally.

1.3 Potential Use Cases

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

The UK Missing Person Bureau released data on missing persons in 2010-2011 [1]. During this period, two thirds of missing people in the UK were under the age of 18. Such a group is particularly vulnerable to abduction and abuse if left unsupervised. Although the majority of missing people were found within the 5 miles of their homes, up 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, and it becomes untenable to search efficiently. Almost 4/5 of missing people are found within the first 16 hours. According to information released from a survey of young runaways [2], 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. Computer vision is already used to facilitate searches, but the volume of data available can exceed the capabilities of existing programs. Furthermore, surveillance data from CCTV equipment produces data at real time speeds, so faster than real time processing speeds are desired. Implementing a parallelised version of image recognition tools should greatly reduce the time an image needs to search. Governmental agencies, such as the police, could have access to very large computer systems, allowing for a high degree of parallelism.

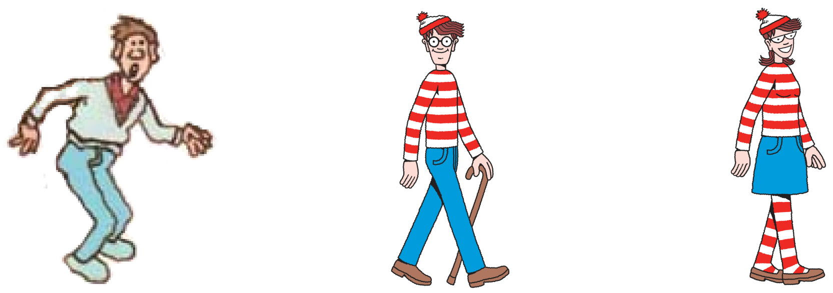
Another potential use can be found with surveying populations of wild animals. Wildlife conservation is a delicate task, which could benefit from rapid computer vision techniques. To accurately know which species are endangered, it is important to have an accurate count of the members of the species for a given region. Actively surveying the population by means of physically interacting with members can have adverse effects on the population. Nielsen [3] discusses the negative effects of electrofishing on rare fish populations. She goes on to indicate 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 be to immediately identify each species correctly. This is rarely the case; a single human is fallible and a team may be 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 could even allow non-experts to survey the populations, freeing up researchers for more critical tasks.

An everyday use of parallel object recognition is with 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. Parallelism would be of use here, as the sheer quantity of data for a large scale system like this would prevent real time analysis.

A less daunting, more approachable usage case can be found in Where's Wally? puzzles.

1.4 Where's Wally? as a Test Case

Where's Wally? puzzles are a good testbed for parallel computer vision. Each puzzle is a simple cartoon, normally a large image filled with various characters, who wear simply coloured clothing, see Figure 1(a). One of these characters is the eponymous Wally, who is dressed distinctly from most others characters, see Figure 1(b). Similarly dressed characters exist, Figure 1(c), adding some complexity to finding Wally correctly. The cartoon nature of the characters means that shapes are boldly coloured and often bordered by a black line. As Where's Wally? is a puzzle, sometimes Wally will be hard to find; he is often obscured, camouflaged or simply small. Creating a program to solve Where's Wally? is non-trivial, requiring a combination of computer vision techniques. Despite this, the puzzle provides a simple base to implement parallel object recognition.



(a) A normal person **(b)** Wally **(c)** Wenda

Figure 1: Characters from Where's Wally?

1.5 Goals

In this report, Where's Wally? puzzles will be used a testbed to determine if High Performance Computer Vision is advanced enough for everyday use. This will include 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. The generality of the suite of functions will be tested on non-Wally puzzles, to determine how generalised the system is.

1.6 Overview of Report

This report will begin by discussing the underlying information required to easily comprehend parallel object recognition. Included within will be literature reviews as appropriate. The next section will contain a discussion of the patterns used to recognise Wally. Each will include an analysis of the algorithm selection and a discussion of the level of parallelism that can be exposed. This will then be combined to produce an implementation of a Where's Wally? solver. Within will be an examination of the parallelism that can be 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 Parallel Programming

When implementing parallelism, it is desirable to not only speed up a program, but also to use an appropriate number of cores. A useful definition is one of speedup, the amount faster that a parallelised version of a program is to an efficient serial implementation.

$$\text{Speedup}(P) := \frac{T_{\text{serial}}}{T_{\text{parallel}}(P)} \tag{1}$$

Equation 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 P; 2 computing units working on a problem will solve it twice as fast as a single unit.

Most implementations do not achieve this, especially for larger numbers of computing units. It is worth noting that some programs actually produce super-linear speedup [4]. This might happen because multiple processors increase the available cache size beyond the size of the problem This means that minimal calls to main memory are required, which are often the major cause of slow down.

A fixed size program with a speedup that scales directly with the number of units used, is said to have 'strong scaling'. An example of this is having a program that calculates the sum of each element in a 1024x1024x1024 matrix. The number of elements being added is very large, so there is plenty of scope for a massively parallel system.

Of course, the scaling is not strong for an infinite number of processors. For more than 1024x1024x512 processors (a minimum of two elements per processor is needed for addition), there is not enough work to be done per processor. This inherently has a negative effect on the speedup. The scaling will most likely stop being strong before this, due to Amdahl's law, seen in equation 3.

If a program is composed of serial and parallelisable portions, and α is the portion of the program that is purely serial, then the time a parallel program takes is:

$$T_{parallel}(P) = T_{serial}\left(\alpha - \frac{1-\alpha}{P}\right) \tag{2}$$

Inserting this into the previous definition of speedup in equation 1, we see Amdahl's Law emerge.

$$\text{Speedup}(P) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(P)} = \frac{T_{\text{serial}}}{T_{\text{serial}}\left(\alpha - frac1 - \alpha P\right)} = \frac{1}{\alpha + \frac{1-\alpha}{P}} \tag{3}$$

It is clear that as P tends to large numbers, the speedup approaches the constant value of $1/\alpha$.

In spite of this, parallel programs regularly achieve 'weak scaling'. This is the case if a program scales with the number of cores for a fixed amount of work per computing unit. Gustafson's law describes the mathematics of weak scaling. Here, P is the number of computing units, and N is the size of the problem.

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

We define $T(P,N)$ as the time for the purely serial components to complete, T_{serial}, with the time it takes for P processors 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. $NT_{\text{parallel},N}$.

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

$$\text{Speedup}(P,\beta P) = \frac{\alpha + \beta P(1-\alpha)}{\alpha + \beta(1-\alpha)} \propto P \tag{4}$$

It is clear that Gustafson's Law produces better scaling, as long as N scales with P. A trivial example of weak scaling is calculating π to 10 digits on each processor, and then adding that to some shared variable. The problem size is fixed per processor, but the overall work increases with the number of processors.

2.2 Shared Memory Parallelism

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

Shared memory multiprocessing is a form of parallelism that relies on the presence of shared memory. This is a region of memory that can be accessed equally by several processors. Normal examples of this are in multicore systems, which have a shared L2 or L3 cache. Having shared memory allows data to be easily shared amongst processors. This removes redundant 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.

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 parallelised "Hello World" loop'

```
// a simple "hello world" loop
for(int i=0; i<4; i++) {
  printf("hello␣world␣from␣\%d", i);
}
```

```
// and again parallelised
#pragma omp parallel for
for(int i=0; i<4; i++) {
  printf("hello␣world␣from␣\%d", 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. Futhermore, 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, unsurprisngly, the practice of 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. Instead, messages are often sent over the bus, meaning that the processors being used can be in different nodes.

One of the most commonly implemented stanard is the Message Passing Interface (MPI)

Shared memory parallelism prone to race conditions.

Shared

OpenMP.

2.3 Computer Vision Libraries

2.4 Parallel Computer Vision

2.5 Parallel Image Recognition

2.6 Shape and Colour Analysis

One obvious way of analysing an image is to break it down into shapes and colours. In linguistic terms, it is easiest 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 and understand, and thus is more intuitive to program. Most images are saved as raster images (e.g. PNG, BMP, GIF), which is a 2D array of colours that directly map to set of pixels on the screen. This is opposed to vector images that store the location and colour of geometric primitives (squares, circles, triangles, etc.). Vector images, rather than raster images, are directly easier to perform shape and colour analysis on. However, input devices such as webcams and scanners typically produce images in raster formats.

Regardless of format, methods of colour analysis are simple to implement programmatically. This is because colour is intrinsic to all methods of storing the data of an image. Shape analysis is less simple for raster images. Shape boundaries must be found, which is normally done through edge detection. These boundaries are then analysed to find points of intersection, and the curvature between them. The group of curves must then be examined to find what shapes exist.

These methods allow for a descriptional, heuristic method for locating characters. No previous image of Wally is required, only a description, such as "red and white stripes" or "black glasses". This allows users to extend the solution past Wally, and towards other characters, who do not have to be seen prior. This form of analysis can lead to false positives, and should be combined with other results.

An example of the usefulness of this, beyond Where's Wally, could be found in augmented reality technology, such as Google Glass. A common problem people experience is losing their keys. Users with access to AR devices could use colour and shape analysis to enhance their searching (i.e. if they are visible, but in a cluttered area). As keys generally have a few well defined shapes and colour schemes, the device would not have to store what the keys look like in advance. Assuming that parallelism is available, this could potentially be done faster than the human eye can search, helping the user significantly.

2.7 Feature Analysis

Some of the most reliable computer vision libraries (such as SIFT [5]), were developed while considering the neuroscience of human vision. Tanaka[6] and Perrett and Oram[7] found that human object recognition identifies objects with features that are invariant to brightness, scale and position. These results have been used as inspiration for feature analysis. This technique finds features; regions of an image which are scale, rotation and illumination invariant. These features are most immediately useful when compared with the features of another image. For example, the features from an image of just Wally can be used to locate Wally in a normal puzzle image.

This method generally requires an existing image to find Wally, which restricts the flexibility of the search. However, this method is very reliable, as long as Wally is not obscured, it will likely locate him correctly. Normally, Wally is obscured, so this method should be combined with other techniques.

Feature analysis benefits from task farming when there is an image needs to be searched for a large number of sub-images. A beyond Wally example would be in detecting employees going into work on a flexitime basis. This would use photos of each employee combined with CCTV to note the time that workers enter and leave their workplace. Users would not need any form of ID other than their own faces, and this can be combined with existing security systems.

3 Literature Review

3.1 Shape and Colour Analysis

nothing here yet

3.2 Feature Recognition

Detecting features within an image is an important technique that can be used in object recognition. One of the most robust algorithms available for this is SIFT (Scale-Invariant Feature Transform), developed by David Lowe [5]. This algorithm uses various techniques to find scale, rotationally and translationally invariant keys, which are partially invariant under affine transforms and changes in illumination. The keys contain feature vectors, which describe the area around them. This algorithm is very reliable at finding a given sub-image within a larger image. However, some of the techniques required to produce reliable results require unexpected amounts of memory. To create a scale-space version of the image, 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.

However, if the algorithm is used for analysing multiple images concurrently, memory constraints would limit it's efficiency. This can be seen with Zhang's parallel implementation of SIFT [8], where the scale-space creation is one of the areas the program spends most time on. This paper shows reasonable parallel efficiency, but only has results up to 32 cores. 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.

SURF (Speeded-Up Robust Features), developed by Bay et. al.[9], 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; calculating versions of the image for the scale space are independent of the previous level and can be done in parallel.

4 Colour Analysis

Colour analysis is arguably the easiest of the three main techniques to implement. It only requires that an image be accessible as an array of pixels, which is the common storage format for images. This project only considers Where's Wally images, with pixels that are accessed in the RGB format. This means that each pixel contains information on how much Red, Green and Blue (RGB) to show. This is in contrast to HSV, or hue, saturation and value. Hue represents the pixel's position on a colour wheel, saturation represents it's vibrancy and value is a measure of it's brightness. RGB maps easily to HSV, but in cartoon images, there isn't generally a large change in an objects lightness level. It is easier to work with RGB images, but the methods described below work for HSV.

What follows is a list of functions and search patterns that utilise colour analysis.

4.1 Function: `get_colour_in_image`

usage `get_colour_in_image(image, colourA, colourB, rg,gr,rb,br,bg,gb)`

image The input image, as loaded by OpenCV.

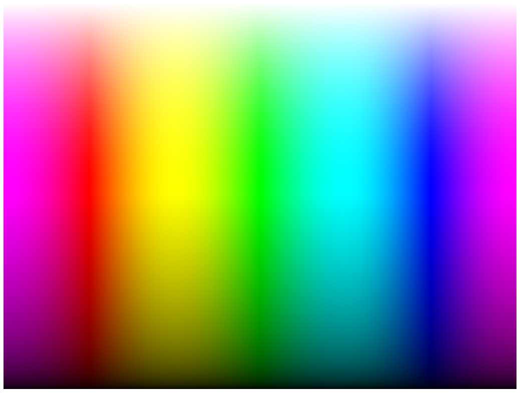
colourA,colourB Strings describing the limits of colours to allow, in the form "#RRGGBB"

rg,gr,... The allowed ratio of each colour. For example, $R >= rg \times G$, and $G >= gr \times R$.

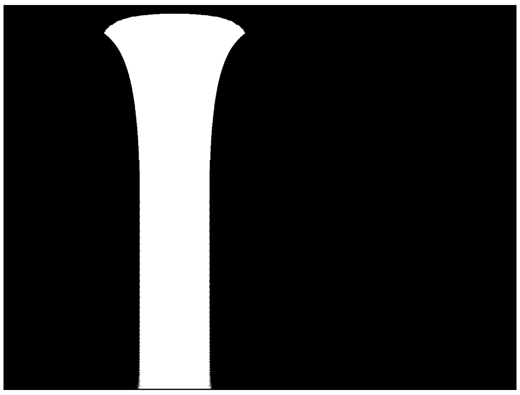
returns Binary mask the same size as the input image, with values of 255 if pixel was within the required values, and 0 otherwise.

This function allows a user to search the image for a specific range of colours. Moreover, the user is able to restrict the results based on the relationships between the pixels colour values. For example, a user can search for colours that are yellow by searching the entire colour space and ensuring that R and G have similar values, and that they are both greater than B;

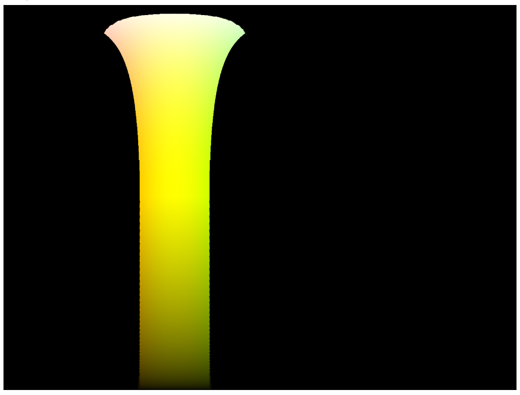
```
yellow = get_colour_in_image(input,"#000000","#FFFFFF",0.8,0.8,1.1,0,1.1,0)
```



(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 overlayed on the original image, revealing "yellow" colours.
Figure 2: Isolating yellow from an image using `get_colour_in_image`

The output of that function can be seen in Figure 2. The description "yellow" is not clearly defined; there is no single colour code for yellow. However, it is fair to describe yellow as some combination of red and green, with little to no blue. Without describing the specific colour code being searched for, yellow colours can be found, and results can be produced. This function allows the user to search for objects in a more robust fashion, if just for some colour.

4.2 Function: `get_greyscale_in_image`

usage `get_greyscale_in_image(image, low, high, tolerance)`

image The input image, as loaded by OpenCV.

low,high The range of values that should be allowed on the greyscale.

tolerance This is a finite tolerance, with non-zero values allowing nearly greyscale colours to be included.

returns Binary mask the same size as the input image, with values of 255 if pixel was within the required values, and 0 otherwise.

Although greyscale is a special case of colour, slightly different requirements were discerned for greyscale searches. Users are able to define a tolerance, which allows for colours that are not strictly grey to be included in results. For example;

```
black = get_greyscale_in_image(input, 0, 40, 20)
```

will include blacks and very dark greys, as well as very dark reds, greens and blues.

4.3 Function: `find_regions_from_mask`

usage `find_regions_from_mask(mask)`

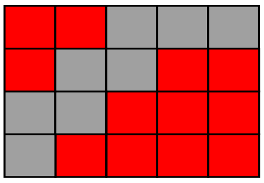
image The input mask, a binary single channel matrix.

returns A list of structs, containing information about the size of the region, the average position and bounding box.

It is important to be able to discretise the results of the above functions. Something is needed that will convert the binary matrix into a list of results. This can be done by counting and collecting data about the distinct regions available in the matrix. A region here is defined as a group of pixels that are connected through non-zero nearest neighbours.

A computationally naive way to do this is shown in Figure 3. Each non-zero pixel is assigned a unique integer value. Each pixel of the matrix is looped over, and it's value becomes the maximum of itself and it's four nearest neighbours. This is repeated until no pixels change 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 have a unique value. By counting the unique values in the matrix, the number of regions can be found. Similarly, properties of the region can be calculated by analysing specific unique values. For example, the mean position of the pixels within the region, the size of the region, and the bounding box around the region.

More computationally efficient methods exist. Lifeng [10] describes the method of connected-component labelling. The removes the need for iteration until the matrix stabilises, and reduces the memory need per pixel. 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 noted. Once the matrix has been traversed, equivalent labels are merged, and the regions have been d. This method, is both simple and memory efficient enough to be suitable for parallelism; temporary labels can be calculated concurrently and the final labels only need a small amount of halo data to be obtained.



(a) Input A

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

(b) B, such that $B_{ij} = iY + j$

1	2	0	0	0
6	0	0	9	10
0	0	13	14	15
0	17	18	19	20

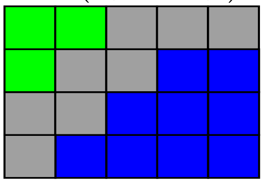
(c) $C^0_{ij} = A_{ij} \cdot B_{ij}$

6	2	0	0	0
6	0	0	14	15
0	0	18	19	20
0	18	18	19	20

(d) $C^{t+1}_{ij} = max\left(C^t_{ij}, nbr(C^t_{ij})\right)$ for $C^t_{ij} \neq 0$

6	6	0	0	0
6	0	0	20	20
0	0	20	20	20
0	20	20	20	20

(e) The final result, when $C^{t+1} = C^t$



(f) A colour representation of the regions

Figure 3: A simplistic region detection algorithm in action. $nbr(x_{ij}$ is a list of the nearest neighbours of x_{ij}

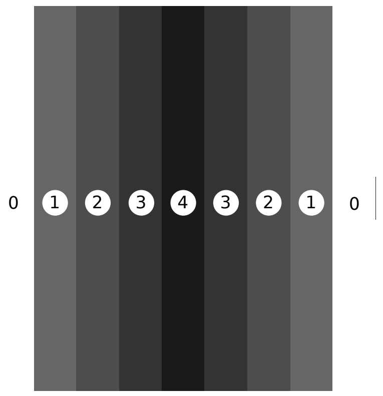
5 Line Width Estimation

In cartoon images, characters and objects are regularly bounded by black lines. Within some 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 assumed that the first image is at twice the scale of the second. This is useful for putting an upper limit on how large a match between an object and a scene can be. Estimating the line width can be done through the combination of a few techniques.

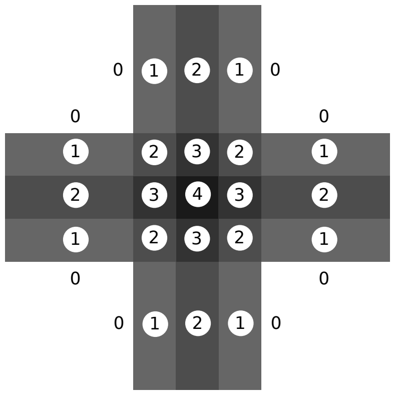
For simple images, with lines that do not intersect, a few steps need only be followed. The first is to produce a mask of all the black in the image. The next step is to count the distance to the nearest zero-valued pixel (called zero-distance hereafter). This can be done with the OpenCV `distanceTransform` function. The final step is to find the maximum zero-distance in the image. In the example image, figure 4(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. Thus, finding the maximum zero-distance could resolve line width of simple images to within 1 pixel.

More complicated images, with intersecting lines, this method fails to produce correct results. Figure 4(b) shows a situation where the method is incorrect. 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 below, relies on two statistical properties of the image; the average distance of the pixels, and standard deviation of each pixel from the average distance.



(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 4: Two examples of lines with their zero-distances calculated.

5.1 Average Distance to a Zero Valued Pixel

By simple inspection of figure 4(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 equation 5, where PixelDistance(n) lists the zero-distances in an n pixel wide line.

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

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 geometric can be conveniently expressed as simple formula, described in equation 6.

$$\text{Sum}(N) := \sum_{i=0}^{N} i = \frac{N(N+1)}{2} \tag{6}$$

First, we consider even valued widths, as this is the simplest case. This is can be calculated as twice the sum of integers in the range $[0, N/2]$, as seen in equation 7.

$$\begin{aligned}\text{EvenSum}(N) :&= 2\ \text{Sum}(N/2)\\ &= 2\sum_{i=0}^{N/2} i\\ &= 2\frac{(N/2)(N/2+1)}{2}\\ &= N\left(\frac{N}{4}+\frac{1}{2}\right)\\ &= \frac{N^2+2N}{4} \end{aligned} \tag{7}$$

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 8.

$$\begin{aligned}\text{OddSum}(N) :&= \frac{N+1}{2} + 2\ \text{Sum}((N-1)/2)\\ &= \frac{N+1}{2} + 2\sum_{i=0}^{(N-1)/2} i\\ &= \frac{N+1}{2} + 2\frac{((N-1)/2)((N-1)/2+1)}{2}\\ &= \frac{N+1}{2} + \frac{N-1}{2}\frac{N+1}{2}\\ &= \frac{N+1}{2}\frac{N+1}{2}\\ &= \frac{N^2+2N+1}{4} \end{aligned} \tag{8}$$

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 9 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.

$$\begin{aligned}\text{AvgDist}(N) :&= \begin{cases} \frac{\text{EvenSum}(N)}{N} = \frac{1}{4}\left(N+2\right) & \text{if } N \text{ even}\\ \frac{\text{OddSum}(N)}{N} = \frac{1}{4}\left(N+2+\frac{1}{4N}\right) & \text{if } N \text{ odd} \end{cases}\\ &= \frac{N+2}{4} + \mathrm{O}(N^{-1}) \end{aligned} \tag{9}$$

For large values of N, the average distance approaches the EvenSum formula. This can be seen in figure 5. The background shading represents the values bounded by the values of OddSum and EvenSum. The figure also shows that the maximum error for any given value of N is at most a quarter of a pixel.

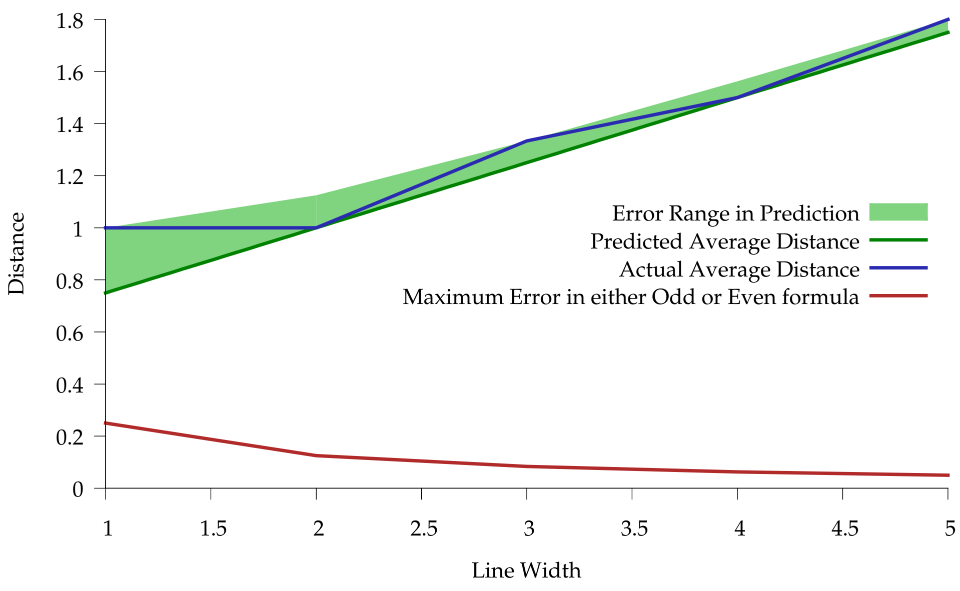


Figure 5: Graph showing the estimation of the average distance to a zero valued pixel using the Odd and Even formulae.

Equation 9 shows that there is an approximately 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 10, to calculate line width from the average zero-distance.

$$\text{LineWidth}(avg) := \text{AvgDist}^{-1}(avg) = \frac{avg - 0.5}{0.25} \tag{10}$$

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 9 was a quarter of a pixel. Equation 10 accordingly has an 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 a upper bound on the scaling between two images.

5.2 Standard Deviation in Distance to Zero Valued Pixels

The line width can also 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.

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

To simplify the mathematics, we act on summation for now. We will call this operand Deviance.

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

As before, x_i will take values from $[0, .., N, ...0]$, and $\text{AvgDist}(N)$ is defined in equation 9. We will show the method for finding the standard deviation for even valued widths, odd values follow from above. We can now write this equation (ignoring the nominal differences between odd and even values of N) as

$$\begin{aligned}
\text{DevEven}(N) &= 2\sum_{i=0}^{N/2}\left(i - \frac{N+2}{4}\right) \\
&= 2\sum_{i=0}^{N/2}\left(i^2 - 2i\frac{N+2}{4} + \frac{(N+2)(N+2)}{16}\right) \\
&= 2\left(\sum_{i=0}^{N/2}(i^2) - 2\frac{N+2}{4}\sum_{i=0}^{N/2}(i) + \sum_{i=0}^{N/2}\left(\frac{(N+2)(N+2)}{16}\right)\right) \\
&= 2\left(\sum_{i=0}^{N/2}(i^2) - \frac{N+2}{4}\frac{N(N/2+1)}{2} + \frac{N}{2}\frac{(N+2)(N+2)}{16}\right) \qquad (11)
\end{aligned}$$

The sum in equation 11 is, as with equation 6, easily expanded using geometric identities. In this case, the formula is described in equation 12.

$$\sum_{i=0}^{N} i^2 = \frac{1}{6}N(N+1)(2N+1) \qquad (12)$$

We can now continue expanding DevEven(N);

$$\begin{aligned}
\text{DevEven}(N) &= \frac{1}{3}\frac{N}{2}(N/2+1)(N+1) - \frac{N(N+2)(N+2)}{4} + \frac{N(N+2)(N+2)}{16} \\
&= \ldots \\
\ldots &= \frac{1}{48}\left(N^3 - 4N\right) \qquad (13)
\end{aligned}$$

Using this formula for DevEven, we can now calculate the standard deviance.

$$\begin{aligned}
\text{StdDevEven} &= \sqrt{\frac{\text{DevEven}(N)}{N}} \\
&= \sqrt{\frac{N^3 - 4N}{48N}} \\
&= \frac{N}{4\sqrt{3}} + O(\sqrt{N}) \qquad (14)
\end{aligned}$$

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

$$\begin{aligned}
\text{StdDevOdd} &= \sqrt{\frac{\text{DevOdd(N)}}{N}} \\
&= \sqrt{\frac{N^2}{16N} + \frac{N^2 - 3N + 2}{48N}} \\
&= \frac{N}{4\sqrt{3}} + O(\sqrt{N}) \qquad (15)
\end{aligned}$$

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

$$\text{StdDev}(N) := \frac{N}{4\sqrt{3}} \qquad (16)$$

Figure 6 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.

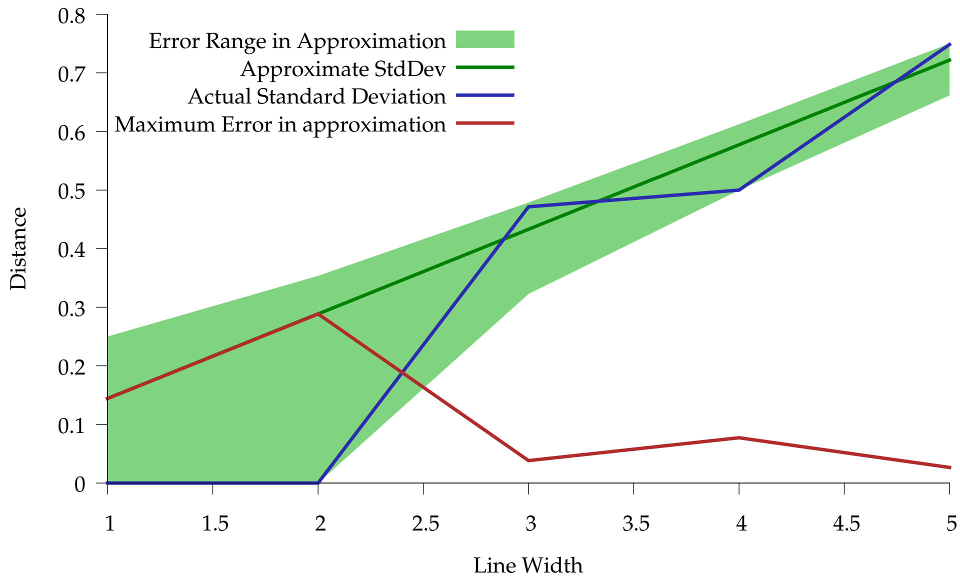


Figure 6: Graph showing the relationship between line width and the Standard Deviation of pixel distance to zero valued pixels.

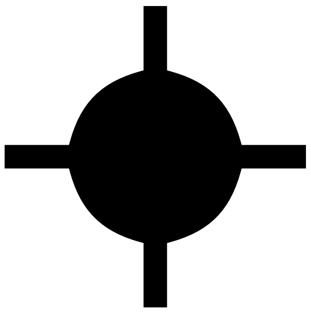
Again, as a linear formula has been produced, the relationship can be reversed, to get line width from standard deviation.

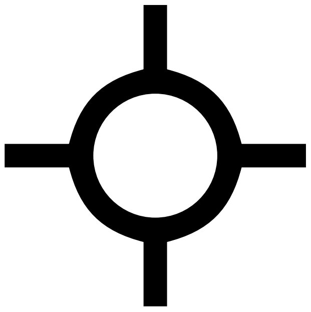
$$\text{LineWidth}(stddev) := \text{StdDev}^{-1}(stddev) = 4\sqrt{3} \cdot stddev \tag{17}$$

Equation 17 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.

5.3 Combining the Methods

Each of these methods is flawed when introduced into an image with overlapping lines, as in figure 4(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, in this case, is a mask 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 7. 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 sane mask, replacing solid circle in the center with it's border

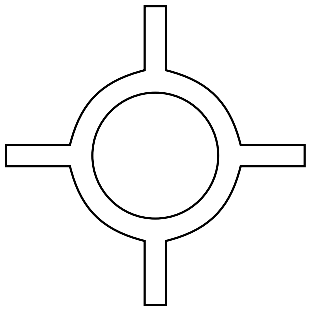
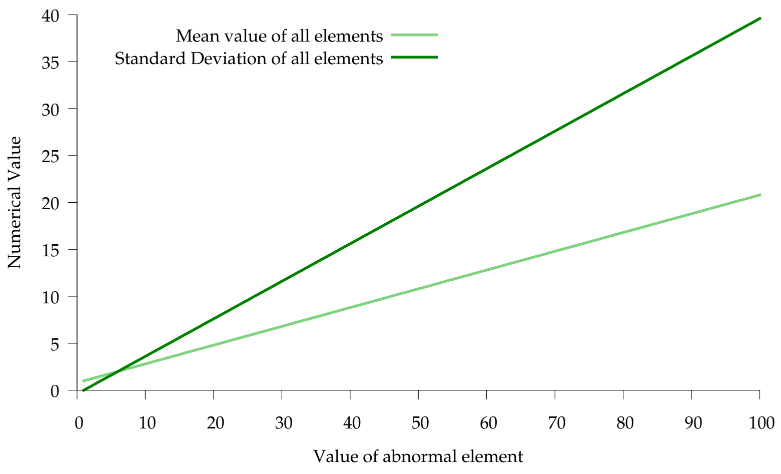
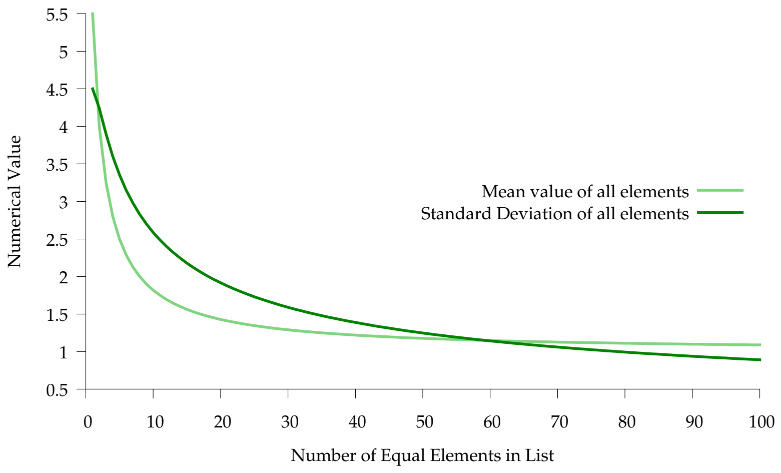

(c) The image after too many maxima have been removed, which would also distort the equation

Figure 7: 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 8, 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 8: 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.

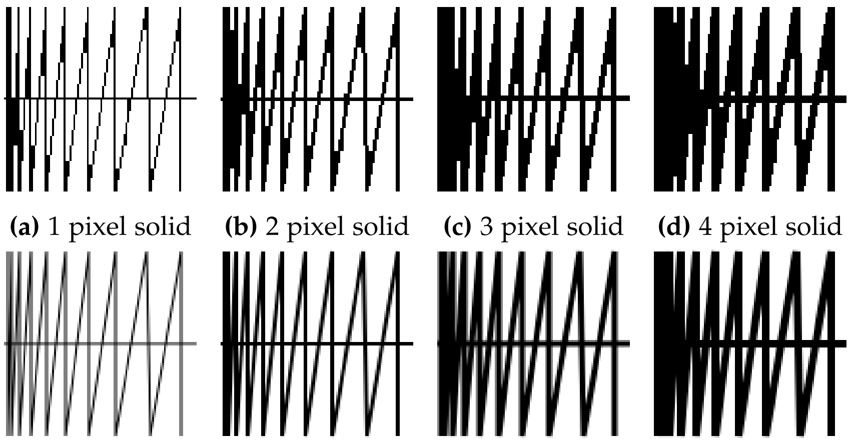
$$Z_1 = \{1\},\ Mean(Z_1) = 1,\ StdDev(Z_1) = 0$$

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

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.

5.4 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 9. 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 10 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.



(a) 1 pixel solid **(b)** 2 pixel solid **(c)** 3 pixel solid **(d)** 4 pixel solid

(e) 1 pixel aliased **(f)** 2 pixel aliased **(g)** 3 pixel aliased **(h)** 4 pixel aliased

Figure 9: 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

Figure 10: A comparison of the line width analysis technique on images from 9 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.

5.5 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.

I will put in some results here when I put them together

6 Red and White

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.

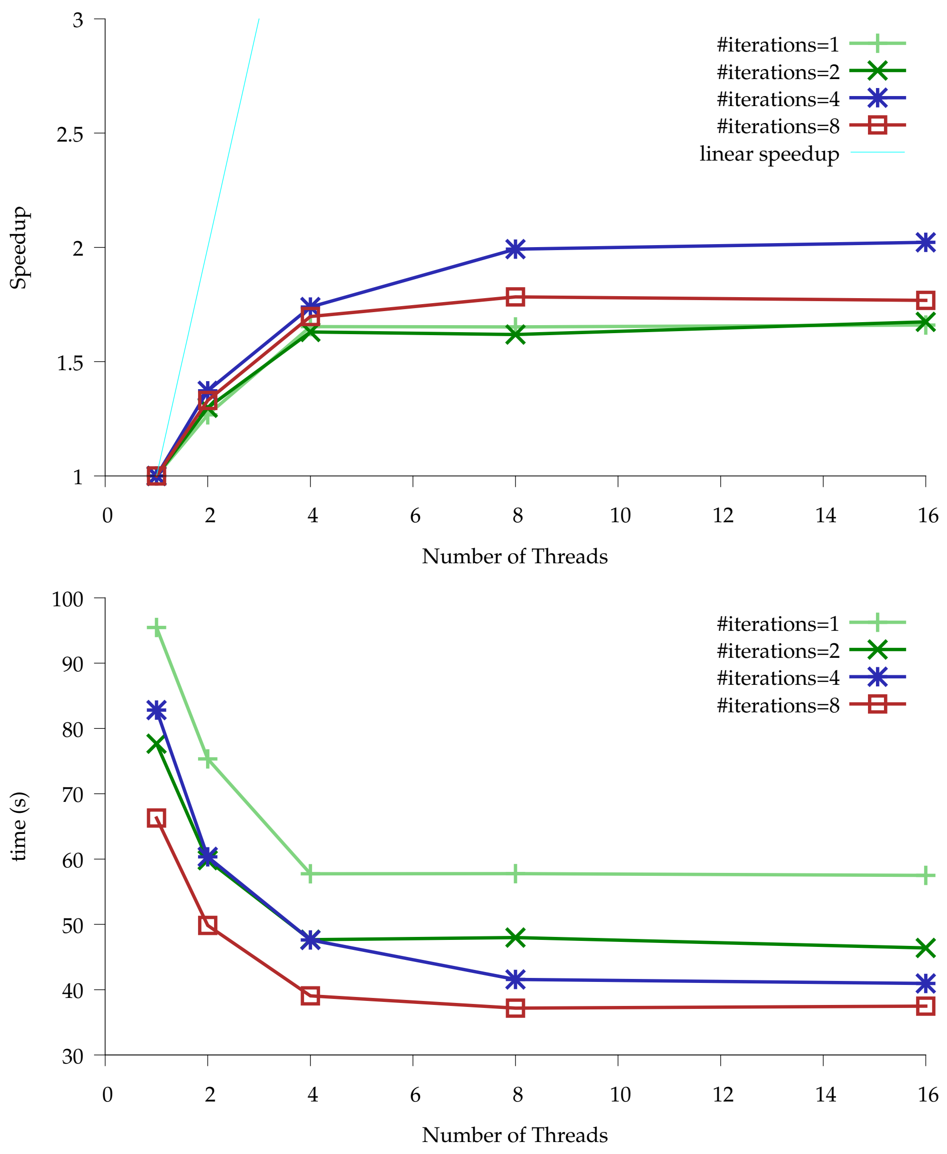
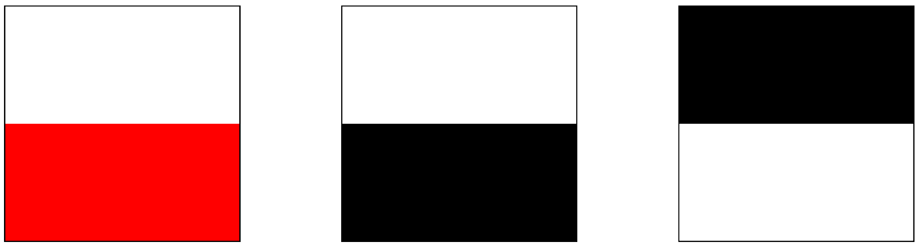


Figure 11: Speedup of line width analysis due to parallelism

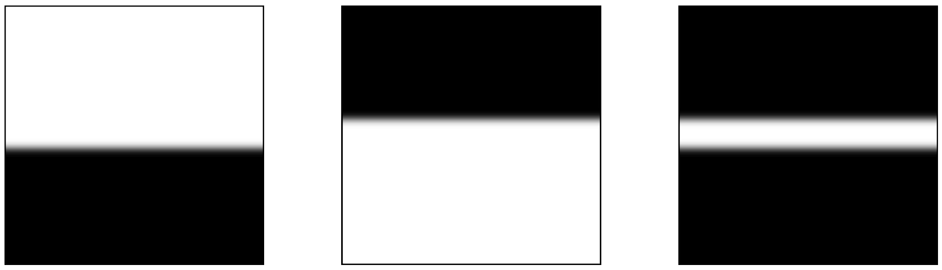
6.1 Implementation

Regions with red and white stripes can be found using a few simple techniques. Here, we find two masks (arrays with binary values) that describe the location of sufficiently red and white regions in the image. The term sufficiently is used, as defining 'red' and 'white' is not a simple task for a computer. The function `get_colour_in_image`, discussed in section 4.1, covers the issues with colour definition.

Once the masks have been created, they are vertically blurred using a Gaussian blur. This is done so that when the two masks are multiplied together, there will be non-zero values found in overlapping areas. Figure 12 shows this graphically.



(a) Red and White Stripes **(b)** White Mask **(c)** Red Mask



(d) White Mask + Blur **(e)** Red Mask + Blur **(f)** Red and White blurs

Figure 12: Example of finding red and white striped regions

The multiplication now describes regions that have red and white regions of colour that are above each other. As seen in figure 12(f), the region found may not encompass the entire area desired. This is solved by blurring the located stripe, making a combined region with nearby stripes. The regions are then identified using the `find_region_from_mask` function.

6.2 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(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 13.

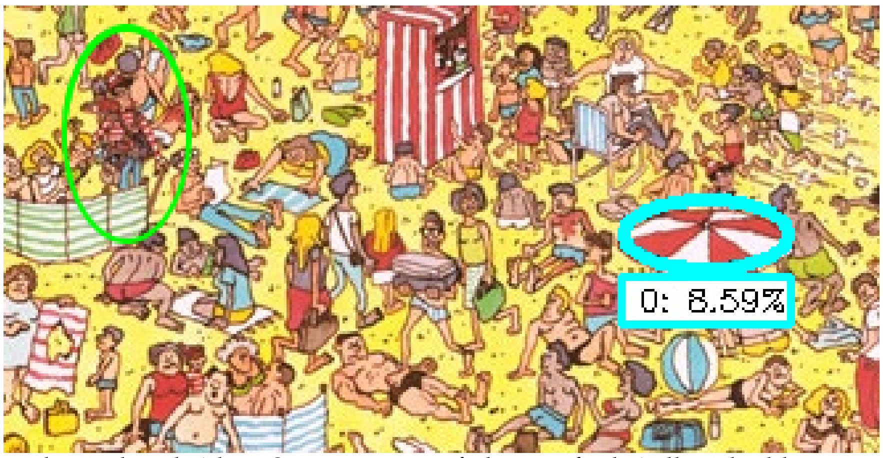
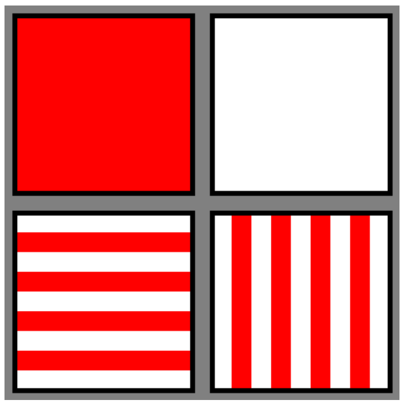


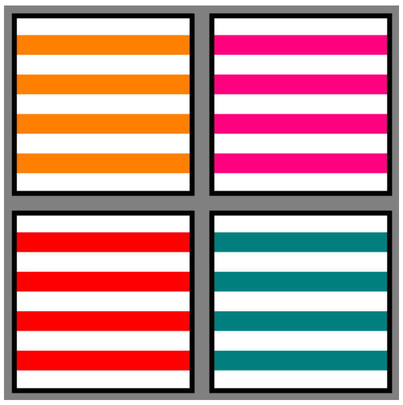
Figure 13: The Red and White Stripes pattern failing to find Wally. The blue ring indicates the top result (an umbrella). The green ring indicates Wally's actual location, which is not even ranked in the top 100 results.

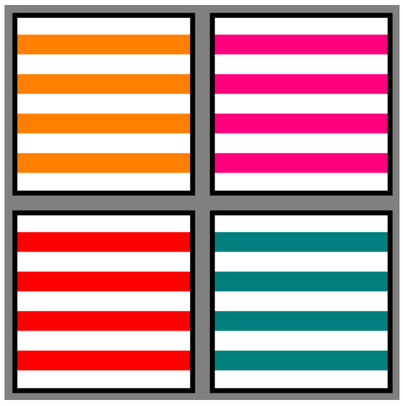
One way to mitigate this is to remove 'horizontal' matches from the vertically blurred mask. Wally is normally found standing upright, and so his jumper normally is striped vertically. Horizontally stripes are unlikely to be from potential Wallys, so they are removed from the search. This involves repeating 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.

6.3 Testing


(a) R&W test 1


(b) R&W test 2


(c) R&W test 3

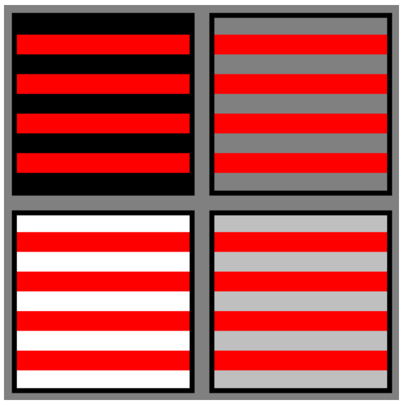

(d) R&W test 2

Figure 14: Test images used to check the red and white pattern

6.4 Parallelism

Finding Wally's stripes is very parallelisable. The image can be split into parallel subimages to find the red and white masks. The vertical blurring only needs a small amount of information from any vertical neighbour, due to the vertical blurring. Choosing to break the image up vertically avoids this entirely. Similarly, the multiplication can be done on subimages. Finding the regions requires more inter-thread interaction. Each thread can calculate the regions for it's own subimage. It must then find out if it's regions are equivalent to regions in another thread. This can be done by sending the top row to the process above it, and the left column to the process left of it. By comparing the two regions a halo element is assigned, the equivalence between threads can be established. Figure 15 displays this graphically.

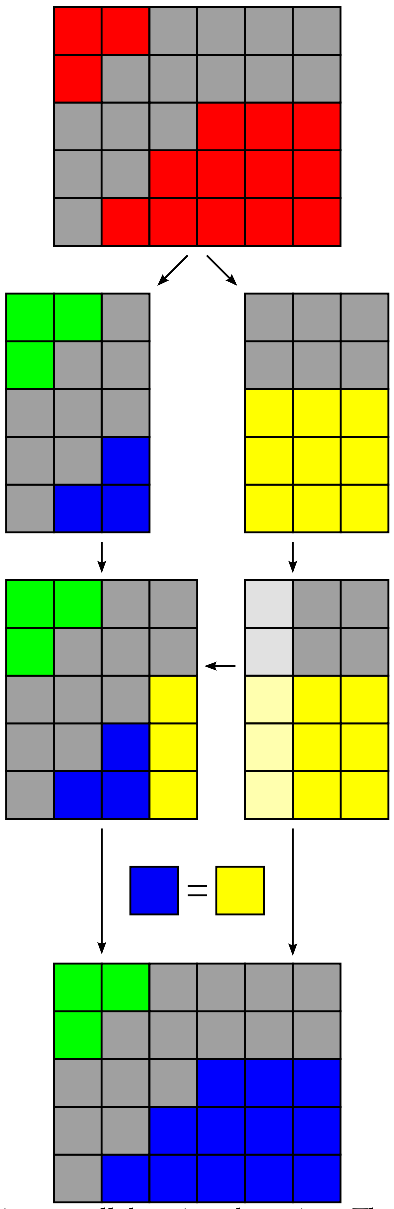


Figure 15: A diagram explaining parallel region detection. The initial image is split into two sections, and each region inside the subimage is given a unique id. One thread sends halo data to the other, which then calculates what threads are the equivalent. The main image is then put back together, with the parallel region equivalences applied.

7 Implementation

8 Results

9 Conclusion and Evaluation

References

[1] 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.

[2] 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`.

[3] J. L. Nielsen, "Scientific sampling effects: Electrofishing california's endangered fish populations," *Fisheries*, vol. 23, no. 12, pp. 6–12, 1998.

[4] J. Gustafson, "Fixed time, tiered memory, and superlinear speedup," in *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pp. 1255–1260, 1990.

[5] 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.

[6] K. Tanaka, "Mechanisms of visual object recognition: monkey and human studies," *Current opinion in neurobiology*, vol. 7, no. 4, pp. 523–529, 1997.

[7] 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.

[8] 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.

[9] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision–ECCV 2006*, pp. 404–417, Springer, 2006.

[10] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.

[11] OpenCV, "Opencv 2.4 cheat sheet (c++)." `docs.opencv.org/trunk/opencv_cheatsheet.pdf`, august 2013.