

Using SISAL to implement the Abingdon Cross Image Processing Benchmark

A. McKay
D. Abramson

Division of Information Technology
C.S.I.R.O.
723 Swanston St,
Carlton, 3053

1. Introduction

Programming parallel computers has been the topic of much research in recent years, and many research and commercial systems have been produced to ease the task. An example of such research is the development of the SISAL language, an applicative language based on VAL [1]. SISAL has been designed by a consortium of industrial and research organisations [2] for the specification and execution of parallel programs. It differs from conventional imperative languages by

- providing a safe execution model free of synchronisation constraints
- using a single assignment rule
- not allowing global variables
- providing a powerful parallel loop construct

SISAL can be compiled for a number of different machines and architectures. Code can be generated for shared memory machines like the Sequent Balance [3], Encore Multimax [3] and Cray X-MP; message passing machines like the Inmos Transputer [4], and dataflow machines like the Manchester machine [5] and the RMIT/CSIRO dataflow machine [6]. The most mature code generator is the shared memory version, developed by Colorado State University (CSU) in collaboration with Lawrence Livermore National Laboratory [3]. SISAL was designed primarily for execution of large numeric programs, and has been applied to A Gauss-Jordan linear equation solver, a particle in cell simulation, a protein simulation program [7], the Lawrence Livermore Loops [8], SIMPLE, a hydrodynamics code [9] and a one level barotropic weather simulation [10]. A recent report compares these applications [11]. A separate study examined the applicability of SISAL to non-numeric programs such as software tools. A SISAL compiler was written in SISAL, and the expressive power and efficiency of the language was evaluated [12].

In this paper we examine the use of SISAL for coding an image processing benchmark. Image processing software typically has a small number of simple operators applied to very large amounts of data, which differs from conventional numeric and symbolic programs. Parallel computing is essential for processing images quickly and cheaply. One of the important outcomes of the research is to determine whether the applicative style of programming has a detrimental affect on the management of large data structures used to hold images. For example, the image must be processed by a number of operators before it is completed. In an applicative program this may involve the creation of many large intermediate results which need to be collected and re-used.

To determine the effectiveness of the SISAL implementation, a parallel program was also written in standard C, with the parallel operators based on the Argonne macros [13]. The Argonne macros are a low level set of portable programming tools which can be applied to conventional C programs. The SISAL times are compared to the parallel C times on the Encore Multimax.

The paper begins with a discussion of the Abingdon Cross benchmark, and the operators required to implement it. It proceeds to describe the parallelism inherent in those operators, and then discusses the C and SISAL implementations. The comparative results are presented.

2. The Abingdon Cross Benchmark

The Abingdon Cross benchmark was developed from discussions which took place at the Abingdon Workshop in 1982 [14,15]. One of the tasks addressed at the workshop was to define a benchmark to be used to compare the performance of computers used in digital image processing. The result was the Abingdon Cross benchmark, which could be executed on most image processing machines, included linear and nonlinear operations, and produced the quality factor and the price performance factor which can be used for later comparison.

The benchmark involves the extraction of a skeletonised line cross from a solid cross imbedded in white gaussian noise, illustrated in the appendix. To extract the line cross first, the solid cross has to be extracted from the white gaussian noise, this is a linear operation. Second, the solid cross has to be thinned to the line cross, this is the nonlinear operation. The result is illustrated in the appendix. The time taken to complete these operations is then used to compare computers performance.

The standard Abingdon Cross image is defined as an image of size 512 pixels by 512 pixels with a depth of eight bits. The size can be varied to suit specific machine implementations. For the standard cross the vertical stroke occupies columns 224 to 288 and rows 64 to 488, and the horizontal stroke occupies columns 64 to 488 and rows 224 to 288. The back ground amplitude is 128, the cross has an amplitude of 160, except where the strokes cross where the amplitude is 192. This solid cross is then imbedded in white gaussian noise. Noise having a zero mean and standard deviation of 32 is added to all points.

For some computers it is impossible or very inefficient to process the full sized (512 by 512 pixels) Abingdon Cross. For example mapping the full sized cross to an array processor with a 96 by 96 processing elements is non-trivial, the likely result will be a slow execution time and hence an unfair comparison. The solution is to scale the cross to fit the array processor. This scaling has to be taken into account when calculating the quality factor and the price performance factor. Calculations are in Figure 1:

Quality Factor =	$\frac{N}{T}$	T = Execution time of the Abingdon Cross benchmark.
Price Performance Factor =	$\frac{N^2}{T \times C}$	N = The size of the image processed by the benchmark. C = The cost of the computer on which the bench mark was executed.

Figure 1, calculations for the quality factor and the price performance factor.

The original published results span some 26 image processing computers. In each case the benchmark was hand coded for the particular machine. The benchmark serves as a good test bed for the application of SISAL to large data problems, and was thus used in the study reported in this paper.

3. Image Processing Operators in the Cross

There is no defined set of image processing operators to process the Abingdon Cross benchmark; different methods of processing images suit different computer architectures. Part of the benchmark is the selection of the image processing operations to process the Abingdon Cross. There are many suggested ways to run the benchmark. It is important to select a method to process the benchmark that best suits the computer on which the benchmark is to be executed. When choosing the image processing operations it was known that neighbourhood operations were well suited for the computer on which the benchmark was to be processed.

There are two main steps in the Abingdon Cross benchmark. First, the removal of noise from the cross to extract the solid cross. Second, the skeletonisation of the solid cross to get the line cross required by the benchmark. The first step allows the most flexibility in the choice of image processing operations. There are many different algorithms which can be used in different combinations to produce the desired result. In general the gaussian noise has to be removed and some type of thresholding operation applied, not necessarily in this order. The second step is more straight forward. To skeletonise the cross using a connectivity preserving thinning algorithm. This only requires a choice from the available thinning algorithms.

3.1 Neighbourhood Operators

In general most digital images can be represented by a matrix DI, where each pixel DI(x,y) can take a value between 0 and 255, where 0 represents black and 255 represents white and values in between represent the grey scales between black and white. This definition gives an eight bit pixel plane on which to work. The matrix is usually square with the size being a power of two ranging from 64 to 1024.

Neighbourhood operators are applied directly to this matrix as shown in Figure 2. A new pixel value is calculated from its corresponding old value and the value of its neighbouring pixels in the matrix. A neighbourhood of pixels is selected, usually a rectangle or a square for simpler processing, and a new value for the pixel at the center of the neighbourhood can be calculated by applying a set of rules or a mathematical operation to the pixels within the neighbourhood.

Care must be taken in applying neighbourhood operators to the edges of the image matrix, in these cases the neighbourhood is not complete and either dummy data must be included in the calculation or the edge of the image matrix clipped.

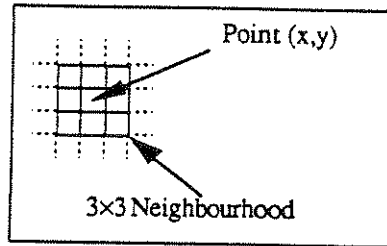


Figure 2, operation of a neighbourhood operator on the image matrix DI

The neighbourhood operators used to execute the abingdon cross benchmark are squares of odd height and width and the edges of the images are clipped.

3.2 Statistical Operations

Statistical operators are well suited for the filtering of noise from an image. They remove sharp changes in intensity which smooths out noise from an image. For a statistical operation a neighbourhood size is chosen and the statistical calculation is performed on the values within the that neighbourhood. The result of the calculation is then used to alter or set the intensity of the pixel or pixels within the neighbourhood.

The two statistical operators used in the implementation of the Abingdon Cross benchmark are median and average, shown in Figure 3. These operators are used to remove gaussian noise by reducing large variations in intensity, this essentially smooths the noise out of the image.

$$NDI(x,y) = \frac{1}{N} \sum_{(m,n) \in S} DI(n,m) \quad NDI(x,y) = X \quad X = \bigcup_{(m,n) \in S} DI(n,m)$$

Where X is the median of X

Figure 3, operation of average and median operators

The average operation is a simple operation. The basic neighbourhood average operation sets the center pixel to the average of the neighbourhood. This operation removes the noise from the Abingdon Cross very effectively, but blurs the image of the cross considerably. This is not important as the actual cross is large relative to the entire image.

The median operation is also a simple operation in principal, but finding an efficient method of calculating a median may be difficult. This operator also can be used to remove noise but keeps edges more intact. The median operator can also be used to do closings, the removal of holes in a solid image. This is a crude method to do a closing, and although it will not remove large holes in a solid image it will effectively and efficiently remove small holes.

3.3 Binary Threshold Operation

The threshold operator, as defined in Figure 4, is used to separate the grey scale of an image. The binary threshold takes an image which contains many grey levels and returns an image with only two grey scales. In this case the two values needed are white and black.

To threshold an image each pixel is examined individually and independently. If the pixel has a value less than the threshold value it is set to black, if the value is above or equal to the threshold value it is set to white.

The binary threshold operator, as defined in Figure 4, is used to emphasize the detail in an image.

$$NDI(x,y) = \begin{cases} 255 & \text{if } DI(x,y) < \text{Threshold,} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 4, binary threshold operator

3.4 Skeletonisation or Thinning Operation

There is no specific definition of a thinned image. The general aim of most thinning algorithms is to get the medial line or center line of the objects in an image, as shown in Figure 5.

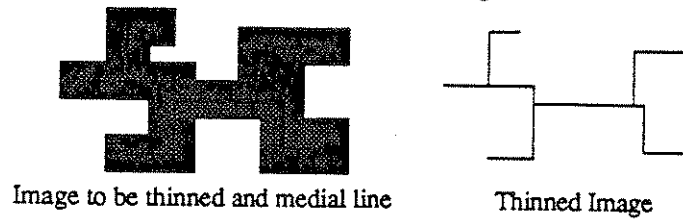


Figure 5, an example of a thinned image.

The medial line is difficult to calculate on even simple examples if an exact calculation is attempted. The next best solution is to use an algorithm that performs an iterative erosion on the solid objects in the image. An iterative erosion assumes that if an object is evenly eroded then the medial line will be left. To erode an object the pixels on the outside of an image are removed leaving a thinner image and revealing further pixels to be eroded. The erosion is stopped when only single pixel wide lines are left.

The algorithm used is put forward by Zang and Suen [17]. An image used by this thinning algorithm must only contain one foreground and one background colour. The image matrix can only contain zero and one, to represent the background and foreground respectively. The algorithm loops between two subiterations. Each subiteration examines every foreground pixel in the image and changes it to a background pixel if it is deletable. After each subiteration a new image is produced which is closer to the totally thinned image. The image is fully thinned when in either of the two subiterations no pixels are deleted from the image.

To determine if a pixel is deletable in a subiteration a set of conditions relating to its direct neighbours must be met. The pixel under consideration for deletion, p1, and its direct neighbours, p2 thru p9, is shown below.

p9	p2	p3
p8	p1	p4
p7	p6	p5

The first two conditions must be met in both subiterations.

- (1) The number of foreground neighbouring pixels must be between 2 and 6 inclusive.
- (2) The number of 01 patterns encountered in a clockwise examination of the neighbouring pixels is one.

The first subiteration must also meet the following two conditions.

- (1) One of the following pixels are zero p2, p4, p6.
- (2) One of the following pixels are zero p4, p6, p8.

The second subiteration must also meet the following two conditions.

- (1) One of the following pixels are zero p2, p4, p6.
- (2) One of the following pixels are zero p4, p6, p8.

4. A Serial and Parallel C Implementation of the Image Processing Operators

Neighbourhood image processing operators lend themselves well to parallelisation on a shared memory machine, each operation is made up of many small and independent operations. Neighbourhood operators apply a set of rules or a mathematical operation to every pixel in the image. The operations applied to each pixel are independent of each other. It is therefore feasible for every operation to be performed in parallel. The Encore Multimax is configurable up to 20 processors, it is therefore desirable to equally divide the work between the available processors.

The C implementation of these image processing operators uses dynamically partitioned loops, in which sections of the image are manipulated by different processors. In practice this involves forking a number of processes at the beginning of the program. Each process selects a row of pixels from the image on which to perform the operation and once finished selects another row, such that each process refers to an independent subsection of the image. Barriers are placed between different operators or sections of an operator to guarantee no overlap of dependant computations occurs. Because the underlying machine is shared memory, there is no need to pass sections of the image between processors.

The skeleton C code for the serial implementation of the average neighbourhood operator is shown below. Note image pixel accessing is done with pointers in C to allow varying image sizes. All images start at 0,0

```

mts = neighbourhoodsize / 2;
pixelsinneighbourhood = neighbourhoodsize * neighbourhoodsize;
for (row = mts; row < height - mts; row++) {
    for (column = mts; column < height - mts; column++) {
        sumation = 0;
        for (nrow = row - mts; nrow <= row + mts; nrow++) {
            for (ncolumn = column - mts; ncolumn <= column + mts; ncolumn++) {
                sumation += *(oldimage + ncolumn + (nrow * width));
            }
        }
        *(newimage + column + (row * width)) = sumation / pixelsinneighbourhood;
    }
}

```

Neighbourhood operators lend themselves well to parallelisation as there is a high degree of data parallelism. Each small neighbourhood is independent of each other, although the neighbourhoods overlap. Calculation of a new pixel value only requires the old pixel values within its neighbourhood and is independent of other new pixel value calculations around it.

The skeleton C code for the parallel implementation of the average neighbourhood operator is shown below.

```

mts = neighbourhoodsize / 2;
pixelsinneighbourhood = neighbourhoodsize * neighbourhoodsize;
createlock(sharedrowlock, unlocked);
sharedrow = middletoside;
fork(numofprocesses);
done = FALSE;
do {
    lock(sharedrowlock);
    if (sharedrow < height - mts) {
        done = TRUE;
        unlock(sharedrowlock);
    }
    else {
        row = sharedrow++;
        unlock(sharedrowlock);
        for (column = mts; column < height - mts; column++) {
            sumation = 0;
            for (nrow = row - mts; nrow <= row + mts; nrow++) {
                for (ncolumn = column - mts; ncolumn <= column + mts; ncolumn++) {
                    sumation += *(oldimage + ncolumn + (nrow * width));
                }
            }
            *(newimage + column + (row * width)) = sumation / pixelsinneighbourhood;
        }
    }
} while (!done);

```

Although each neighbourhood is independent it is not efficient to split up the image so that each image is processed individually as the amount of synchronisation would slow up the operation.

5. A SISAL Implementation of the Image Processing Operators

SISAL is an applicative language. There are no global variables, although local variables can be shared amongst functions and all functions and iterative constructs return values which can be assigned to variables or passed into other functions and iterative constructs. The SISAL language has the standard iterative constructs which occur in many regular languages, for example FOR WHILE and REPEAT, although they are arranged to cater for arrays and streams of data. The differences between applicative and imperative languages like C mean that the programmer has a different paradigm. The aim of this study is to determine the effectiveness of SISAL in the image processing arena, which is characterised by simple operators on large data sets.

SISAL has no specific parallel instructions or synchronisation. This to hide machine specific parallel operations and the partitioning of an algorithm. The extraction of parallelism from the coded program is ideally handled by the compiler and requires no human interaction, in actuality the way an algorithm is coded in SISAL can result

in large variations in the amount of parallelism extracted by the compiler. In contrast, the C implementation requires explicit code for implementing the parallel constructs.

Because data structures in SISAL cannot simply be updated in the normal way as in imperative languages, each image processing loop returns a new copy of the data after processing. This template matches the scheme used in the C code, in which new images are created as a result of a loop. The allocation of loop slices to processors is handled by the SISAL run-time system, and is thus invisible to the programmer. The skeleton SISAL code for the serial and parallel implementation of the average neighbourhood operator is shown below. Note SISAL will cater for varying sized images. Images start at 1,1 but this is not necessary.

```

let
  mts = neighbourhoodsize / 2;
  pixelsinneighbourhood = neighbourhoodsize * neighbourhoodsize;
in
  for row in 1, height cross column in 1, width
  returns array of
    if (row < mts + 1) | (row > height - mts) | (column < mts + 1) | (column > height - mts) then
      0
    else
      for nrow in row - mts, row + mts cross ncolumn in column - mts, column + mts
      returns value of sum oldimage[nrow][ncolumn]
      end for / areaarea
    end if
  end for
end let

```

6. Results and Comparison

To implement the Abingdon Cross benchmark the image processing operators were first written as a serial implementation in C. This was to check the benchmark was obtainable with the set of image processing operators available. The implementation serial in C was straight forward.

Converting the algorithms from serial C to parallel C was considerably more difficult than the original coding. The conversion, as shown in section 4, involved unravelling the outermost For loop to allow concurrent processing of rows.

Adding the synchronisation primitives and shared memory to the original code and then debugging the code was particularly difficult. The Argonne lab m4 macros are not handled by the standard C source level debuggers. The source level debugger does not show the original source code but the C code with the expanded macros embedded. This makes an already cumbersome task of debugging parallel code more difficult.

Three of the algorithms, average, threshold, and median were difficult to code, but once the first was completed the next two were straight forward. Implementation of the parallel thinning operation required the use of barriers to separate the iterations of the thinning algorithm and the gathering of the iteration termination variable. This required some intensive and time consuming parallel debugging.

Converting the algorithms from serial C to Sisal was fairly straight forward once a basic understanding of Sisal was attained. Sisal is a strongly typed language, many of the type errors and syntactic misinterpretations that occur in C and have to be removed with debugging appear at the Sisal syntax and type checking stages. There are few options to removing run-time errors from Sisal. Sisal can not at the moment be debugged at the source level. The only options are to debug the code at the two intermediate stages between the Sisal source and the executable code, these are IF1 and C. Seeing the relationship between the IF1 and C, and the Sisal source is difficult but possible. A high level debugger for Sisal would solve this problem.

	Serial C	Parallel C	Sisal
Average	49 (1420)	87 (2330)	38 (929)
Threshold	56 (1396)	137 (3020)	39 (711)
Median	80 (2096)	121 (2990)	114 (2761)
Thinning	173 (3969)	225 (6268)	161 (4137)
Include/Control	172 (3763)	193 (4182)	113 (3034)

Table 1, Source code size used to implement the Abingdon Cross Benchmark for C and Sisal in lines and (bytes). Note: no comments were included in these sizes.

Table 1 shows the size of the source code for the serial C and the Sisal are similar. The Sisal is in general smaller in size even though its key words are longer than C's. Sisal's operators are functionally more powerful than C's but also more complex. In general it takes the same amount of time to write an algorithm in C source and Sisal source. There are some cases, for example the median operator, where this is not true. The algorithm used to get the median is the standard divide and conquer iterative algorithm, which is difficult to convert to Sisal, and results in lengthy, convoluted code and slow execution.

The parallel C code is considerably larger than the Sisal code, which is functionally identical whether run sequentially or in parallel. A large component of the parallel C code is related to control overheads.

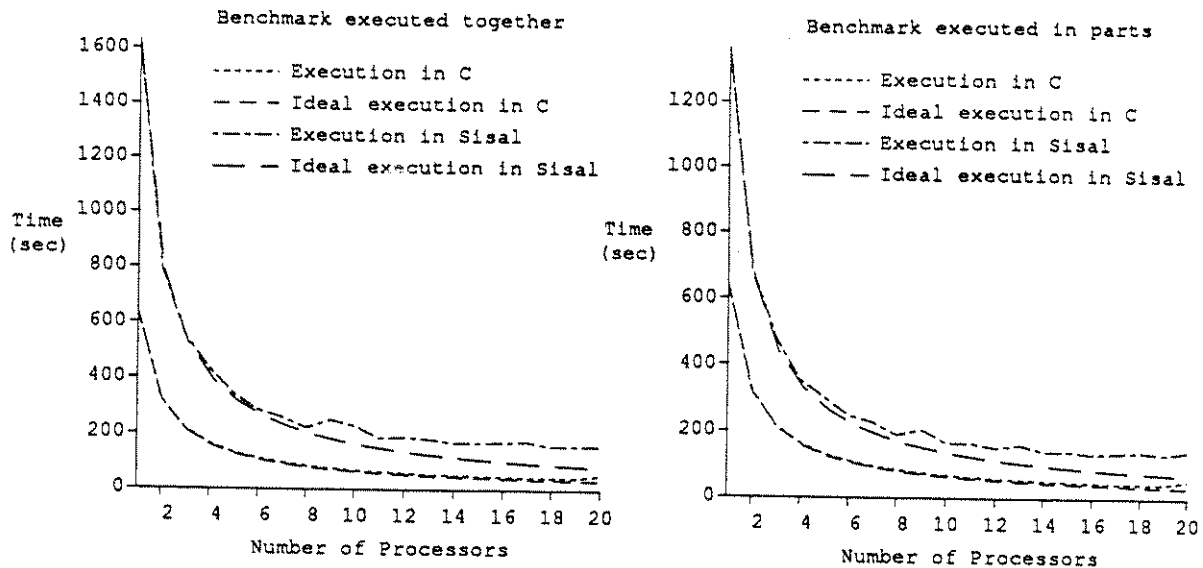


Figure 6, speed up curves for C and Sisal implementations of the Abingdon Cross Benchmark
Note: executed in parts means each image processing operation was executed as a separate program, also all times do not include I/O.

The speed up curves for the parallel C code and the Sisal are shown in Figure 6. The Sisal code is much slower than the Parallel C code. Although the timings for the serial C are not shown it is important to note in all cases the timings for the serial C are very close to the timings for the parallel C executing with one processor. This shows almost no system or synchronisation overhead. The above speedup graphs for Parallel C which are very close to ideal.

The speed up for Sisal is close to ideal until about 8 processors, from then on only minimal speed up is obtained. This is because the Sisal tasking becomes a significant overhead around 8 processors. The time for execution of Abingdon Cross Benchmark as a single program and as four separate programs are similar in time and speed up performance. The overhead of using large data structures has not significantly effected the performance of Sisal.

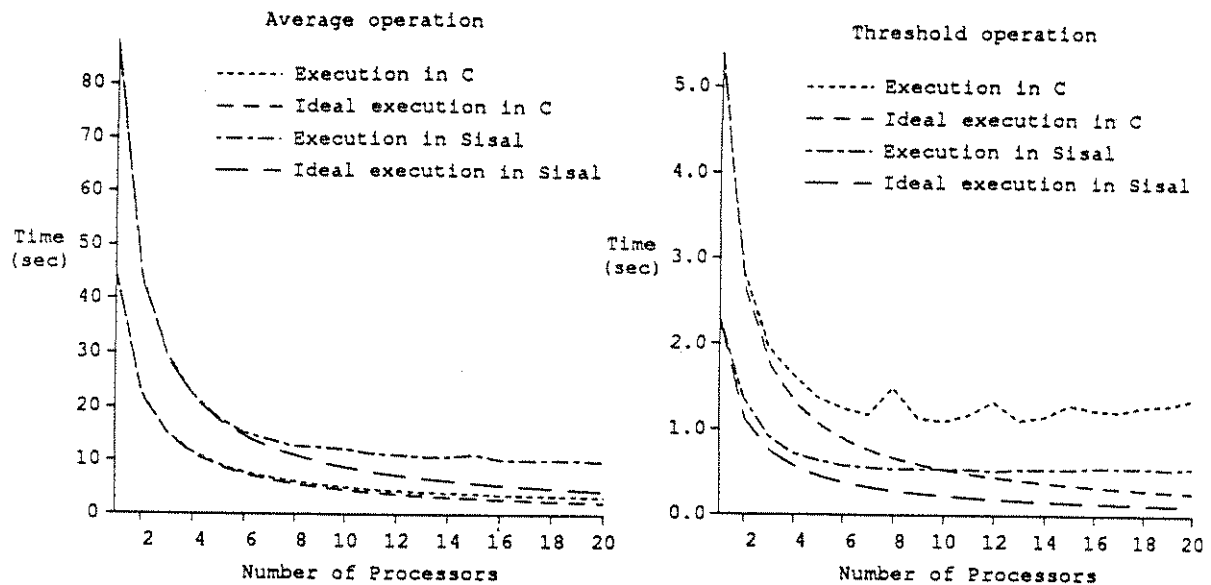


Figure 7, speed up curves for C and Sisal implementations of the average and threshold operators
Note: all times do not include I/O.

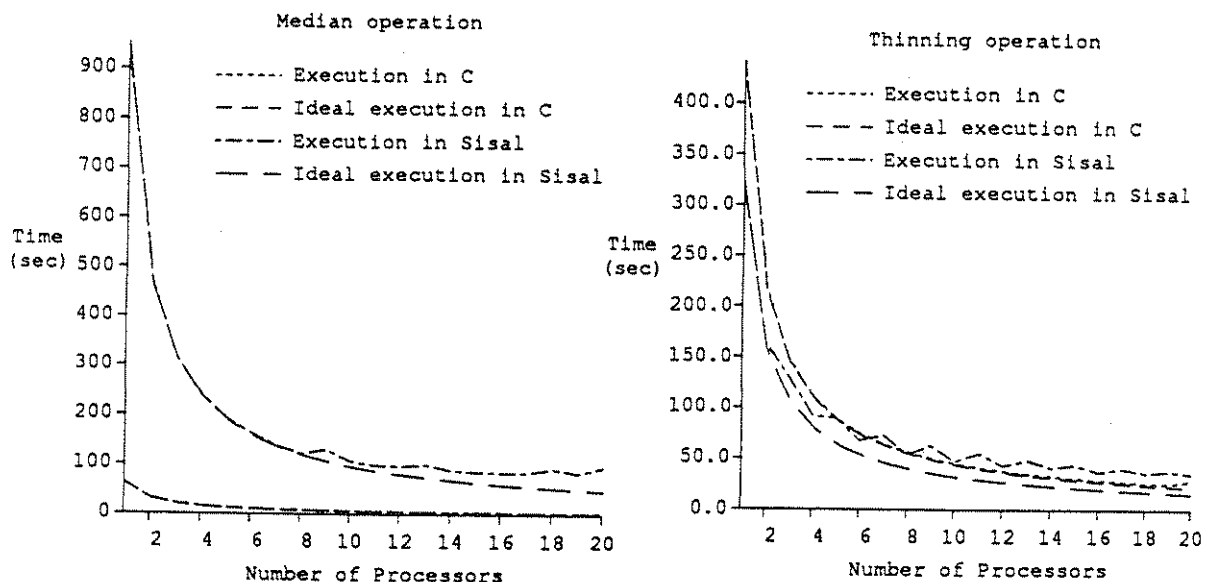


Figure 8, speed up curves for C and Sisal implementations of the median and thinning operators
Note: all times do not include I/O.

From the graphs of the performance of the individual image processing operators in the Abingdon Cross Benchmark it can be seen that the major difference between the time of the parallel C and the Sisal is contained within one operator, the median operator. The problems encountered with coding the median function have resulted in a very poor execution time. The algorithm is not well suited to Sisal, a new algorithm would dramatically improve this performance.

The thinning algorithm shows a better one processor speed, but at around 8 processors speed up degrades making it slower than parallel C. This is significant because to write a parallel C program to match the performance of the Sisal for 1 to eight processors would be very time consuming.

7. Conclusions

This paper has extended the range and type of benchmarks which have been applied to the SISAL language. It has demonstrated that the Abingdon Cross image processing operators can be efficiently coded in an applicative programming language without significant loss of performance. This result is important because this class of

program has simple algorithmic structure but operates on very large data structures, which to date, have been inefficiently managed in applicative languages.

One of the operators, median, demonstrated lower absolute performance than the others. Work is continuing on finding a more efficient form of this operator in SISAL. The other operators shows comparable performance to the C results.

The SISAL speedup curves were close to ideal until 8 processors. Ongoing work will be to identify the cause of this performance loss.

ACKNOWLEDGEMENTS

The High Performance Computation Project is a joint project between the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) Division of Information Technology and the Royal Melbourne Institute of Technology (RMIT).

REFERENCES

- [1] J.R. McGraw. "Data-flow computing: the val language". ACM Transactions on Programming Languages and systems, 4(1), Jan. 1982.
- [2] J.R. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J.R.W Glauert, I. Dobes, and P. Hohensee. "SISAL-Streams and Iterations in a Single Assignment Language", Language Reference Manual, Version 1.2. Technical Report TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.
- [3] R. R. Oldehoeft and D. C. Cann. "Applicative parallelism on a shared-memory multiprocessor". IEEE Software, January 1988.
- [4] J.-L. Gaudiot and L.T. Lee. "Occamflow: a methodology for programming multiprocessor systems". Journal of Parallel and Distributed Computing, In Press 1989.
- [5] J.R. Gurd, C.C. Kirkham, and I. Watson. "The Manchester data-flow prototype". Communications of the ACM, 28(1), January 1985.
- [6] D. Abramson and G.K. Egan, "An Overview of the CSIRO/RMIT Parallel Systems Architecture Project", Australian Computer Journal, Vol 20, No 3, August 1988.
- [7] J.Cann, E.York, J. Stewart, J. vera and R.Maccioni. "Small zone gel chromatography of interacting systems: Theoretical and experimental evaluation of elution profiles for kinetically controlled macromolecule-ligand reactions", Analytical Biochemistry, (175), December 1988.
- [8] F.H.McMahon "The Livermore Fortran kernels: A computer test of the numerical performance range", Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.
- [9] W.P.Crowley, C.P. henderson and T.E. Rudy, "The simple code", Technical Report UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, February 1978.
- [10] P. Chang and G. Egan, "An Implementation of a Barotropic Numerical Weather Prediction Model in the Functional Language SISAL", Proceedings of the SIGPLAN 1990 Symposium on Principles and Practice of Parallel Programming, March 1990.
- [11] D. Cann, "Vectorization of an Applicative Language: Current Results and Future Directions", Technical Report UCRL JC 105654, Lawrence Livermore National Laboratory, Livermore, CA, November 1990.
- [12] A. Wendelborn, H. Garsden, G. Irlam, I. McDonald and G. Smith, "The Development and Efficient Execution of SISAL Programs", Chapter 17, Advanced Topics in Data-flow Computing, Prentice Hall, Edited by Gaudiot and Bic, 1990.
- [13] R. Stevens et al, "Portable Programs for Parallel Processors", Holt Rinehart and Winston Inc, 1987.
- [14] K Preston, Jr., "Benchmark results -- The Abingdon Cross", Evaluation of Multicomputers for Image Processing, L. Uhr et al.,eds., Academic Press, Cambridge, Mass., 1986.
- [15] K Preston, Jr., "The Abingdon Cross Benchmark Survey", IEEE Computer, July 1989.
- [16] W. Niblack, "An Introduction to Digital Image Processing", Prentice Hall.
- [17] T.Y.Zhang and C.Y.Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns", Communications of the ACM, March 1984, Volume 27, Number 3.

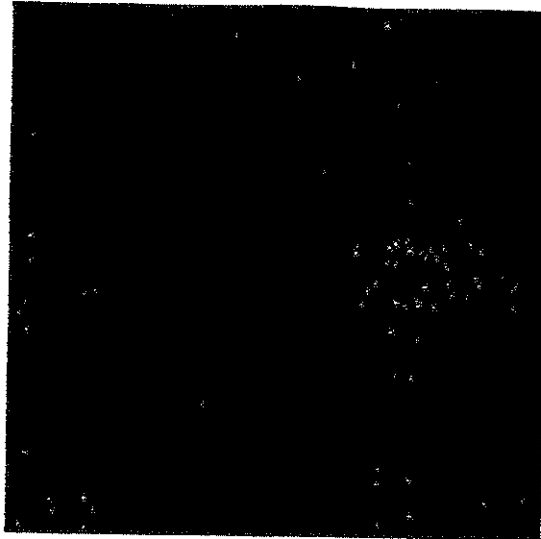


Fig 1, The Abingdon Cross before execution of the benchmark.

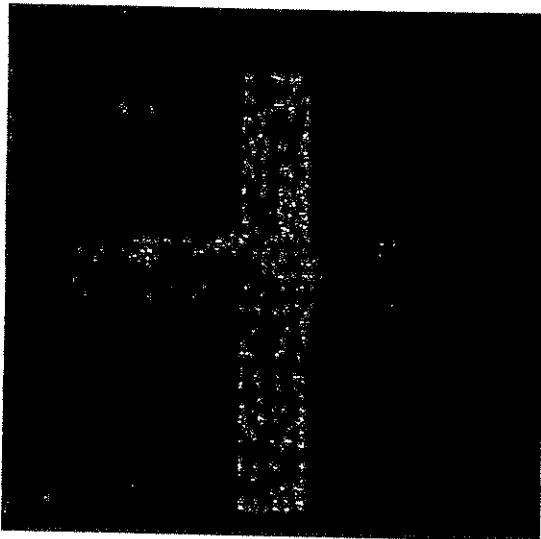


Fig 2, The Abingdon Cross after the Average operation.

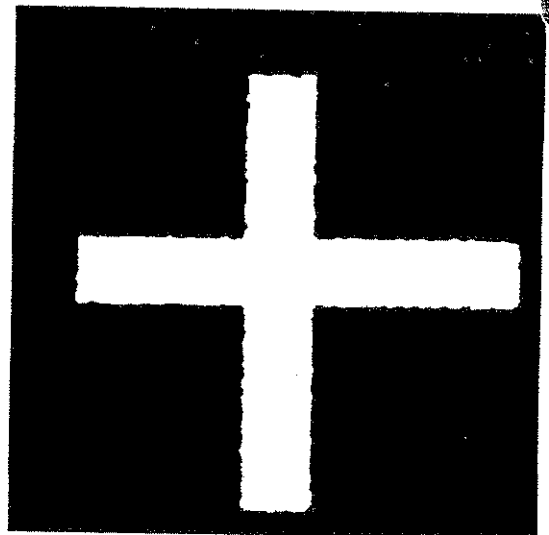


Fig 3, The Abingdon Cross after the Threshold operation.

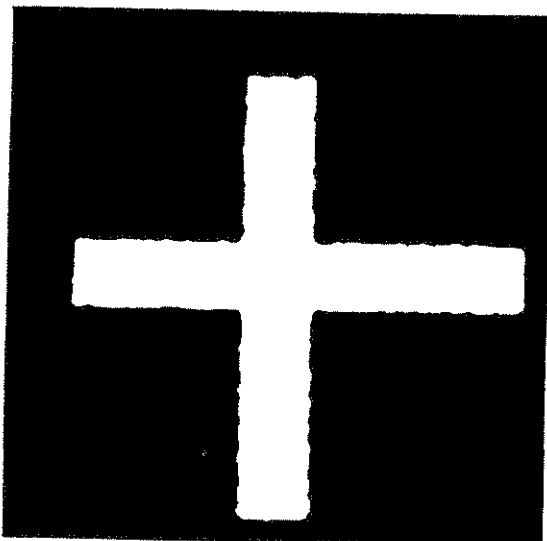


Fig 4, The Abingdon Cross after the Median operation.

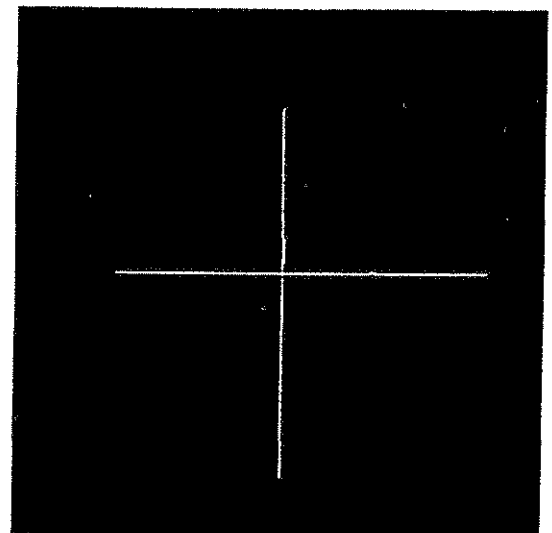
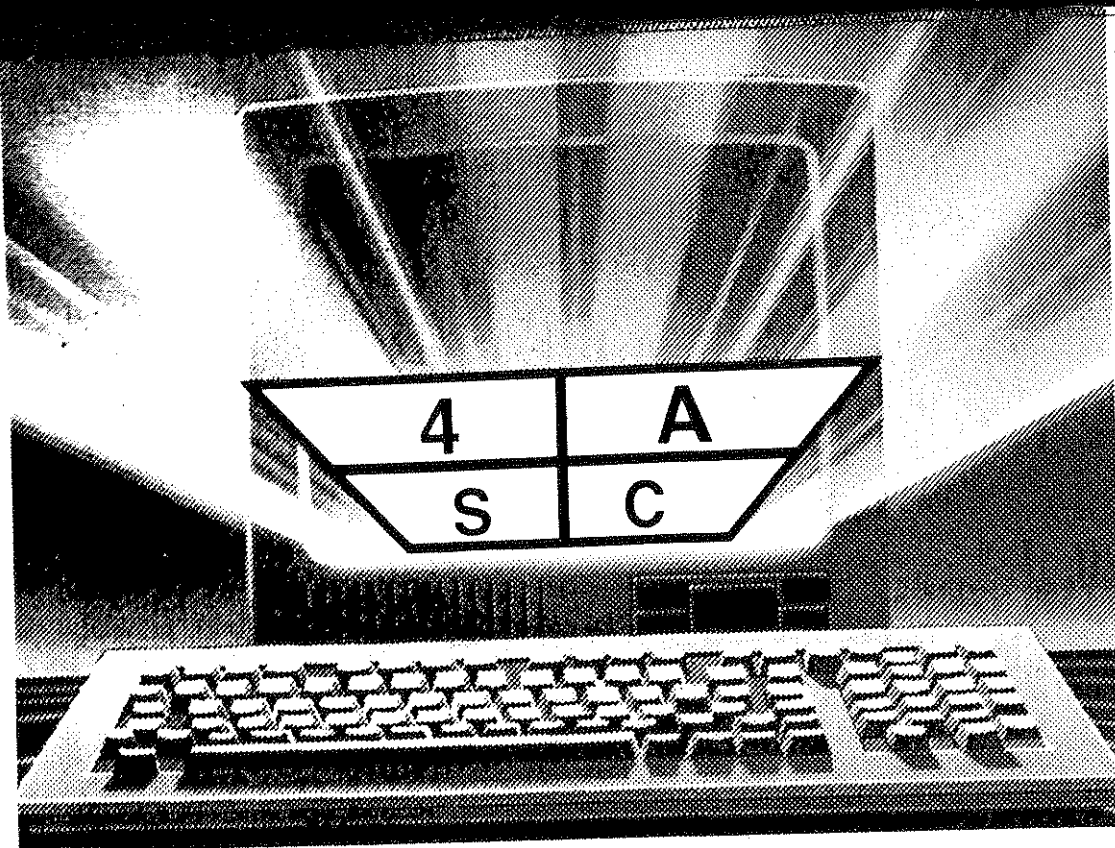


Fig 5, The Abingdon Cross after the Thinning operation.



FOURTH AUSTRALIAN SUPERCOMPUTER CONFERENCE

"VISUALISE THE POSSIBILITIES"

UNIVERSITY PARK HOTEL - BOND UNIVERSITY - GOLD COAST
QUEENSLAND 2 - 5 DECEMBER 1991

Held in Queensland
at the initiative of the Department of Business, Industry and Regional Development

HANDBOOK AND PROCEEDINGS



CONVEX

Hosted by

TECHQUAD

digital



Promoted by



Australian
Computer
Society



nes Corporation