Chapter 8

AMSS and Hough Transform speed evaluation

Some parts of the algorithms described in the previous chapters were impractically slow. Although the creation of highly optimised and efficient algorithms that deliver maximum speed was not the primary focus of this thesis, part of the second goal outlined in Section 3.3.4 was that fractures should be detected in the shortest time possible. To make the algorithms usable for rapid, semi-automatic detection of long-bone fractures, it was necessary to decrease the computation time, while still utilising the available resources. This chapter examines the ways in which a more rapid detection can be achieved.

The first part of this chapter examines the execution speed of the AMSS algorithm described in Chapter 4, and explains why the smoothing is slow to perform. Some methods by which the calculation time can be decreased are suggested, and the improvements from these methods are quantified. The remainder of the chapter examines the execution speed of the standard and modified Hough Transforms from Section 5.3, and explains why they too are slow. A method of decreasing the Hough Transformation time is detailed, and the reduction in calculation time from this implementation is quantified.

The remaining parts of the long-bone segmentation and fracture detection algorithm were relatively fast to compute, so their calculation speed did not need to be reviewed.

8.1 Why smoothing is slow

As stated in Section 4.5 on page 69, the nonlinear partial differential equations for the AMSS could not be solved analytically, so a discrete numerical method was implemented. Depending on the final scale of smoothing t that was required, the smoothing process could require a large number of iterations of Equation 4.20 on page 71, with the number of iterations n determined by Equation 4.21. The relationship between t and n was shown in Figure 4.12 on page 72. For example, smoothing an image to the scale $t = 20$, with a scale step of $\triangle t = 0.1$ required $n = 407$ iterations. During a typical smoothing iteration, every pixel in the x x y image had to be iteratively analysed. The gradient at that point was calculated using the eight neighbouring pixels of the 3 x 3 stencil, which was then used to calculate $I_{\xi\xi}$ and therefore the smoothed image. As a result, the smoothing required a set of calculations to be performed at x x y x n pixels, so that a typical 3600 x 1200 pixel image smoothed to scale $t = 20$, required analysis of close to 1.85 billion pixels. Testing[1] revealed that smoothing any of the development images (using a single processor) took around 27 minutes to complete.

8.2 Decreasing the smoothing time

A method of decreasing the smoothing time was required, since 27 minutes was an unacceptably long time. Achieving the same level of smoothing in a shorter period of time could be accomplished in a number of ways:

1. Increase the scale step $\triangle t$ so that each iteration produced a greater amount of smoothing, thereby requiring a smaller total number of iterations. The problem with this method was that increasing $\triangle t$ beyond the value 0.1 resulted in instability, since $\triangle t$ was already at its maximum (as discussed in Section 4.5 on page 69). This was therefore not an adequate solution.

2. Reduce the size of the region to be smoothed. Unfortunately this was not practical in many cases, since the bone often filled the entire image.

[1] Performed on a Sun Microsystems Dual 1.8GHz AMD Opteron system with 1MB level 2 cache, 8GB RAM, running RedHat ES 3 (64bit mode)

3. Reduce the image resolution, such that there were fewer pixels that had to be analysed in each iteration. However this removal of image information could compromise the fracture detection process, due to interpolation effects and more importantly, insufficient image resolution.

All of these methods attempted to minimise the total number of pixels that had to be analysed, although they also produced inferior results. Rather than reducing the total number of pixels that had to be analysed, a better solution was to use a multi-processor system in which the computation load was shared such that each processor analysed a smaller number of pixels, thus completing more rapidly. This type of parallelisation was implemented to reduce the smoothing time.

8.2.1 Smoothing parallelisation and boundary extension

In a computer cluster containing multiple machines, each containing multiple processors (or alternatively, on newer computers containing multi-core CPUs), a large number of calculations can be performed simultaneously. For example, if a large number of tests needed to be performed on a single image, the tests could be split up so that each processor (or core) simultaneously performed a separate test, thereby drastically reducing calculation time. Unfortunately the AMSS algorithm could not be easily adapted for multiple processors, because the input for each iteration was the output of the previous one. A seemingly simple solution to this problem was to split the input image into multiple smaller image stripes, smooth these smaller stripes on separate processors and then recombine them to produce the output. An example of the type of image that resulted from splitting the image into two stripes is shown in Figure 8.1. It compares the image obtained by smoothing the complete image with the image obtained by smoothing two stripes, and shows that the two are not the same.

As this example demonstrates, the results produced by the two methods were not identical. While a large proportion of the pixels in the images in Figure 8.1b and 8.1c appeared to be the same, there was a distinct vertical line through the centre of image 8.1c that resulted from the edge effects that were introduced by splitting the image. During the first iteration of the smoothing, the value of any chosen pixel

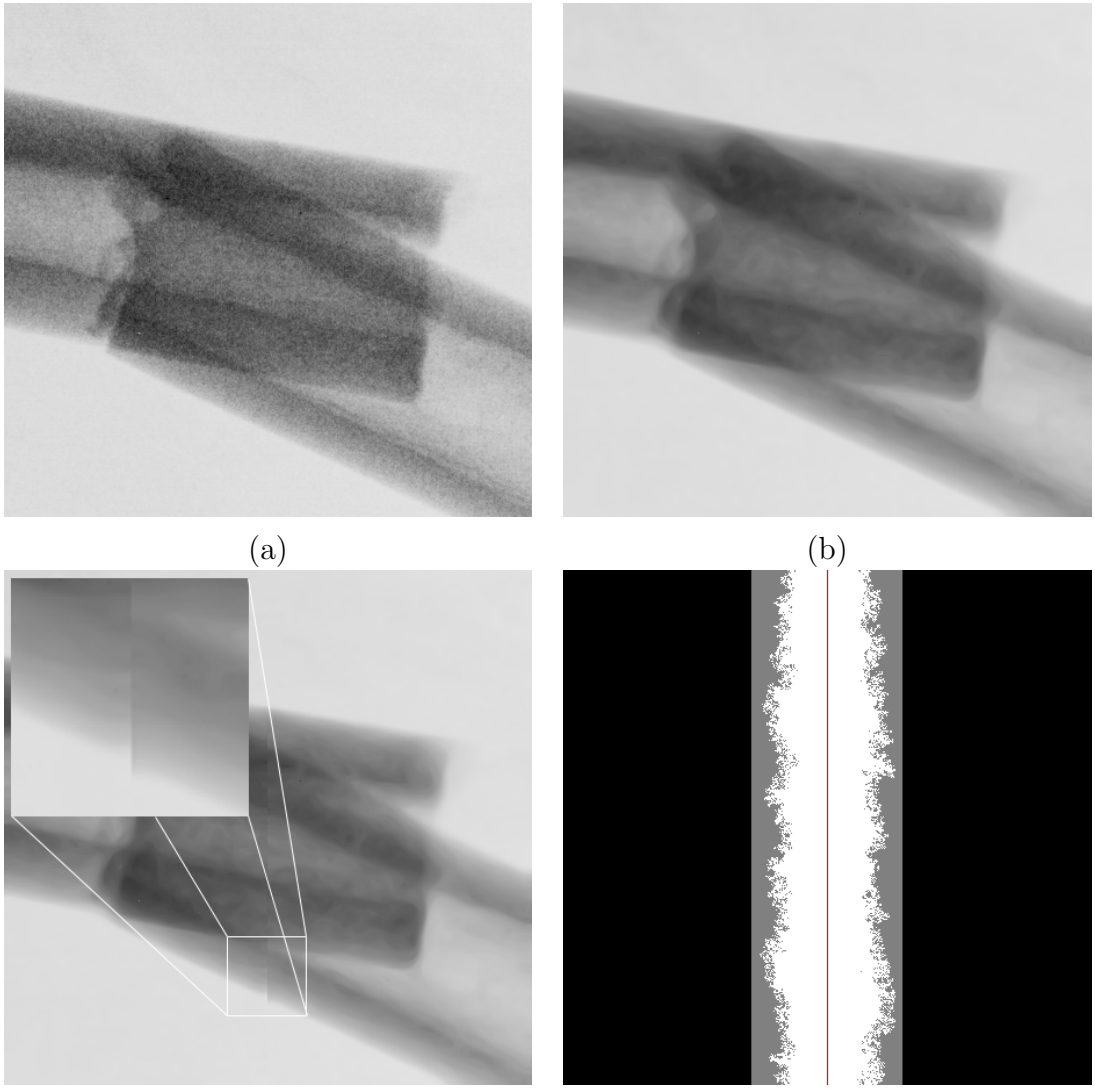


Figure 8.1: A comparison of AMSS smoothings. (a) The input image. (b) The standard AMSS smoothing to a scale $t = 5$ ($n = 64$ iterations). (c) Dividing the image into two parts and smoothing each separately before combining to produce the output image. The vertical line down the centre of the image indicates that the resulting images are not the same. This is due to the effects of diffusion across the image boundaries, and becomes more noticeable as the scale increases. (d) The white pixels are those points at which these two methods produce different values. The grey area is the region in which the two methods can theoretically produce different values. In the black region, the two methods should theoretically produce the same values.

(x, y) in the image was modified based on the values of the 8 pixels in the immediate 3 x 3 neighbourhood surrounding that pixel. However as the smoothing proceeded, the values of the surrounding pixels were also modified, thus influencing how the value of that chosen pixel(x, y) changed in later iterations. The distance at which a pixel would affect the value of that chosen pixel was directly proportional to the number of iterations n performed. When the image was split, some of the pixels that would normally influence the value of the chosen pixel (x, y) were removed, resulting in a different smoothing. This was most noticeable as a mismatch in contours when the images were recombined to produce the output.

Thus, in order to ensure that a pixel reached its correct value, the surrounding pixels had to be retained using a process termed boundary extension. The number of surrounding pixels to be retained was determined by the number of iterations n that had to be performed, and therefore the scale of smoothing t (Figure 4.12 on page 72). The white pixels in Figure 8.1d corresponded to the locations at which the pixel values in b and c were not identical. The grey region extended $n = 64$ pixels on either side of the red line along which the image was split—since 64 iterations were used—and was the region in which different values could theoretically be produced. The white region always lay within the grey region.

The correct smoothing could be produced by including all those pixels within the grey region, when creating the stripes from the original image. For example, when dividing the image in Figure 8.2a into two stripes, 8.2c and 8.2d along the vertical white line, $n = 64$ extra pixels were added to the respective images at that split point. The extra pixels added to the right side of 8.2c ensured that the entire region left of the line smoothed correctly, and similarly the pixels added to the left side of 8.2d ensured that the region to the right of the line smoothed correctly. Of course, the smoothing results within these two extra areas were incorrect, although both of these were discarded before the two stripes were combined to produce the output, since the incorrect part of 8.2d was correctly represented in 8.2e and vice versa. When multiple processors were used, and the two smoothings occured simultaneously, the calculation time was almost halved.

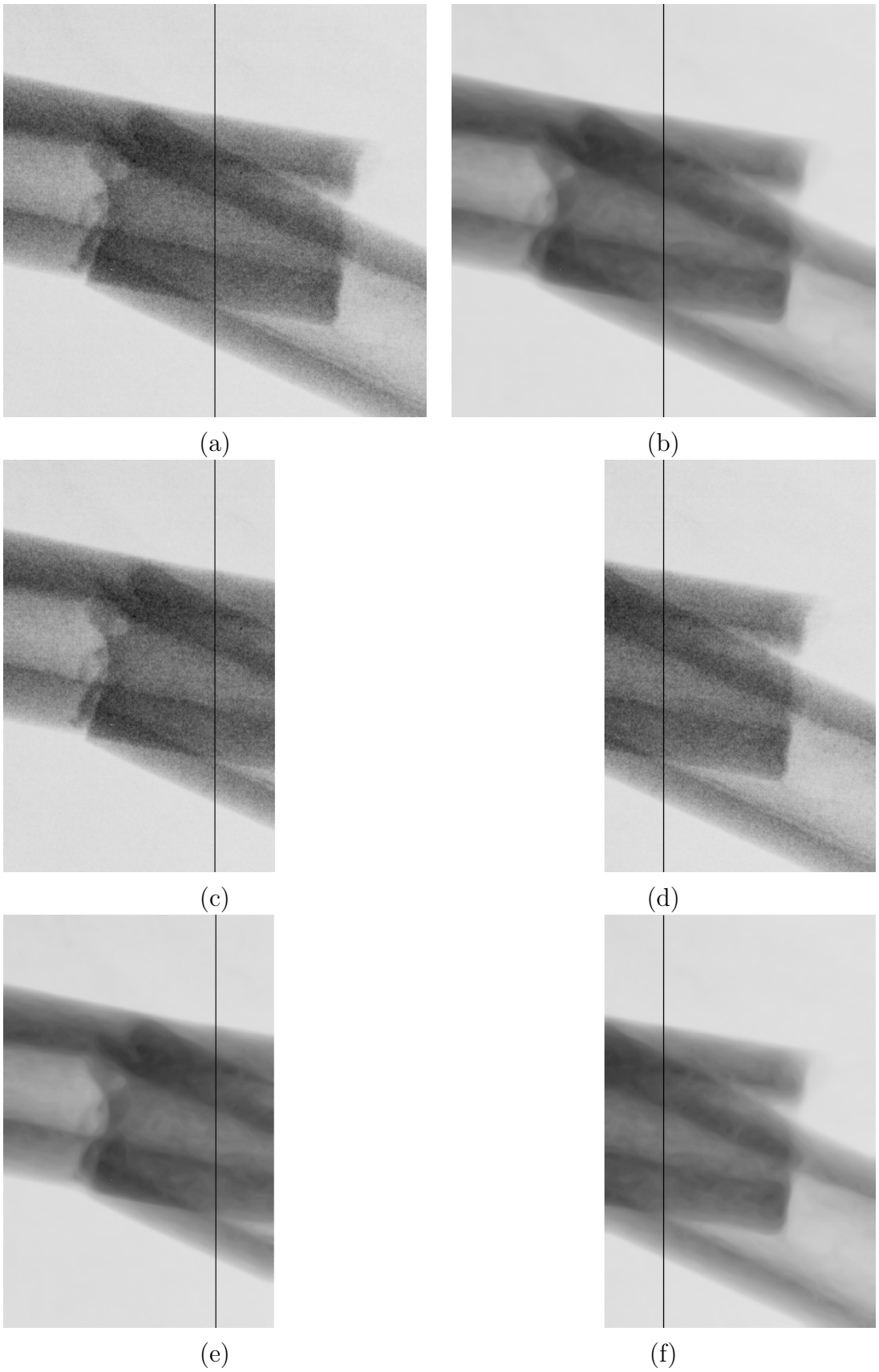


Figure 8.2: The black lines illustrate how an image (a) could be split into (c) left and (d) right sub-images, by including $n = 64$ extra pixels at the split location, such that the correct smoothing (b) at scale $t = 5$ can be obtained after combining the separately smoothed images (e) and (f).

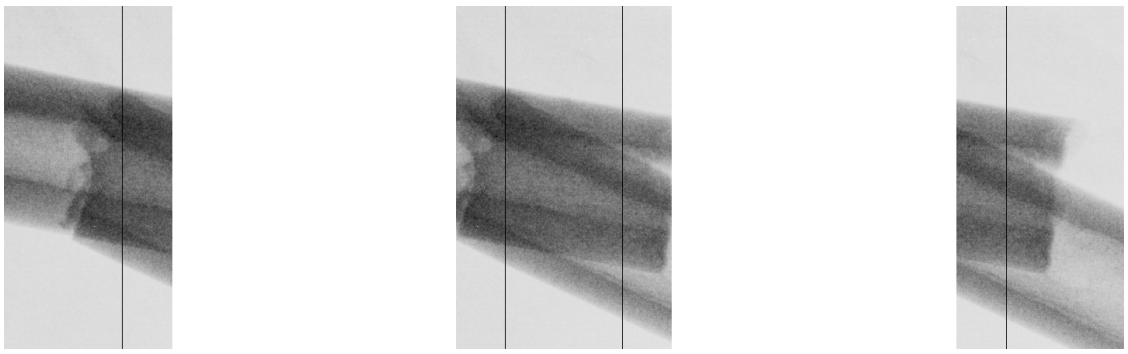
It was possible to perform this image splitting because the AMSS is non-linear and the smoothing has a finite speed of propagation. This is distinct from a linear smoothing equation, where the propagation speed is infinite.

8.2.2 How to split the image

The possibilities were not limited to splitting the image into only two stripes, indeed a larger number of image stripes could result in a quicker smoothing in some cases. In addition, tiles could be created instead of stripes, so that both dimensions of the image were split. The number of tiles or stripes was limited by:

1. The number of processors P available to simultaneously smooth the images. For example, it would not be sensible to split an image into 100 tiles if there were only two processors available in the multi-processor system. This was because the overheads involved in splitting the images (as described above), and the additional calculations that would have to be performed on all the extra pixels included at the split points (for each of the 100 images), would far outweigh any benefit obtained from the parallelisation. Indeed, tests showed that the maximum number of tiles/stripes that should be created was determined by the number of available processors, so that each processor only smoothed a single tile/stripe. The cluster that was used for developing and evaluating the parallel smoothing algorithms consisted of 8 dual processor machines, containing a total of $P = 16$ processors. As a result, when splitting an image, the maximum number of tiles/stripes was limited to 16.

2. The scale of smoothing t. By way of illustration, if smoothing the small 456 x 444 example image in Figure 8.2a to a scale of $t = 20$—thus requiring $n = 407$ iterations—it would not have been sensible to split the image into a large number of tiles/stripes. This was because at each point at which the image was split, 407 pixels had to be added to produce the correct output, and this was larger than the number of pixels available, even if the image was only split into two stripes of width 228 pixels. The maximum number of tiles L_x and L_y (in the x and y directions, respectively) into which an image could be split was dependent on the size of



(a) (b) (c)

Figure 8.3: Equation 8.1 shows that for a 456 x 444 example image smoothed to scale $t = 5$, the maximum number of image tiles was $L_x = 3$, as shown here.

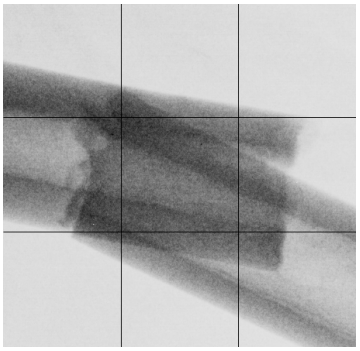
that image, and could be calculated using:

$$L_x = \left\lfloor \frac{x}{2n} \right\rfloor \qquad L_y = \left\lfloor \frac{y}{2n} \right\rfloor \tag{8.1}$$

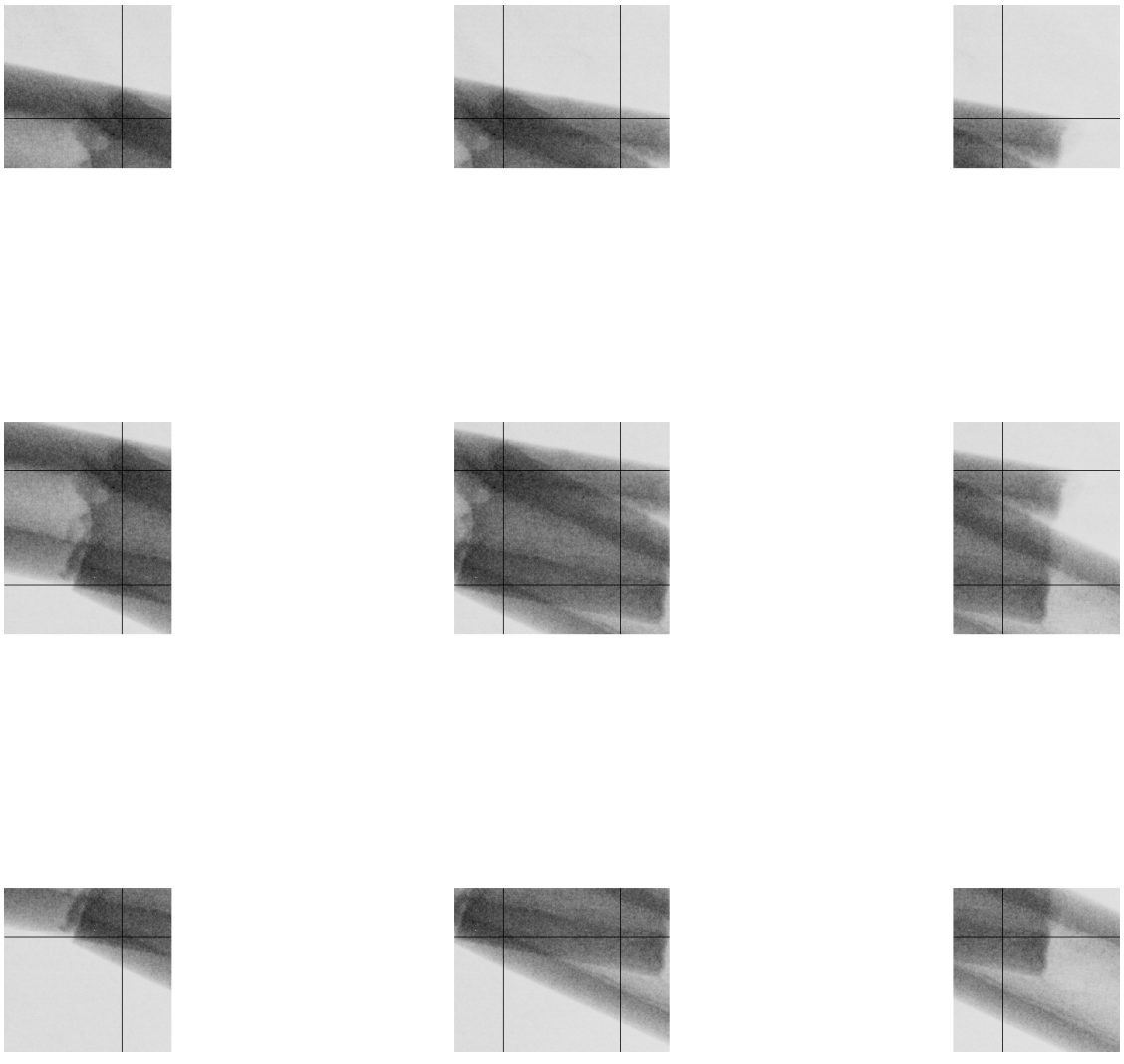
Thus for the example image with $x = 456$, $y = 444$ and $t = 20$ ($n = 407$), the maximum number of sub-images to create was actually $L_x = 0$ and $L_y = 0$. That is, the image should not be split before smoothing. However, if the image was to be smoothed to the scale $t = 5$ ($n = 64$), the maximum number of sub-images that could have been produced would be $L_x = 3$, as shown in Figure 8.3.

The image could also be split into tiles using a number of different patterns. The demonstrations so far only involved creating vertical stripes, although it was also possible to make square rather than rectangular tiles, by dividing both dimensions. Of course, the number of horizontal divisions L_x and vertical divisions L_y did not necessarily have to be equal. An example of splitting the same image into nine tiles is shown in Figure 8.4.

The tiles at the image boundary did not require extension, while those in the middle required extension on all four sides, so different size tiles were produced. As a result, the middle tiles took much longer to smooth than the boundary (especially corner) tiles. However, the maximum speed increase occured when the tiles were all approximately the same size, so the splitting method was modified to take this into account. The following two equations were produced for $L_y > 2$, in which B_h was the height of the



(a)



(b)

Figure 8.4: The example image (a) could also be split into tiles (b) by dividing both the horizontal and vertical dimensions into L_h and L_v split points respectively. Note how the resulting images differed in size and therefore took different lengths of time to smooth, with the centre tile being the longest and the corner tiles being the shortest.

boundary tiles, and C_h was the height of the centre tiles before boundary extension:

$$B_h + n = C_h + 2n \qquad y = 2B_h + (L_y - 2)\,C_h \tag{8.2}$$

Which was simplified to give expressions for B_h and C_h:

$$B_h = \left\lfloor \frac{y + n\,(L_y - 2)}{L_y} \right\rfloor \qquad C_h = \left\lfloor \frac{y - 2n}{L_y} \right\rfloor \tag{8.3}$$

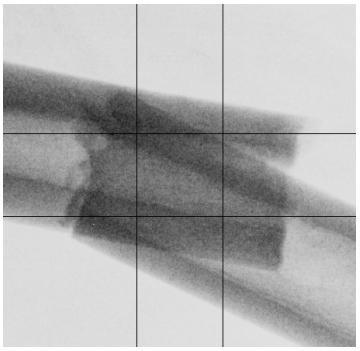
Similarly in the horizontal direction for the width of the boundary and centre tiles, B_w and C_w respectively:

$$B_w = \left\lfloor \frac{x + n\,(L_x - 2)}{L_x} \right\rfloor \qquad C_w = \left\lfloor \frac{x - 2n}{L_x} \right\rfloor \tag{8.4}$$

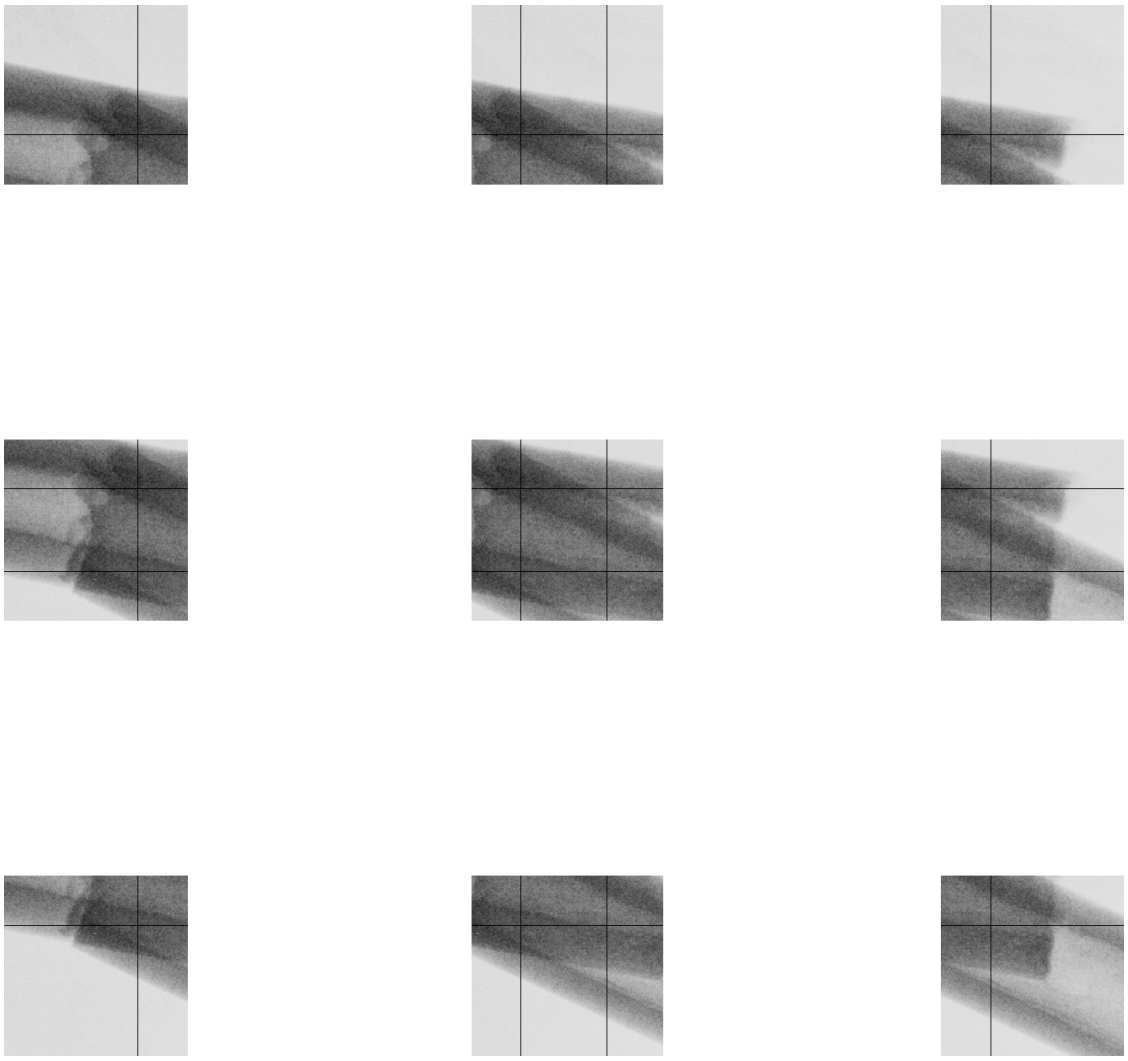
An example of the same image split into equal size tiles using Equations 8.3 and 8.4 is shown in Figure 8.5, with $B_h = 169$, $C_h = 105$, $B_w = 173$ and $C_w = 109$ before boundary extension. While the lines in 8.5a produced regions of greatly differing size, the addition of n pixels to the boundaries at which the image was split created tiles that were nearly identical in size. Since an integer number of pixels was required for each tile, the equations round down to the nearest whole number of pixels. Then, if necessary, the number of pixels in the last tile was increased so that all pixels in the image were included. For example, tiles of width 173 pixels, 109 pixels and 174 pixels (increased by one pixel to include the entire image) were used for the example image. The amount of time required for smoothing was then almost identical for all nine tiles.

For comparison purposes each of these methods was tested on a single processor[2] to determine the relative speed of smoothing per stripe/tile, with the results shown in Table 8.1. All the methods of creating stripes or tiles resulted in much faster smoothing (at least two times as fast) per stripe or tile than smoothing the complete image. As expected, smoothing nine tiles was also significantly faster than smoothing three stripes, but for the nine uneven tiles there was significant variation in smoothing time due to the different tile sizes (shown in Figure 8.4). When all nine tiles were the

[2]Performed on a Sun Microsystems Dual 1.8GHz AMD Opteron system with 1MB level 2 cache, 8GB RAM, running RedHat ES 3 (64bit mode)



(a)



(b)

Figure 8.5: As an alternative to Figure 8.4, the image could be divided so that the resulting tiles were all approximately the same size, so that the smoothing of each took approximately the same amount of time.

Method	Figure	smoothing time (sec)	% of standard
Original Image	8.1a	11.53*	100
Three vertical stripes	8.3a/c—outer stripes	5.5	47.7
	8.3b—centre stripe	7.04*	61.1
Nine uneven tiles	8.4b—corner tiles	2.61	22.6
	8.4b—centre tile	4.32*	37.5
Nine even tiles	8.5b—all tiles	3.14*	27.2

Table 8.1: Comparison of the amount of time required per stripe or tile for AMSS smoothing to scale $t = 5$ using the various splitting methods identified. Since the total smoothing time was determined by the largest time (*) for a particular method, using nine even tiles produced the fastest smoothing of all the methods tested.

same size (shown in Figure 8.5), they all took almost identical amounts of time to smooth, and this time was less than the maximum smoothing time for the unevenly sized tiles. These results indicate that to produce the fastest smoothing (for an image of approximately this size) the image should be split into L_x by L_y tiles, and Equations 8.3 and 8.4 should be used to determine the size of those tiles.

Regardless of the method chosen by which to split the images, the two limitations identified at the start of this section on page 186 still applied. Unfortunately most x-ray images are very large, in this case 3600 x 1200 pixels, so the biggest limitation was generally the number of processors available for smoothing. As a result, the quickest smoothing was produced when the image was split into $P = 16$ equal sized vertical stripes.

8.2.3 Implementing the parallelised smoothing algorithm

To achieve the AMSS parallelisation described in the previous section, an algorithm containing the following steps was implemented in C:

1. Read the input image (in either the TIFF or raw format described in Section 4.5 on page 69), and the file containing the scale information (also in raw format).

2. Determine the required overlap by calculating the number of iterations n of the AMSS equation, based on the maximum scale required.

3. Determine the number of stripes L_x (or L_y) to split the image into using Equation 8.1. Modify this value if it is larger than P, the number of available processors.

4. Determine the width (or height) of the boundary and centre stripes using Equation 8.4 (or 8.3), and if needed increase the size of the last stripe so that the entire image is covered.

5. For each stripe, determine the start and end coordinates based on the stripe widths and the overlap required, before saving the stripe data in the same format as the input image (either TIFF or raw).

For an image *test.tif*, the result was a series of L_x or L_y images of almost identical size, with modified file names *s001_test.tif* onwards. Each of these images could individually have the AMSS algorithm applied to it, before being combined in the reverse process. The process of splitting an image into separate files, submitting each of these jobs for processing, and combining the results was fully automated using a set of shell scripts. To allow each job to run simultaneously on its own processor the Sun Microsystems Grid Engine software was installed on the cluster to completely control the job allocation and submission.

8.2.4 Analysis of the parallelised smoothing algorithm

Analysis of the parallelised smoothing algorithm was performed in two parts. The first tests were used to determine if the smoothed images produced using the parallelised method were indeed identical to those produced by the standard method. Scales $t = [1, 5, 20]$ were chosen for testing—since the scales $t_1 = 5$ and $t_2 = 20$ were used in the long-bone segmentation and fracture detection algorithms—and a series of three test images (including the image in Figure 8.1a) were used. The maximum value of the absolute difference between the smoothed images created by the two methods was examined. A non-zero result would indicate that some pixels attained different values by the two methods, while a zero result would indicate identical smoothing results. All nine images produced a zero result, indicating that the parallelised method did indeed produce identical results to the standard AMSS implementation.

The second series of tests measured the length of time taken by the standard and parallelised methods to smooth the images to a range of scales. The test image was taken from the development set, and was a 3600 x 1200 pixel x-ray of a midshaft

femur fracture. While the choice of image did affect the smoothing time slightly, the differences were only minor. The results are tabulated in Table C.1 on page C-2 and are displayed in Figures 8.6 and 8.7. They indicated that for all the tested scales the parallelised method was at least as quick as the standard non-parallelised AMSS smoothing method. Figure 8.7 showed that the speed increase was dependent on the scale, since this affected the number of tiles into which the image could be split, while still retaining a sufficient overlap n to produce the correct output. The maximum speed increase occurred at a scale of $t = 6$, for which the parallelised method was just over five times faster than the non-parallelised method. As shown in Table C.1 on page C-2, this was the largest scale at which all $P = 16$ processors in the cluster could be simultaneously utilised.

8.2.5 Reducing the size of the overlap

Figure 8.1d on page 188 showed that splitting an image prior to smoothing produced a region where the smoothed image was not correct, if boundary extension was not performed. The gray area of width n pixels either side of the split point was noted to be the region in which different values could theoretically occur. The magnitude of this difference was not shown in Figure 8.1d, but is shown in Figure 8.8a, for the same two images shown in Figures 8.1b and 8.1c. As the colour bar indicated, regions with higher values (those that were redder) were those which contained greater differences between the two images. Note that the colour bar values were scaled rather than absolute values. The largest differences were close to the split line, while the values further away were much smaller. In addition, the greatest differences occured at those points along the split line where the gradient was largest, that is, the bone edges.

Figure 8.8b shows a vertical projection of the region between the white dotted lines in 8.8a, obtained by summing all the pixels in each column. Again, the largest differences occurred at the location where the images were split, with much smaller differences away from the split point. The blue dashed lines in 8.8b are located $n = 64$ pixels either side of the split point, and indicate the region that would be retained to ensure that the resulting images were identical, as described in Section 8.2.1. Indeed,

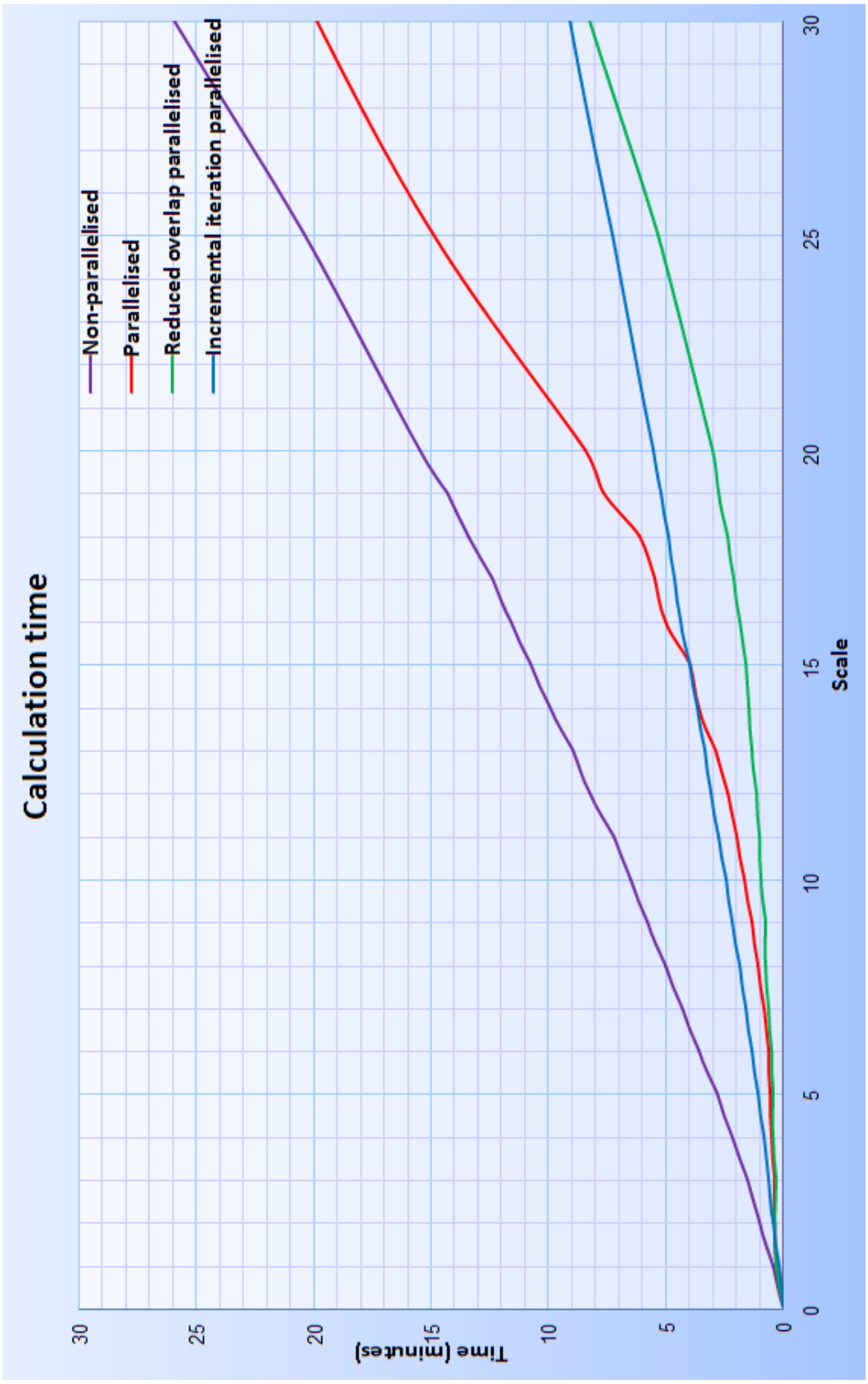


Figure 8.6: Results of the comparison between the three parallelised and one non-parallelised smoothing methods. The calculation times showed that the three parallelised methods were all faster than the non-parallelised method, for all tested scales.

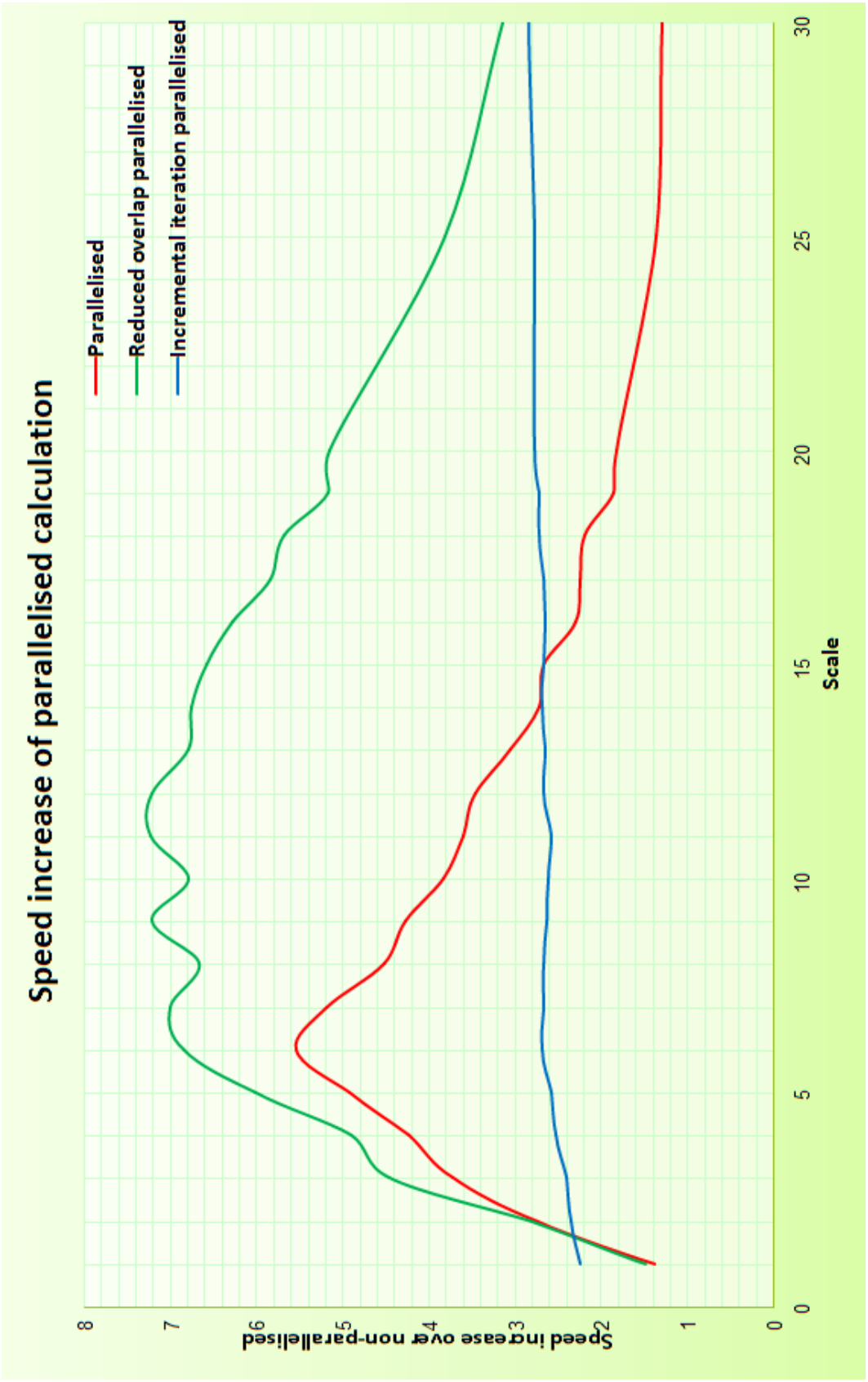


Figure 8.7: Results of the comparison between the three parallelised and one non-parallelised smoothing methods. The speed increase resulting from the various parallelisations was measured relative to the non-parallelised method.

further than $n = 64$ pixels either side of the split point the difference was zero. However for much of the region closer to the split point the difference was also very close to zero, indicating the possibility of decreasing the overlap to be less than n.

Although it was possible to decrease the overlap to improve the calculation time, the problem was choosing how much to decrease it by. Examination of Figure 8.8b showed that if the overlap was reduced to $\left\lfloor \frac{n}{3} \right\rfloor = 21$ pixels, then although the resulting image was not identical to the unsplit image, the maximum pixel difference was 3.14 x 10^{-4}. This was less than 0.1% of the maximum pixel difference that resulted from the AMSS smoothing. Decreasing the overlap any further began to produce larger differences between the two images, and was therefore not appropriate. Since the chosen value $\left\lfloor \frac{n}{3} \right\rfloor$ was dependent on n it also automatically varied with the chosen scale t. Regardless of the image to be smoothed, or the final scale chosen, the point $\left\lfloor \frac{n}{3} \right\rfloor$ was always on the flat portion of the projection curve.

Altering the amount of overlap needed also required modification of Equations 8.1, 8.3 and 8.4. The maximum number of horizontal and vertical tiles that the image could be split into was modified to:

$$L_x = \left\lfloor \frac{3x}{2n} \right\rfloor \qquad L_y = \left\lfloor \frac{3y}{2n} \right\rfloor \tag{8.5}$$

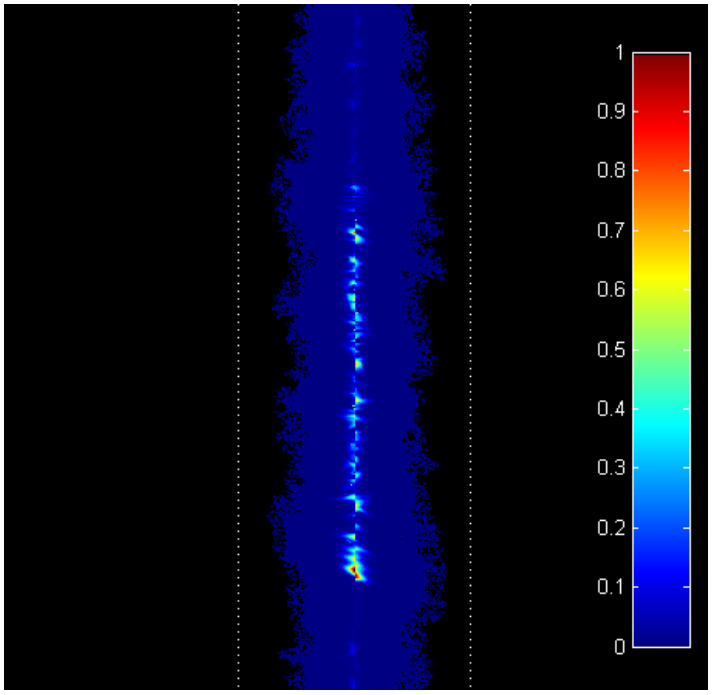
So the corresponding heights of the boundary (B_h) and centre (C_h) tiles then became:

$$B_h = \frac{y + \frac{n}{3}(L_y - 2)}{L_y} \qquad C_h = \frac{y - \frac{2n}{3}}{L_y} \tag{8.6}$$

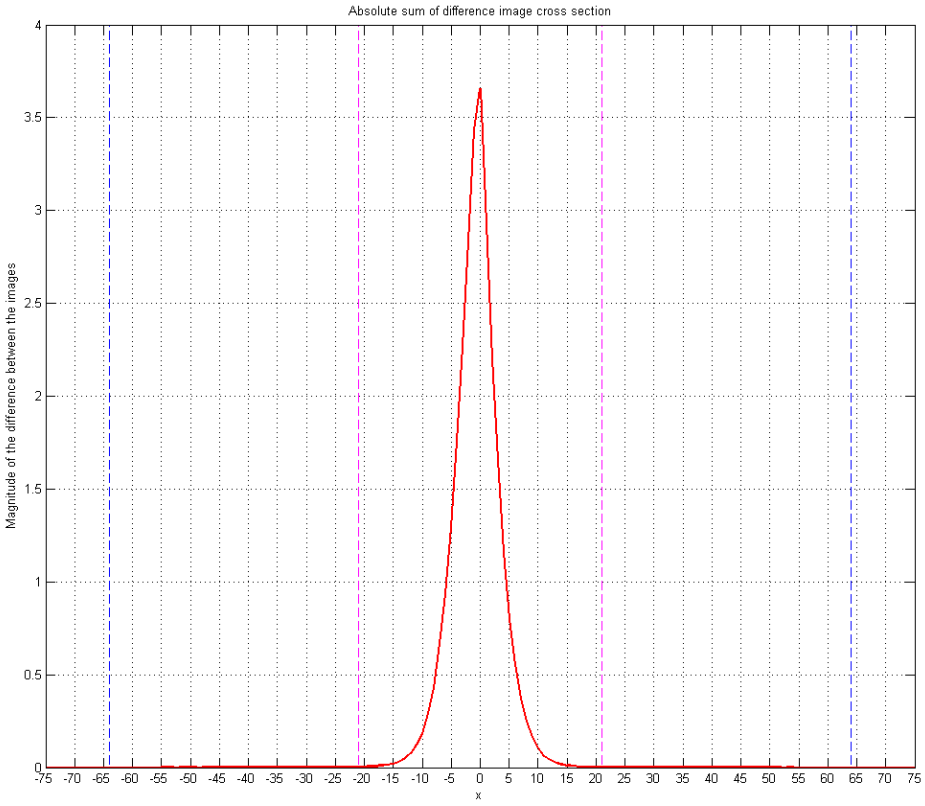
Finally, the corresponding widths of the boundary (B_w) and centre (C_w) tiles became:

$$B_w = \frac{x + \frac{n}{3}(L_x - 2)}{L_x} \qquad C_w = \frac{x - \frac{2n}{3}}{L_x} \tag{8.7}$$

This meant that for the 456 x 444 test image smoothed to a scale $t = 5$ ($n = 64$) as shown in Figure 8.2a, the maximum number of sub-images could be increased from $L_x = 3$ and $L_y = 3$, to $L_x = 10$ and $L_y = 10$. Of course, as mentioned in Section 8.2.2, the number of tiles or stripes was still limited by P, the number of available processors, so producing 100 tiles from this image would not lead to a faster smoothing time than



(a)



(b)

Figure 8.8: (a) The same image as 8.1d on page 188, but colored to show where the largest absolute differences occured. Note that the image was normalised to show the differences on the scale $[0,1]$*. A projection of the region between the white dotted lines is shown in (b). Again this showed that the largest differences were located close to point at which the image was split. The blue dashed lines show the points* $n = 64$ *pixels either side of the split point, while the pink dashed lines show the points* $\lfloor \frac{n}{3} \rfloor = 21$ *pixels either side of the split point.*

splitting it into $P = 16$ tiles. However, splitting the image into 16 tiles rather than 9 did produce a significantly faster smoothing. Additionally, decreasing the overlap allowed higher scales to benefit from the use of all P processors.

8.2.6 Analysis of the reduced overlap smoothing algorithm

The tests described in Section 8.2.4 were run again, changing only the overlap from n to $\left\lfloor \frac{n}{3} \right\rfloor$, to determine the effect of reducing the amount of overlap. The same test image of a 3600 x 1200 pixel x-ray of a midshaft femur fracture from Section 8.2.4 was used. The images at scales $t = [1, 5, 20]$ were again compared to the images smoothed by the non-parallelised AMSS method, to determine the effect of reducing the overlap. In all cases there were certainly no perceptible differences between the images, but a closer comparison revealed that, as expected, some pixels had slightly different values in the two images. Again though, the largest pixel difference was less than 0.1% of the maximum pixel difference that resulted from the AMSS smoothing. This meant that the reduced overlap parallelisation was not completely accurate, because it was not producing exactly the same output as the standard method described in Chapter 4.

The calculation times for the reduced overlap parallelisation are shown in Table C.2 on page C-3 and plotted in Figures 8.6 and 8.7. As expected, the calculation time was much shorter than both the non-parallelised version and the standard parallelisation tested in Section 8.2.4. This time the maximum speed increase occurred at a scale of $t = 11$, for which the reduced overlap parallelised method was just over seven times faster than the non-parallelised method. Tables C.1 on page C-2 and C.2 on page C-3 showed that the number of stripes that could be utilised at higher scales was much greater when using the reduced overlap method, since the last scale at which all $P = 16$ processors were utilised increased from $t = 6$ to $t = 15$. This reduction in the number of processors that could be utilised at high scales was the cause of the oscillations in the traces in Figure 8.7 for the parallelised and reduced-overlap methods. For both of these methods, the number of stripes that could be utilised, and therefore the calculation time, were determined by the final scale of smoothing. The calculation times

for the parallelised and reduced overlap methods tended toward the non-parallelised calculation time as t increased, and were actually equivalent for scales above $t = 54$ and $t = 122$ respectively.

These values were also affected by the image size, with smaller images less able to utilise the maximum number of processors. Fortunately, the reduced overlap method coped with small images better than the standard parallelisation, because more stripes could be utilised.

8.2.7 Incremental iteration parallelisation

The results presented so far in this section clearly show that parallelisation had the potential to significantly increase the speed of the AMSS smoothing process. In Section 8.2.5 it was shown that it was possible to reduce the size of the overlap thereby trading some accuracy for an increase in calculation speed. This method essentially allowed higher scales to utilise all P processors, because Equation 8.5 permitted the image to be split into a larger number of tiles than Equation 8.1. To obtain a speed increase without trading accuracy, one final method of parallelisation was investigated.

The standard parallelisation and the reduced overlap parallelisation were based on Equations 8.1 and 8.5 respectively. These were used to calculate the number of tiles into which the image should be split, based on the final scale of smoothing t, and therefore the number of iterations n of the smoothing equation that were required. Using this method, choosing a final scale of $t = 20$ meant that an overlap of $n = 407$ pixels was required, so a 3600 x 1200 image was initially split into only $L_x = 3$ stripes to maintain the correct overlap at the final scale. Since the AMSS was an iterative process, all the scales $t = \{1, 2, 3, ..., 18, 19\}$ had to be calculated before $t = 20$. In fact, because any given scale was derived from the previous scale, only a relatively small number of incremental iterations were required to obtain the next scale from the previous one. Capitalising on this fact allowed all $P = 16$ processors to be utilised more effectively.

Table 8.2 shows that obtaining the image at scale $t = 20$ required only $\Delta n = 27$ iterations of Equation 4.20 applied to the image at scale $t = 19$, compared to $n = 407$ iterations applied to the original image. In Section 8.2.1 it was established that the

Scale	Iterations		Maximum number of stripes L_x		
t	n Total	Δn Incremental	Parallelised	Reduced overlap	Incremental iteration
1	8	8	16	16	16
2	19	11	16	16	16
3	32	14	16	16	16
4	48	15	16	16	16
5	64	17	16	16	16
6	82	18	16	16	16
7	100	19	16	16	16
8	120	20	15	16	16
9	140	20	12	16	16
10	162	21	11	16	16
11	183	22	9	16	16
12	206	23	8	16	16
13	229	23	7	16	16
14	253	24	7	16	16
15	277	24	6	16	16
16	302	25	5	16	16
17	328	25	5	16	16
18	354	26	5	15	16
19	380	26	4	14	16
20	407	27	4	13	16
25	548	29	3	9	16
30	699	31	2	7	16

Table 8.2: The number of stripes that could be utilised by the standard parallelisation, reduced overlap parallelisation and the incremental iteration parallelisation. These Figures were calculated assuming that $P = 16$ processors were available, and that the input image was of size 3600 x 1200 pixels.

required overlap was determined by the number of iterations. Therefore reducing the overlap from $n = 407$ pixels to $\Delta n = 27$ pixels produced a substantial increase in the number of stripes that could be used at scale $t = 20$. Table 8.2 also shows that when using this method, even at scale $t = 30$, the maximum number of stripes that could be utilised was still $P = 16$. Indeed if it were possible to increase the number of available processors, this method could use up to $P = 58$ processors at scale $t = 30$, while the standard and reduced overlap methods could only utilise 2 and 7 processors respectively. When using $P = 16$ processors, it was not until scale $t = 1425$ that the number that could be utilised dropped from 16 to 15.

The downside of this method was that the complete image had to be recreated at each scale, so that the subsequent scale could be created by applying the appropriate number of incremental iterations (Δn). By contrast, the previous two methods only required the complete image to be produced at the final scale. This implied that in the incremental iteration method, after a particular scale was calculated, the stripes had to be combined and then immediately split for the next scale. That is, to produce the image at scale t, it was necessary to perform a total of t split and t combine operations, rather than single split and combine operations with the previous two methods. Since the split/combine operations were disk intensive (and on average took a total of 6 seconds to perform for $L_x = 16$ stripes) they contributed significantly to the overall calculation time.

8.2.8 Analysis of the incremental iteration smoothing algorithm

The tests described in Section 8.2.4 were run again, this time utilising the incremental iteration parallelisation. Again the images at scales $t = [1, 5, 20]$ were compared to the images smoothed by the non-parallelised AMSS method, to determine if the incremental iteration method produced the same results. When the intermediate images were written as TIFF files (Section 8.2.3, step 5) there was a perceptible difference at higher scales, as rounding errors at each scale began to accumulate. If the images were written as raw files, thereby retaining full accuracy, the results were identical to the

non-parallelised method. Using the raw format did increase calculation time slightly since the raw files were approximately ten times as large as the TIFF files, and took longer to both read and write.

The calculation times for the incremental iteration parallelisation are shown in Table C.3 on page C-4 and are plotted in Figures 8.6 and 8.7. As expected, the calculation time was much shorter than the non-parallelised version, but was not always shorter than the standard parallelisation and reduced overlap parallelisation implementations tested in Sections 8.2.4 and 8.2.6. In fact, the incremental iteration method was slower than the reduced overlap parallelisation at all tested scales, and slower than the standard parallelisation at scales less than $t = 15$. Interestingly, because all $p = 16$ processors could be utilised at all scales, the resulting trace was much smoother than for the other two methods. In addition, the speed increase was almost independent of scale, remaining relatively constant at just under three times as fast as the non-parallelised method. Unlike the previous two methods of parallelisation, the calculation time did not tend toward the non-parallelised calculation time within any useful range of t. By comparison, the calculation times for this method would only become equivalent at scales above $t = 3.375$ x 10^6, which is four to five magnitudes greater than the largest usable scale. Thus for high scales (approximately $t > 30$) the incremental iteration parallelisation was clearly the best choice.

8.3 Smoothing algorithm parallelisation conclusions

Other possibilities for parallelisation involved combining the strengths of the standard and incremental iteration parallelisations, and minimising their weaknesses. The standard method of parallelisation was not overly disk intensive (apart from single split and combine operations), but due to the large overlaps for each stripe it was very processor intensive and could not always utilise all $P = 16$ processors. On the other hand, the incremental iteration parallelisation was disk intensive because the stripes had to be combined and then re-split at every scale. However, due to the small overlaps required it was not as processor intensive and all $P = 16$ processors could be utilised. In the future, a suitable trade-off could be to combine the images at every two, three

or four scales so that the disk use is decreased and all processors can still be utilised at all scales. There are many possible combinations of ways in which this could be performed, but these were not tested here.

The results shown in Figure 8.6 on page 199 indicated that parallelisation of the AMSS algorithm had the potential to significantly decrease the smoothing time. Of the three methods discussed in Section 8.2, the reduced overlap parallelisation (Section 8.2.5) had the greatest potential to decrease the calculation time, but also had the drawback that it did not retain the full accuracy of the standard method. Therefore the standard and incremental iteration parallelisations were more appropriate methods of decreasing the calculation time. There were, however, drawbacks to both methods. The standard parallelisation was highly dependent on scale and only produced good speed increases at low scales, while the incremental iteration method was very disk intensive so overall did not produce as much of a speed improvement with the test hardware.

It was estimated that with better coding—including the use of compression during the data transfers between machines within the cluster to minimise the effect of network latency, as well as better memory management to minimise the number of disk intensive steps—that the incremental iteration method could have been almost as fast as the reduced overlap parallelisation at all scales. This would definitely have been the case when run on a multi-core (i. e. quad-core or 8-core) machine, where the system memory would shared between all the cores, the data transfers would happen at the clock speed of the board rather than at the speed of the local area network, and almost no file IO would be required. In any case, because it retained full accuracy and could smooth an image to the chosen scale $t_2 = 20$ faster than the standard parallelisation, it was the smoothing method of choice.

8.4 Why the Hough Transform is slow

As mentioned in Section 5.2.2 on page 88, the Hough Transform was very memory and computation intensive, since Equation 5.1 on page 87 had to be calculated at every pixel (x, y) for every θ in the range $[0, \pi]$. Various groups have attempted to decrease

the calculation time in a variety of ways.

One method of increasing performance is the fast Hough Transform (FHT), first described by Li et al. [58]. The limitation with the standard method described in Chapter 5 was that to find the parameters (ρ, θ) accurately, the parameter space had to be finely divided and an accumulator assigned to each block. When calculating the transform, these accumulator blocks were time consuming to fill. Rather than evenly dividing the parameter space into blocks, the FHT ignores those areas of the parameter space that are relatively devoid of votes by homing in on the solution. This results in a considerable speed increase and reduction in memory requirements. For higher-dimensional parameter spaces, the relative speed advantages of the FHT increase. Unfortunately a disadvantage of the FHT is that it can only supply one best-fit solution. In this case, the parameters for multiple local maxima (Section 5.4 on page 96) in the accumulator are required, rather than the parameters of a single global maxima. Thus unfortunately, the FHT was unsuitable for long-bone parameter approximation.

Other groups have attempted to create efficient Hough Transform algorithms such as the adaptive Hough Transform (AHT) described by Illingworth and Kittler [49], and the multi-resolution Hough Transform (MHT) described by Atiquzzaman [12]. In addition, Atiquzzaman [13] describes a pipelined implementation of the Hough Transform in a pyramid multi-processor, which produces a coarse to fine search for a single set of parameters. Again the detection of multiple peaks is not supported, so like the FHT this method was unsuitable. Other researchers have identified that since the Hough Transform involves many similar operations, parallel implementations could be used to improve performance, although no algorithms for performing this were located.

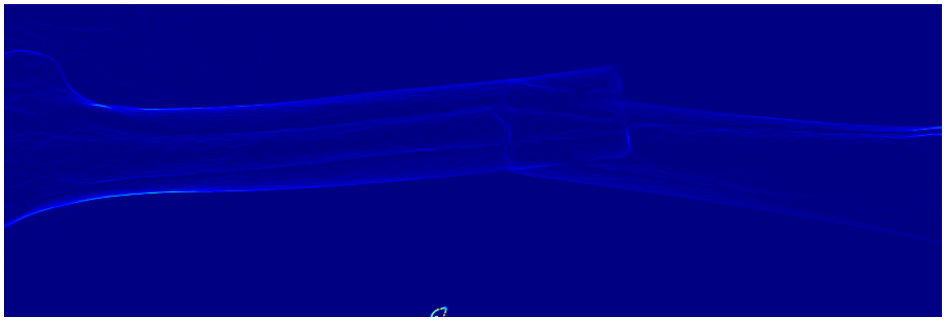
8.5 Increasing the speed of the Hough Transform

Rather than rewrite the Hough Transform algorithm or utilise a different scheme to calculate it, the aim was to simply to increase execution speed of the current algorithm, again through parallelisation. The calculation speed of the Hough Transform could be increased more easily than the AMSS algorithm, since it was not an iterative process

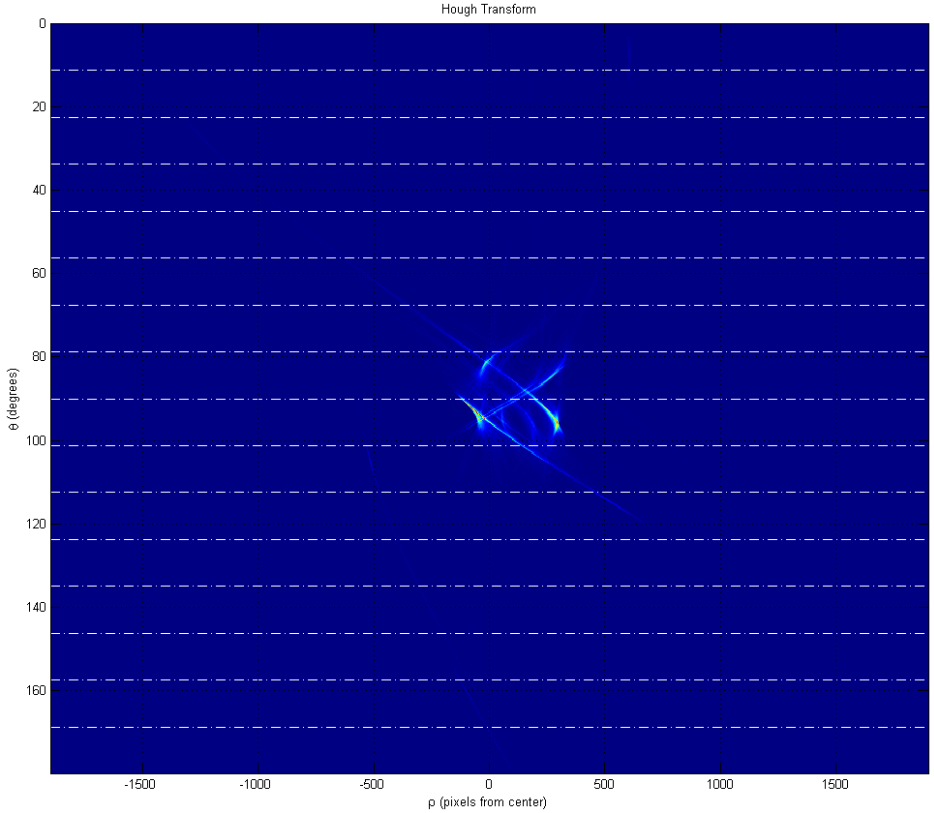
that relied on the output of the previous iteration to perform the next. As described in Section 5.3 on page 92, both the standard and modified Hough Transforms calculated Equation 5.1 on page 87 at every (x, y) in the image, for every value of θ. The two methods varied in the range of θ at which the equation had to be calculated. Unlike the AMSS algorithm, it was not suitable to split the image into tiles or stripes and determine the Hough Transform of each of these, because the resulting accumulators were of different dimensions, and the results could not be easily combined to produce the required output. Instead it was possible to split θ so that each processor only calculated Equation 5.1 at a smaller range of θ. For example, the un-parallelised method calculated Equation 5.1 at every θ in the range $0 \leq \theta < 180°$ at intervals of $0.25°$, requiring 720 calculations. This range was able to be split into $P = 16$ smaller ranges such that each spanned $11.5°$, still at an interval of $0.25°$, thus reducing the number of calculations of Equation 5.1 to 45 per processor.

8.5.1 Parallel Hough algorithm implementation

Unlike the AMSS algorithm which required boundary extension at the locations where the image was split, parallelisation of the Hough algorithm was much simpler, and consisted of splitting the file containing the θ information into P approximately equally sized smaller θ files. The same set of input images (gradient magnitude and direction) were used in each parallel process, while the θ files varied. As in the AMSS parallelisation, the Sun Grid Engine software was used to schedule and submit each job for processing. Once all processors had completed their transforms, the resulting images were simply assembled vertically edge to edge to create the complete Hough accumulator. Regardless of the input image, the Hough accumulator produced using the parallelised method was always identical to that created by the non-parallelised method. All the standard and modified Hough Transform images shown in this thesis were created using the parallelised method. An example demonstrating how the horizontal stripes were assembled is shown in Figure 8.9b.



(a)



(b)

Figure 8.9: Demonstration of parallelised Hough Transform. (a) The input gradient image, and (b) the resulting Hough Transform in which each horizontal stripe (shown between the dashed lines) was calculated on a separate processor.

8.5.2 Analysis of the parallelised Hough Transform

Calculating this image using the non-parallelised method took 58.0 seconds whereas the parallelised method took 21.6 seconds, about a 63% reduction in the total transform time. Interestingly the 2.7 times speed increase was not as dramatic as would be expected when utilising 16 processors, and was certainly not even close to 16 times faster. This was due to the design of the modified Hough Transform algorithm, and specifically the need to determine if the angle of the gradient ϕ at the pixel (x, y) lay within the specified θ range (as described in Sections 5.2.3 and 5.3). As a result, for any pixel this match could only occur on one of the P processors. The overhead required for checking the match range was significant, and contributed to the smaller than expected increase in performance. When parallelising the standard Hough Transform, in which each angle in θ was iterated and during every iteration Equation 5.1 was calculated at every (x, y), the speed increase was indeed much closer to 16 times. Calculating the standard Hough Transform of the same image in Figure 8.9a using the non-parallelised method took 37 minutes and 49 seconds, while the parallelised method with $P = 16$ took 2 minutes and 29 seconds, corresponding to a speed increase of just over 15 times. While the speed increase from parallelisation of the modified Hough Transform was not as great as expected, the calculation time of 21.6 seconds is still significantly faster than any of the other methods, being 105 times faster than the non-parallelised standard Hough Transform.

8.6 Hough Transform parallelisation conclusions

The results showed that, by utilising multiple processors in parallel, it was possible to significantly decrease the amount of time required to calculate the Hough Transform of an image, without sacrificing any accuracy. Unlike the standard Hough Transform, the speed increase from parallelising the modified Hough Transform did not change linearly with the number of processors P used. However, there was still an improvement in calculation time of close to three times. As mentioned, by using the parallelised modified Hough Transform over a standard non-parallelised transform, a speed increase

of greater than two orders of magnitude was possible.

8.7 Summary

Both the AMSS and the modified Hough Transform were very CPU intensive, and required large calculation times for standard image sizes. The AMSS smoothing was slow because a large number of iterations of equation 4.20 had to be performed to reach the desired scale t. Even on a fast computer this process took close to 30 minutes to complete. Three methods of reducing the overall smoothing time were identified, although they all traded accuracy for calculation time. The chosen solution was to perform parts of the calculation in parallel, so that the same accuracy could be retained while the calculation time was reduced.

Unfortunately parallelisation of the AMSS algorithm could not be achieved by simply splitting the image into stripes or tiles, due to the smoothing effects that occured across the boundaries. To ensure that the output was correct, boundary extension had to be performed wherever the image was split. The method by which the image should be split was examined, and it was shown that the smoothing times were the shortest when all tiles were the same size. The number and size of the tiles or stripes was determined by both the number of available processors, and the final scale of smoothing. The equations to determine the tile sizes were also derived, and their use demonstrated.

The parallelised AMSS algorithm was implemented in C, and tested to determine how effective it was. This implementation involved splitting the image into the required number of stripes, including the appropriate boundary extension, before each stripe was written as an individual file. Each file was smoothed on an individual processor in the cluster, and when all smoothings were complete the stripes were recombined to produce the output image. Firstly, the smoothed images obtained using the parallelised method were identical to the non-parallelised method in all cases. Secondly, the speed increase was measured to be a maximum of five times at the scale $t = 6$.

The effect of reducing the size of the boundary overlap was investigated, to determine how much more the speed could be increased. Although the smoothed images from the parallelised and non-parallelised methods were no longer identical, the speed

increased to a maximum of seven times at a scale $t = 11$, due to the greater utilisation of processors at higher scales. Increasing the calculation speed without sacrificing accuracy was achieved in the incremental iteration method, where the required overlap was modified and adapted for each scale, rather than just the maximum scale. This allowed much better utilisation of all processors and retained full accuracy, despite being much more disk intensive. Since the incremental iteration method was the fastest method to smooth an image to the scale $t_2 = 20$, and still retain full accuracy, it was the smoothing method of choice.

The Hough Transform was also parallelised. This was a much simpler task than parallelising the AMSS, since the output of one iteration was not used as the input to the next. Rather, the θ values at which the transform was to be calculated could be split into P smaller equal size groups, each of which could then be calculated on separate processors. When parallelised, the standard method was over 15 times faster for $P = 16$ processors, so the speed increase was very close to P times. Due to the design of the algorithm, the modified Hough Transform was approximately three times faster when $P = 16$ processors were utilised.

Although not the primary focus of this thesis, these parallelisation methods proved to be very useful when testing required that results be produced in a timely manner. They also allowed the total fracture detection time to be greatly reduced, helping to meet the second goal outlined in Section 3.3.4 on page 51.