

# Evaluating the Performance of a SISAL implementation of the Abingdon Cross Image Processing Benchmark

David Abramson  
School of Computing and Information Technology  
Griffith University  
Kessels Rd, Nathan, Qld 4111  
davida@cit.gu.edu.au

A. McKay  
Department of Computer Science  
Royal Melbourne Institute of Technology  
P.O. Box 2476V  
Melbourne 3001.

Contact Information: [davida@cit.gu.edu.au](mailto:davida@cit.gu.edu.au)

## Abstract

There are many paradigms being promoted and explored for programming parallel computers, including modified sequential languages, new imperative languages and applicative languages. SISAL is an applicative language which has been designed by a consortium of industrial and research organisations for the specification and execution of parallel programs. It allows programs to be written with little concern for the structure of the underlying machine, thus the programmer is free to explore different ways of expressing the parallelism. A major problem with applicative languages has been their poor efficiency at handling large data structures. To counter this problem SISAL includes some advanced memory management techniques for reducing the amount of data copying that occurs. In this paper we discuss the implementation of some image processing benchmarks in SISAL and C to evaluate the effectiveness of the memory management code. In general, the SISAL program was easier to code than the C (augmented with the PARMACS macros) because we were not concerned with the parallel implementation details. We found that the SISAL performance was in general comparable to C, and that it could be brought in line with an efficient parallel C implementation by some programmer specified code transformations.

## 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 Y-MP; message passing machines like the Inmos Transputer [4], NCube 2000 and Intel Paragon, 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 Lawrence Livermore National Laboratory in collaboration with Colorado State University (CSU)

[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, 12]. 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 [13].

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 like the ones cited in the previous paragraph. Parallel computing is essential for processing images quickly and cheaply. Providing an efficient and portable programming environment for image processing algorithms remains an important challenge for language systems. SISAL has been designed to provide a language which is independent of the underlying platform (both sequential and parallel), and thus is well suited to the requirements.

Traditionally, functional and data flow languages have suffered because they do not manage data structures efficiently. The *single assignment rule* means that, in principle, a data structure must be copied every time an assignment is performed. If a program manipulates large data structures repeatedly this can represent an enormous overhead above conventional imperative languages. The Colorado State University SISAL run time system incorporates a number of optimisations which improve the efficiency of the code when it manipulates both scalars and large data structures. These include conventional code optimisations like loop invariant motion, etc, which assist with scalar manipulation, plus a number of novel transforms which detect when data structure copying is not required. The latter transforms detect when an update can be performed in place without the need to copy the entire data structure.

Image processing algorithms are typically quite small, but manipulate a number of very large data structures. The Abingdon Cross benchmark, which is introduced in the next section, requires the application of multiple operators on one image. In an applicative program this may involve the creation of many large intermediate results which need to be collected and re-used. Thus, image processing operators serve as a good benchmark to measure the effectiveness of the data structure manipulation optimisations. The purpose of this paper is to determine how well the CSU optimisations perform when stressed with data intensive applications like image processing. This application provides a more intensive test of these optimisations than most of the other studies.

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 [14]. 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 [15, 16]. 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. The benchmark 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 solid 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.

The original published results span some 26 image processing computers. In each case the benchmark was hand coded for the particular machine. Whilst the benchmark was primarily intended as a method for evaluating hardware systems, it also allows the performance of compilers to be evaluated. 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.

We chose the Abingdon Cross benchmark for this study for a number of reasons. First, as stated in the introduction, the purpose of the research was to determine how well SISAL managed the parallel manipulation of large data structures. Previous work on using SISAL to implement large numeric codes has concentrated on quite complex algorithms which manipulate moderately large structures of floating point numbers. For example, as part of our evaluation of SISAL, Chang developed a weather simulation program which consisted of about 3000 lines of SISAL [10]. This code, whilst much smaller than a production model, represents most of the complex operations present in such models. It manipulates a number of moderately large data structures containing real numbers. These programs require good code and data structure optimisations in order to execute efficiently. On the other hand, the Abingdon Cross Benchmark consists of about 500 lines of SISAL and manipulates a number of very large data structures of integers. Given this ratio of source code to data size, plus the focus on integer rather than floating point arithmetic, any inefficiency in data structure manipulation is likely to be exaggerated in the Abingdon Cross code compared to the other programs. Thus, any inefficiency in the data structure manipulation algorithms is likely to be highlighted by the Abingdon Cross benchmark. Second, the algorithms were small enough to allow rapid coding in both SISAL and C. This allowed us to make a controlled comparison between the two implementations. Many of the other studies mentioned in the introduction were performed by taking an existing piece of code written by one author and then translating it into SISAL by a different programmer. This technique becomes difficult when the second programmer does not completely understand the original algorithm. It also does not account for different programming practices and levels of experience between programmers. In this study we used the same programmer to write both pieces of code, thus eliminating differences in programming style and understanding of the original algorithm. Third, the Abingdon Cross benchmark requires the application of a number of independent operators on one image. This requires both intensive in-situ operations on the image plus the ability to pass the structure efficiently from one operator to the next. Finally, whilst the code has many of the properties of an artificial benchmark (such as the Spec benchmark), it also represents a real world problem which must be solved using both parallel and sequential machines.

### **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 operators to process the Abingdon Cross.

There are many suggested ways to run the benchmark, and it is important to select an appropriate method which suits the computer on which it is to be executed. The machines being used in the study reported in this paper are all shared memory multiprocessors, thus issues of interprocessor communication are of little importance.

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 1. 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. Later in the paper we show that the inclusion of a boundary region complicates the SISAL implementation because all elements of the image must be re-built after an operation.

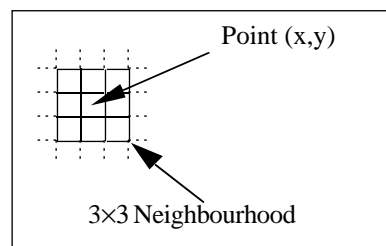


Figure 1 - Operation of a neighbourhood operator on the image matrix  $DI$

The neighbourhood operators used to execute the Abingdon Cross benchmark are squares of odd height and width.

### 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 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 2. These operators are used to remove gaussian noise by reducing large variations in intensity, this essentially smooths the noise out of the image.

$$AverageDI(x,y) = \frac{\sum DI(m,n)}{N^2}$$

$$MedianDI(x,y) = middle(sort(DI(m,n)))$$

where.

$(m,n) \in Neighbourhood\ of\ size\ N$

$sort()$  returns the list of elements sorted by value

$middle()$  returns the middle element of list

Figure 2 - 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 3, 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 3, is used to emphasize the detail in an image.

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

Figure 3 - 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 4.

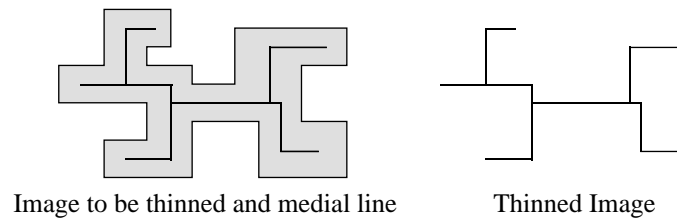


Figure 4 - An example of a thinned image.

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 the one devised by Zang and Suen [18]. 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 sub-iterations. Each sub-iteration examines every foreground pixel in the image and changes it to a background pixel if it is deletable. After each sub-iteration a new image is produced which is closer to the totally thinned image. The image is fully thinned when in either of the two sub-iterations no pixels are deleted from the image.

To determine if a pixel is deletable in a sub-iteration 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 sub-iterations.

- (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 sub-iteration 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 sub-iteration must also meet the following two conditions.

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

This section has given sufficient detail of the image processing primitives used by the benchmark. In the next section we will discuss the C implementations.

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

Neighbourhood operators apply a set of rules or mathematical operations which are independent of each other to every pixel in the image. It is therefore feasible for every operation to be performed in parallel. The Encore Multimax used in this study contains 20 processors and it is therefore desirable to divide the work equally 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 sub-section 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 = mts;
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);

```

The sequential code is parallelised by allowing multiple processors to work on rows of the image concurrently. There are two main techniques for allocating rows of the image to processors. The first uses a static division and allocates a fixed number of rows to each processor. The second method dynamically allocates rows from a central pool. The static case has the advantage that it is possible for each processor to operate independently of each other. Dynamic allocation has the advantage that it can account for variability in processing time between rows and help even the work load distribution. It has the disadvantage that each processor must access a shared pointer to receive the next row to manipulate. Most of the operators lend themselves to a static allocation because there is very little variability in row processing time. However, the thinning operator processes variable amounts of data in each row. Thus, the dynamic allocation strategy assists with maintaining an even workload. In our experiments, there was almost no difference in the efficiency of static and dynamic strategies for the basic operators, but a significant advantage in using dynamic for the thinning operator. There was also almost no difference in the complexity and size of the code. Thus, in order to simplify code maintenance we chose a dynamic strategy for all operators. Also, since SISAL uses a dynamic strategy, using dynamic for the C code made it easier to compare the performance of the C and SISAL implementations.

Each processor chooses a row from a centrally shared pool. Access to the pool is controlled by locks. The output of the code is deterministic, however, the workload distribution is not. Whilst the parallel code appears relatively simple, extra control and synchronisation code had to be added to the sequential C version. Care is required in the correct placement of lock and unlock primitives.

Coding of three of the algorithms, average, threshold, and median was relatively 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. Below is a skeleton for the parallel thinning algorithm. As described in section 3.4, the thinning operation consists of a sub iteration embedded in another major iteration. A barrier is used to separate the major iterations of the algorithm. The second sub iteration cannot occur until the

first one is complete, and since there are multiple processors involved, a barrier must be used to guarantee that all processors are at the same point. Also, the termination condition cannot be tested until all processes have lodged their termination value.

```

while (TRUE) {
    wait_barrier(subiteronebarr);          /* wait for the start of the next thinning iteration */
    workerthinningdone = TRUE;
    workersubiterone(oimage, nimage, &workerthinningdone);
    if (! workerthinningdone) {          /* Combine worker erosion results */
        wait_semaphore(thinningdonelock);
        allthinningdone = FALSE;
        signal_semaphore(thinningdonelock);
    }
    wait_barrier(checkthindonebarr);      /* wait for all processes to lodge their erosion result */
    if (allthinningdone) break;
    swapimages(oimage, nimage);
    workersubitertwo(oimage, nimage, &workerthinningdone);
    swapimages(oimage, nimage);
}

```

In the next section we examine the SISAL form of the operators.

## 5. A SISAL Implementation of the Image Processing Operators

SISAL is an *applicative* language. There are no variables in the conventional sense, only *names* which are bound to values. Names cannot be declared and accessed globally, although local names can be passed as parameters between functions. All functions and iterative constructs return values which can either be bound to names or passed into other functions and iterative constructs. SISAL has the standard iterative constructs which occur in many imperative languages, for example FOR, WHILE and REPEAT, although they are arranged to cater for arrays and streams of data.

All SISAL constructs are available to the programmer regardless of whether the underlying machine is sequential or parallel. This to hide machine specific parallel operations and the partitioning of an algorithm. The extraction of parallelism from the program is managed by the compiler and requires no human interaction. In contrast, the C implementation requires explicit code for implementing the parallel constructs. In practice we observed wide variations in the performance of parallel algorithms written in SISAL depending on the way the code was written. Later in the paper we demonstrate the effects of this re-coding. However, it is important to emphasize that the programmer experiments with different forms of *algorithms* in SISAL, not with different parallelisation *techniques*. In contrast the C program exposes a variety of different parallelisation *techniques* which are independent of the *algorithm*.

Because data structures in SISAL can not simply be updated, as they are in imperative languages, each image processing loop returns a new copy of the data after processing. It is therefore important to limit the number of new images that are produced because they each contain 256 KBytes of data. This loop template matches the scheme used in the C code, in which new images are created as a result of a loop. However, in the C code the pointer is passed out as the 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 as an arbitrary choice but this is not necessary. The same basic template as shown is applicable to most of the operators.

```

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 / pixelsinneighbourhood
    end if
  end for
end let

```

Earlier in the paper we commented that the boundary region required by the neighbourhood operators complicates the SISAL code. This is observed in the code above. The loop must produce a complete copy of the image in spite of the boundary elements. In this case the loop body is guarded by a conditional statement, which must be executed on every iteration. We experimented with different techniques for protecting the border regions (for example using catenate operators) and could not devise a more efficient technique. The problem arises because SISAL does not currently allow the replacement of a sub array in a larger array with one statement. Thus the code is somewhat more complicated than necessary.

## 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, i.e. the cross could be extracted from the noise. 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 National Laboratory 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. This property is true of parallel extensions to sequential languages that make use of macro pre-processors.

Converting the algorithms from serial C to SISAL was fairly straight forward in spite of the difference in programming paradigms between applicative and imperative languages. Because SISAL is a strongly typed language, many of the type errors and syntactic misinterpretations that occur in C, and result in programming bugs, are trapped in the compilation phase. Whilst this property is important for sequential software, it is even more important in parallel programs given the non-deterministic errors that can occur through program bugs. SISAL guarantees deterministic behaviour of the code, and thus this class of error is not present in SISAL programs.

Operator	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)
Total	530 (12644)	763 (18790)	465 (11572)

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 for the algorithms implemented we found that 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 a standard divide and conquer iterative algorithm, which is difficult to convert to SISAL because of the lack of memory update operations. Later in the paper we show a few different forms of the median operator.

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.

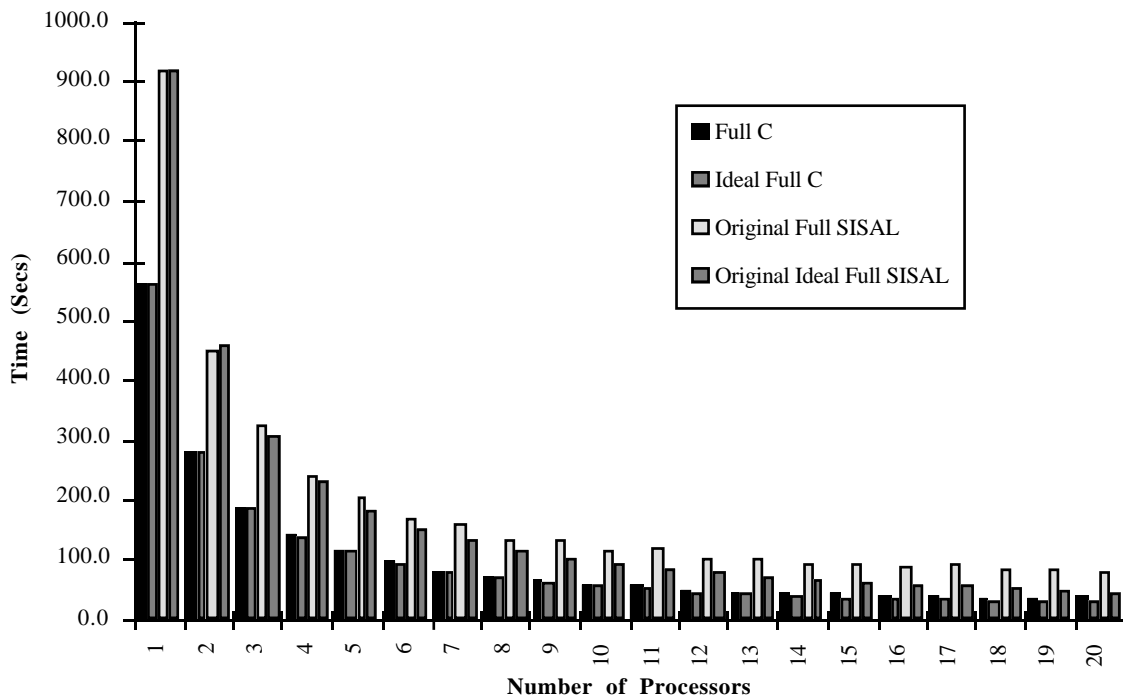


Figure 5a - Execution times for Full version of SISAL and C codes

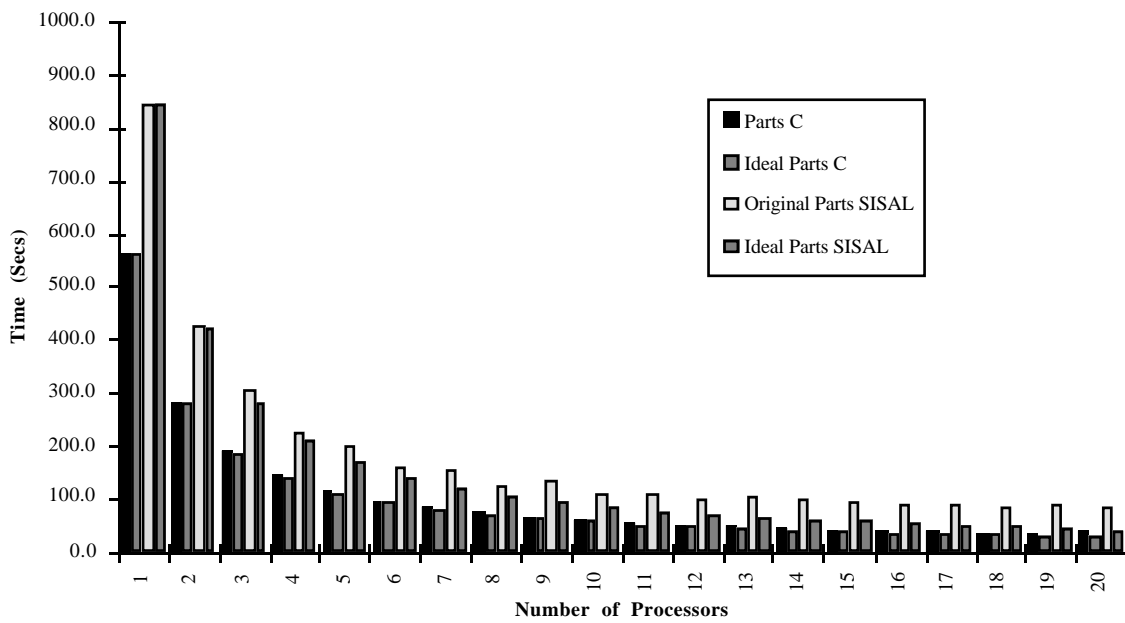


Figure 5b - Execution times for Parts version of SISAL and C codes

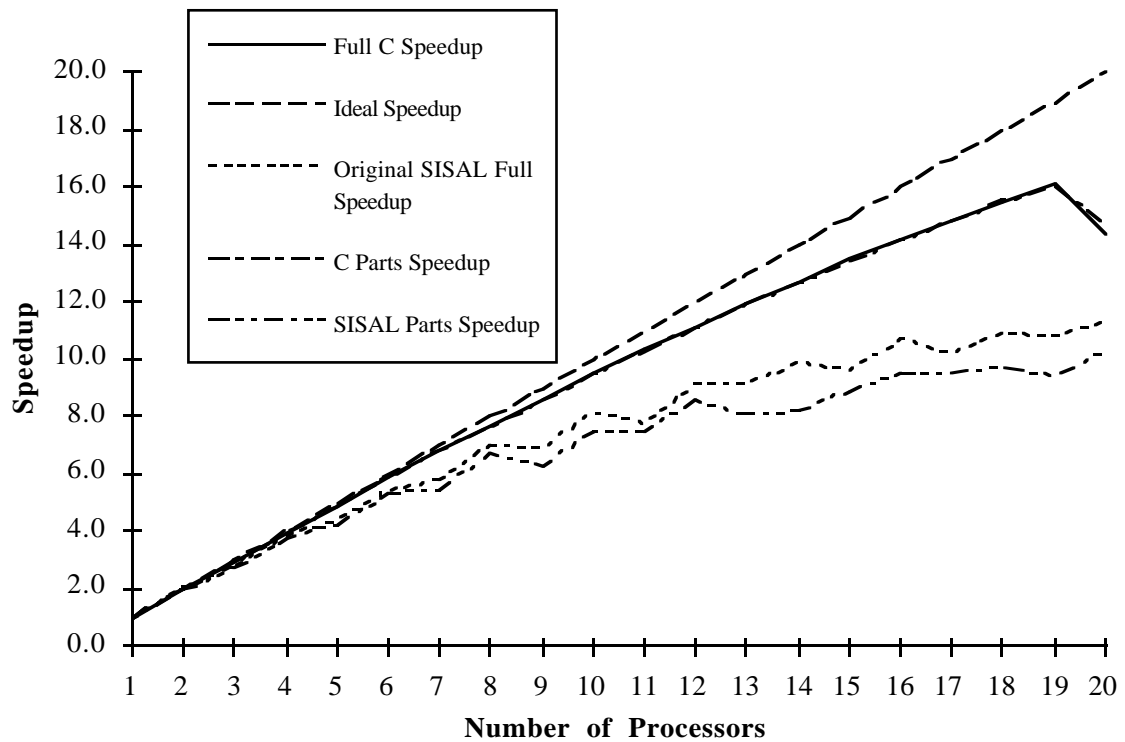


Figure 5c -Speed up curves for C and SISAL implementations of the Abingdon Cross Benchmark

Note: executed in parts means each image processing operation was executed as a separate program, also all times do not include I/O.

The execution times and speed up curves for the parallel C code and the SISAL are shown in Figure 5 (a, b & c). The *ideal* times and speedup curves are computed based on 100% efficiency in the parallelisation. It can be seen that 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 are very close to ideal, except for an anomaly which occurs at 20 processors. The two full and parts C curves are superimposed.

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 system becomes a significant overhead around 8 processors. It should be noted that the uni-processor SISAL code was identical to the multiprocessor code, unlike the C program.

The benchmark was executed in two forms. In the first, the image is processed consecutively by the four operators by passing the data structures between function calls. In the second form, the operators are applied individually and only the time spent in the operators is recorded. These are compared in Figures 5a and 5b. It can be seen that the first form is slower in SISAL than the second. In spite of the lack of warnings from the compiler indicating that copying was necessary, it is our belief that the time difference was caused by rebuilding the large image structures after each routine call. More recent versions of the compiler use more sophisticated algorithms for removing copying, and may avoid this rebuilding operation altogether. The C code simply passes a pointer between functions, and thus the cost is the same in both forms.

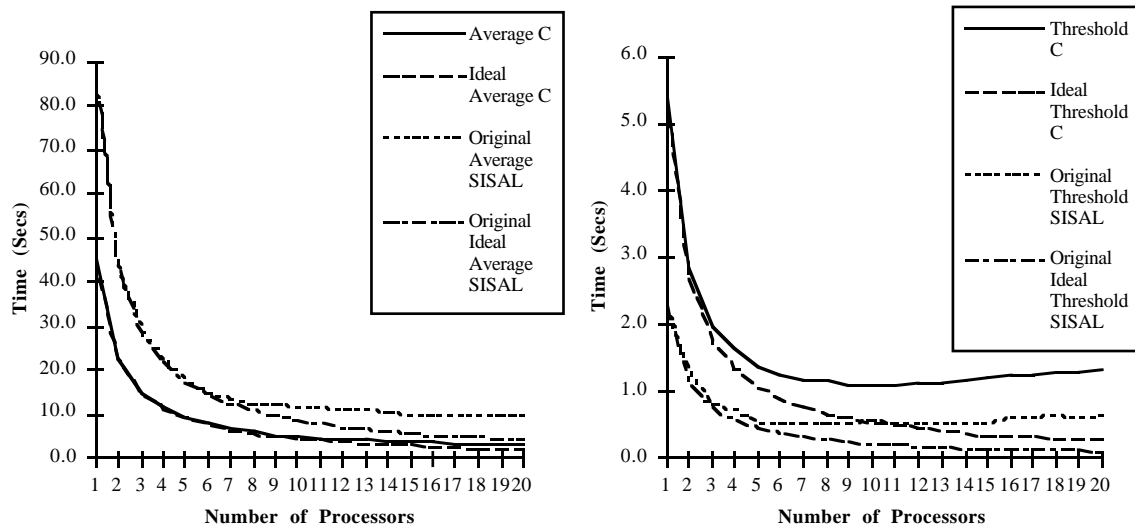


Figure 6, speed up curves for C and SISAL implementations of the average and threshold operators

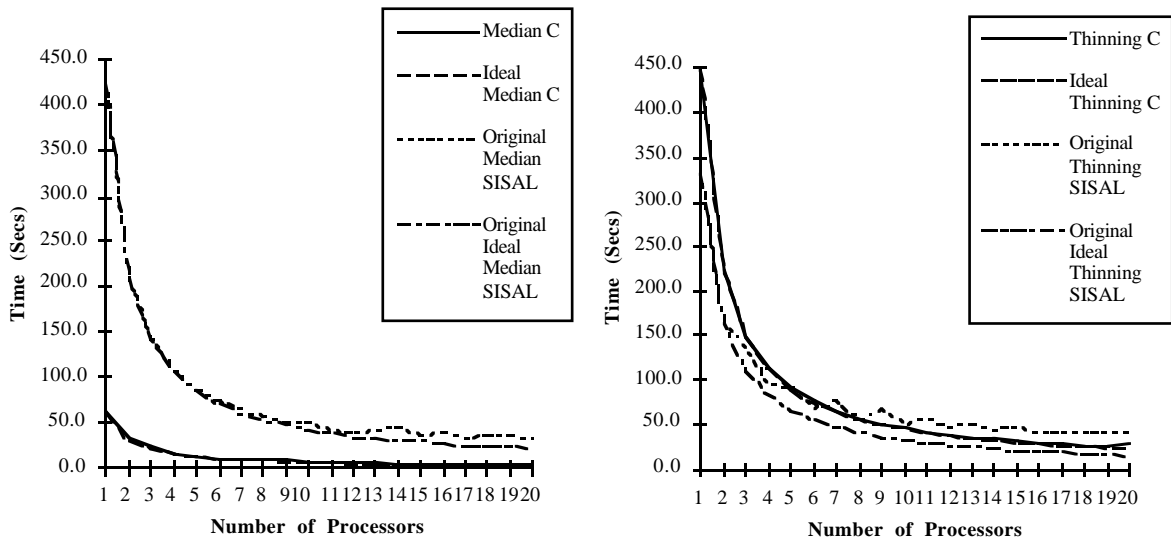


Figure 7, speed up curves for C and SISAL code of the median and thinning operators

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, and in the next section we present a new algorithm which dramatically improves this performance. The average operation is also slower, but because they do not constitute a major fraction of the total execution time, they do not show up as significantly.

The SISAL thinning algorithm shows a better one processor speed than the C code, but at around 8 processors speed up degrades making it slower than parallel C. In the next section we consider a number of optimisations that improve the performance of the SISAL code.

## 7. Optimisations of SISAL Code

Whilst the results contained in the previous section are promising, there is still room for improvement in the performance of the SISAL code. In this section we examine a number of optimisations that were performed on the code.

### 7.1 Optimisation of the Median Operator

Figure 8 shows the pre and post optimised versions of the median operator. The median operator extracts the median value from a set stored in the array *data*. The original code was converted directly from the C code to SISAL code with out consideration to the SISAL code optimisers. To swap two elements in an array, every element in the array is examined, and copied to the new array in the same position except if the array element is the *leftbound* or *rightbound*. If array element is *leftbound* then it is copied to the *rightbound* position and vice versa. As there is no temporary variable, as required in an single array two element swap, the array can not be updated in place, which results in the new array being copied. The second version makes use of the SISAL update operator, which is optimised by the compiler to perform an update in place. This optimisation results in the time of the median operator being reduced from 38 seconds to 16 seconds on 15 processors. The coding changes indicate that the replacement operator is an important optimisation for applicative languages because it gives the compiler the chance to recognise when an update in-place operation can be performed. Interestingly, the code is also much shorter.

## 7.2 Optimisation of the Average Operator

The average for a neighbourhood is calculated by summing all the values in the neighbourhood and dividing by the number of elements in the neighbourhood. When averaging a whole image, columns of the neighbourhoods overlap and the cost of summing mutual columns can be shared.

This optimisation, which was included in the C code, was not initially implemented in the SISAL version. We thought the cost of creating and managing extra data structures would be more expensive than the saving accrued, because data structure management in applicative languages has the potential to be more expensive than performing some simple addition operations again. The unoptimised algorithm shown in Figure 9 (a) scans the image and accumulates the average for each cell. The revised algorithm in 9 (b) forms column sums and reuses these as the window moves across the image. The improvements result in a reduction of the time taken on 15 processors from 10.3 seconds to 2.1 seconds; a substantial improvement.

## 7.3 Optimisation of the Thinning Operator

The major problem with the efficiency of the thinning algorithm arises because the sub-arrays are bigger than the number of loop iterations. When the thinning algorithm is applied to an image each pixel is examined and eroded if it is eligible, however, the edge of an image has to be treated differently as they do not have a complete neighbourhood. There are a few solutions to this problem.

One solution is to process the whole image and use an **if** statement to exclude the outer pixels of the image from processing. This results in an **if** statement being applied to every pixel in the image, significantly adding to the processing time. A solution to this problem was to only apply the **if** statement to the pixels who were candidates for thinning as only these pixels require a complete neighbourhood of pixels, all other pixels are copied through to the next iteration. This reduced the number of **if** statements executed and decreased the execution time.

Another solution, and the one which was adopted, is to process only the pixels with neighbourhoods, leaving the image reduced in size by one pixel around the outside. To add a layer of pixels around the outside of the image, single pixels are added to the start and end of each line, and an array of pixels the length of the initial image are added to the start and end of the image array, both using the **Catenate** primitive. However, we found it necessary to use the less general array **add high** and **add low** operations rather than the **Catenate** primitive because the optimisation procedures did not generate update in place code when **catenate** was used. With the better optimisation the execution time was decreased. These alternatives are shown in Figure 10 (a) - (d).

Figure 11 shows the speedup curves with the optimised code instead of the original. The results indicate the C and SISAL execution times are now much closer.

```

begin
data := if leftbound <= rightbound then
  for dataelem in old data at dataelemnum
  returns array of
    if dataelemnum = leftbound then
      old data[rightbound]
    elseif dataelemnum = rightbound then
      old data[leftbound]
    else
      dataelem
    end if
  end for
else
  old data
end if;
nextleftbound := if leftbound <= rightbound
then
  leftbound + 1
else
  leftbound
end if;
nextrightbound := if leftbound <= rightbound
then
  rightbound - 1
else
  rightbound
end if
end

```

(a) Before optimisation

```

begin
data, nextleftbound, nextrightbound :=
if leftbound <= rightbound then
  let
    rightdata := old data[rightbound];
    leftdata := old data[leftbound];
  in
    old
      data [leftbound:rightdata]
          [rightbound:leftdata]
    end let,
    leftbound + 1,
    rightbound - 1
else
  old data, leftbound, rightbound
end if
end

```

(b) After optimisation

Figure 8 - Optimisation of the Median operator

```

begin
for row in 1, height cross
  column in 1, width
  newpixelvalue :=
  if (row < distedgela + 1)
    | (row > height - distedgela)
    | (column < distedgela + 1)
    | (column > height - distedgela) then
    0
  else
    for localrow in
      row - distedgela, row + distedgela
    cross localcolumn in
      column - distedgela,
      column + distedgela
    returns value of
      sum oldimage[localrow][localcolumn]
    end for / pixelsinneighbourhood
  end if
returns array of
  if (abs(newpixelvalue
    - oldimage[row][column]) >
    threshold) then
    oldimage[row][column]
  else
    newpixelvalue
  end if
end for
end

```

(a) Before optimisation

```

begin
for row in 1, width
  columnsums :=
  if (row < distedgela + 1) |
    (row > height - distedgela) then
    array_fill(1,1,0)
  else
    for column in 1, height
      returns array of
      for localrow in row - distedgela,
        row + distedgela
      returns value of sum
        oldimage[localrow][column]
      end for
    end for
  end if;
  imagerow :=
  for column in 1, height
    newpixelvalue :=
    if (row < distedgela + 1)
      | (row > height - distedgela)
      | (column < distedgela + 1)
      | (column > height - distedgela)
      then 0
    else
      for localcolumn in
        column - distedgela,
        column + distedgela
      returns value of sum
        columnsums[localcolumn]
      end for / pixelsinneighbourhood
    end if
    returns array of
    if (abs(newpixelvalue
      - oldimage[row][column]) >
      threshold) then
      oldimage[row][column]
    else
      newpixelvalue
    end if
  end for
  returns array of imagerow
end for
end

```

(b) After optimisation

Figure 9 - Optimisation of the Average operator

```

begin
for row in 1, height cross
  column in 1, width
  newpixel, thinningdone := if (row = 1) |
    (row = height) | (column = 1) |
    (column = width) then
    0, True
  else
    if oldimage[row][column] = 255 then
      if iteronecheck
        (oldimage, row, column) then
        0, False
      else
        255, True
      end if
    else
      0, True
    end if
  end if
returns array of newpixel
value of product thinningdone
end for
end

```

10 (a) Before optimisation

```

begin
for row in 1, height cross
  column in 1, width
  newpixel, thinningdone :=
  if oldimage[row][column] = 255 then
    if (row = 1) | (row = height) |
      (column = 1) | (column = width) then
      0, True
    else
      if iteronecheck
        (oldimage, row, column) then
        0, False
      else
        255, True
      end if
    end if
  else
    0, True
  end if
  returns array of newpixel
  value of product thinningdone
end for
end

```

10 (b) With swapped if statment optimisation.

```

begin
let
  image, thinningdone :=
  for row in 2, height - 1
    imagerow, rowthinningdone :=
    for column in 2, width - 1
      newpixel, pixelthinningdone := if
        oldimage[row][column] = 255 then
          if iteronecheck
            (oldimage, row, column) then
            0, False
          else
            255, True
          end if
        else
          0, True
        end if
      returns array of newpixel
      value of product pixelthinningdone
    end for;
    paddedimagerow := rowpad || imagerow ||
rowpad
  returns array of paddedimagerow
  value of product rowthinningdone
  end for
in
  columnpad || image || columnpad,
  thinningdone
end let
end

```

10 (c) With catenation optimisation.

```

begin
let
  image, thinningdone :=
  for row in 2, height - 1
    imagerow, rowthinningdone :=
    for column in 2, width - 1
      newpixel, pixelthinningdone :=
      if oldimage[row][column] = 255 then
        if iteronecheck
          (oldimage, row, column) then
          0, False
        else
          255, True
        end if
      else
        0, True
      end if
      returns array of newpixel
      value of product pixelthinningdone
    end for;
    paddedimagerow :=
    array_addh(array_addl(imagerow, 0), 0)
  returns array of paddedimagerow
  value of product rowthinningdone
  end for
in
  array_addh(array_addl(image, columnpad),
  columnpad),
  thinningdone
end let
end

```

10 (d) With array add optimisation.

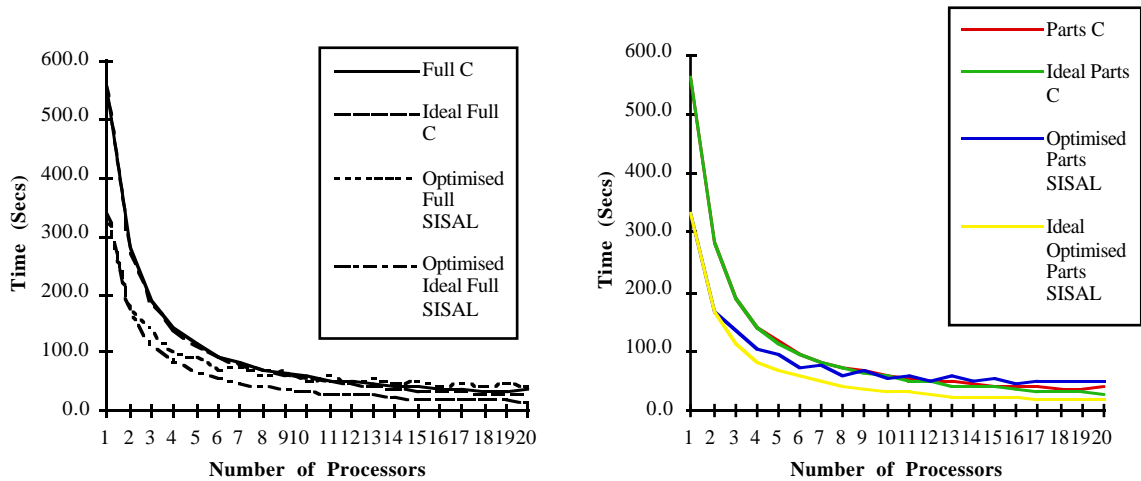


Figure 11, speed up curves for C and Optimised SISAL code of the Benchmark

## 8. 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 with out significant loss of performance. In fact, the resulting program for small number of processors is more efficient than its C counterpart. 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.

We did discover that it was necessary to recode the operations to improve their efficiency. These changes took into account knowledge of the optimisations that are performed by the SISAL compiler, namely that the use of the *replacement* operator can result in optimised code performing an update in place rather than creating a new copy of the data structure. Also, in the case of the average operator, we were too concerned by the cost of building new data structures against performing some re-computation. In this case the recomputation was still more expensive than building some intermediate data structure which could be used for caching certain computations. Similar experience has been reported elsewhere [19]. In this case the poor results we obtained initially were as a result of our pre-conceived ideas about structure handling in functional languages.

Another interesting effect of the optimisation is that the difference in time between the program when executed in parts and in full has been removed. This re-enforces our conclusion in section 6 that the data structure was being rebuilt between each image processing function call. When each of the operators was optimised, the compiler was able to pass pointers between routines.

The conclusion of this work is that the performance of programs written in applicative languages need not be worse than programs written in imperative languages. It was not our intention to compare the expressive power or structure of the language with other parallel programming languages, only to address the issue of large data structure manipulation. However, in spite of this, the lack of non-deterministic side effects significantly aided the debugging of parallel code. Also, the program size for the SISAL code was significantly shorter than the parallel C version. Whilst we did not observe any difference in the size of the C code between the versions which used static and dynamic loop unravelling, the code was, however, different. In the SISAL code this decision is left to the run time system, which currently chooses a dynamic distribution, and the source code does not need to change in the event of a different scheme being used. Thus, the run

time system is left free to choose the scheme dynamically for different program sections and under different machine loads. It is difficult to argue that one form of the program is any clearer than any other. Thus, we would not say that SISAL has provided a higher level specification of the problem. In fact, other functional and data flow languages use much higher level and abstract programming constructs than SISAL.

The work does highlight the need to understand the optimisations which are performed by a compiler in order to achieve optimal performance. This is true for sequential languages as well as parallel ones, however, since sequential languages are common place, most programmers have some *folklore* in mind when they write sequential programs. It will be some time before a similar *folklore* develops for parallel functional languages.

## ACKNOWLEDGEMENTS

This work was performed as part of The High Performance Computation Program, which was a joint program between the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) Division of Information Technology and the Royal Melbourne Institute of Technology (RMIT) from 1986 to 1993. Thanks go to Simon Wail and John Feo who reviewed a draft of this paper.

## REFERENCES

- [1] J.R. McGraw. "Data-flow computing: the VAL language". ACM Transactions on Programming Languages and systems, 4(1), Jan. 1982.
- [2] J. T. Feo, D. C. Cann and R. R. Oldehoeft. "A Report on the Sisal Language Project.", Journal of Parallel and Distributed Computing, December 1990.
- [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] D. C. Cann, "Retire Fortran: A Debate Rekindled", Communications of the ACM, August, 1992.
- [13] 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.
- [14] R. Stevens et al, "Portable Programs for Parallel Processors", Holt Rinehart and Winston Inc, 1987.
- [15] K. Preston, Jr., "Benchmark results -- The Abingdon Cross", Evaluation of Multicomputers for Image Processing, L. Uhr et al.,eds., Academic Press, Cambridge, Mass., 1986.
- [16] K. Preston, Jr., "The Abingdon Cross Benchmark Survey", IEEE Computer, July 1989.
- [17] W. Niblack, "An Introduction to Digital Image Processing", Prentice Hall.
- [18] 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.
- [19] D. Cann, "The Optimizing SISAL Compiler", Users Manual, Lawrence Livermore National Laboratories, Computing Research Group L-306.

## **Appendix - Input and output of image processing operators**

Image before processing

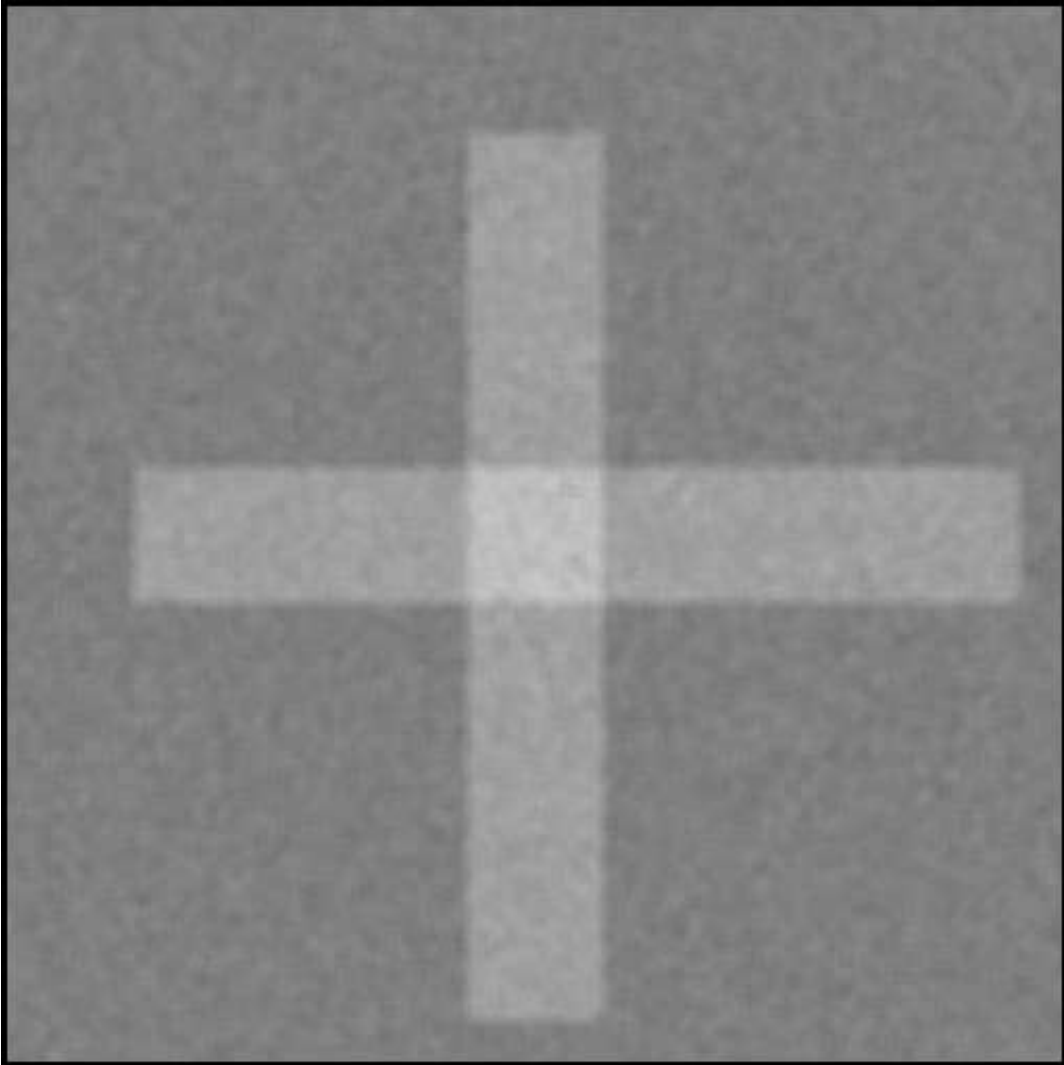


Image after average operation

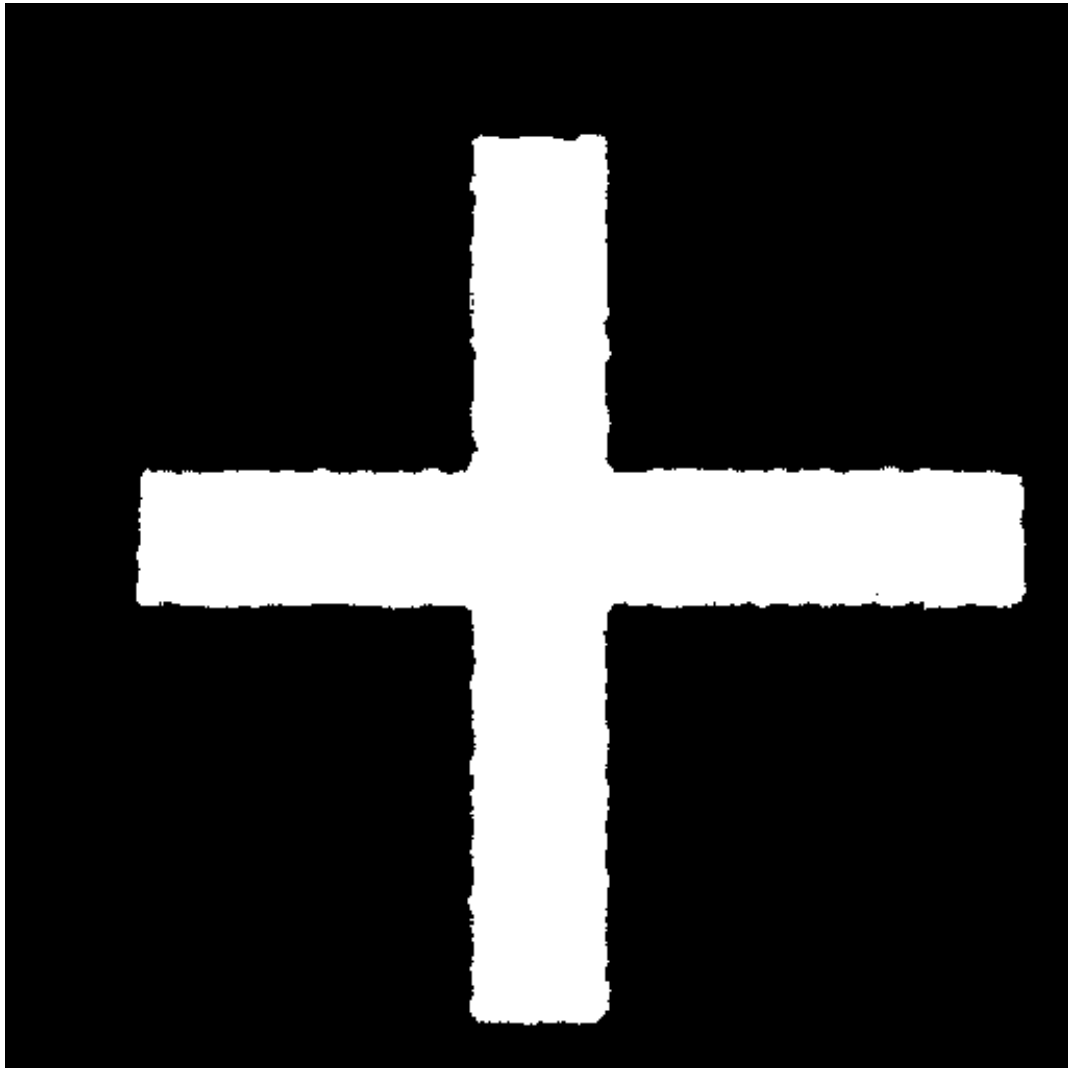


Image after threshold operation

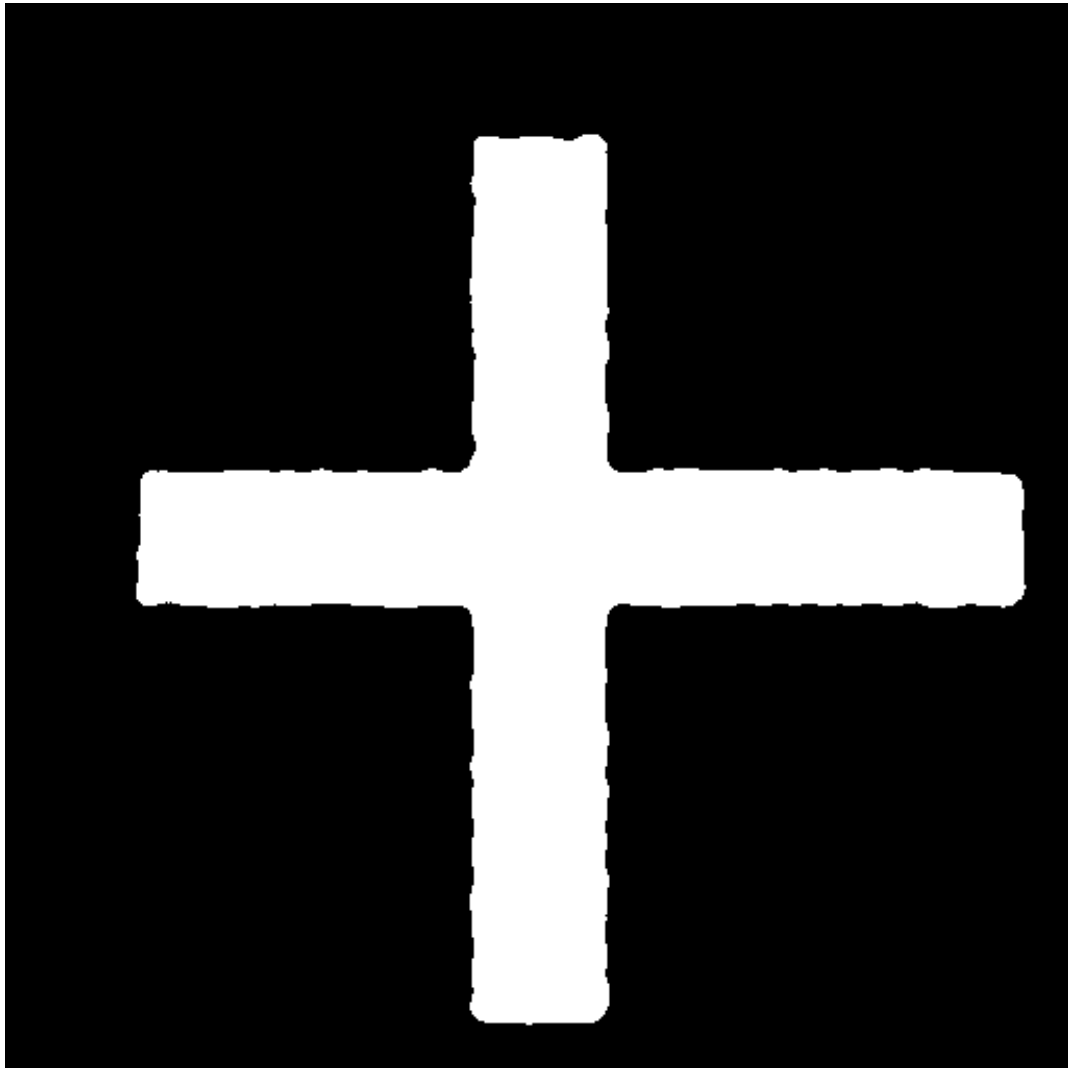


Image after median operation

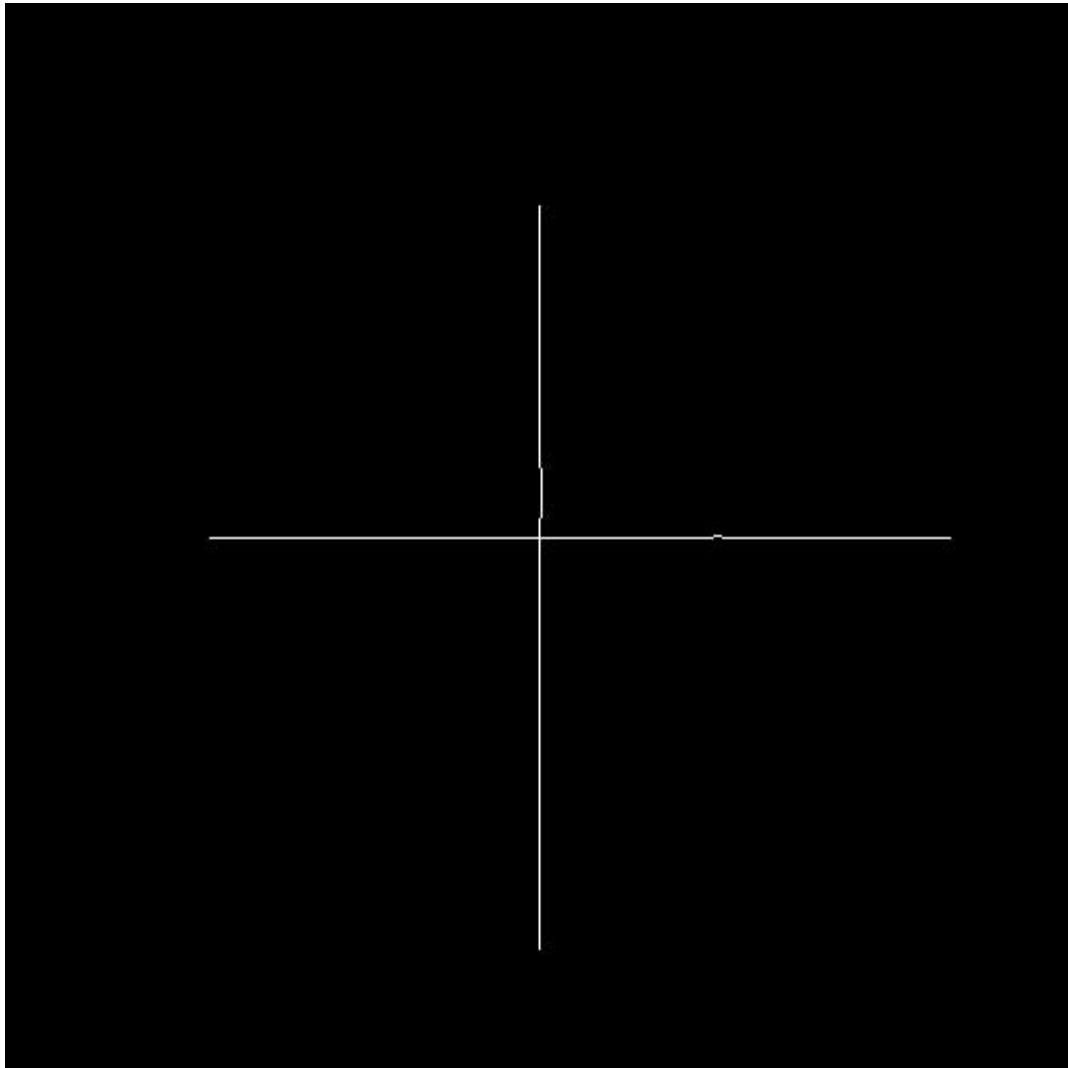


Image after thinning operation