FFT Algorithms on a Shared-Memory Multiprocessor

A.L. Cricenti and G.K. Egan.*

Abstract

This paper deals with the coding of some FFT algorithms in the functional language SISAL, to exploit the available concurrency, on a shared memory multiprocessor (Encore Multimax). Run times and speed-up are presented for two conventional array based and two pipeline stream based FFT algorithms. The performance of the stream based algorithms is compared with that of the array based algorithms.

Introduction

The Discrete Fourier Transform (DFT), and other related transforms, are of key importance in the field of digital signal processing. Calculation of the DFT from its definition is computationally expensive requiring $O(N^2)$ multiplications and a similar number of additions. However much effort has been put into developing fast methods of calculating the DFT, since efficient calculation of the DFT makes much of discrete signal processing possible. Several good algorithms now exist for the efficient calculation of the DFT, these algorithms are generally known as Fast Fourier Transforms (FFT).

To speed up the calculation of the DFT further, either faster algorithms need to be developed, or any concurrency in the algorithm be exploited. Some researchers have been looking into the parallel implementation of the FFT. Pease [Pea68] pioneered work in this area by suggesting a parallel FFT algorithm. Norton and Silberger [NS87] have presented results for a FORTRAN implementation of the Cooley-Tuckey FFT on a shared memory architecture (IBM-RP3 machine) while Cvetanovic [Cve87] presents methods for performance analysis of two FFT algorithms on shared memory machines. Adams et. al.[ABC*91] present results for parallel FFT algorithms on a connection machine and a Cray 2. Recently the performance of some FFT algorithms coded in SISAL have been presented by Cann[Can91] and Bollman[BSS92].

This paper deals with the coding of some FFT algorithms; and is also concerned with the use of the SISAL data type stream to implement pipeline FFT algorithms. The performance of these pipeline algorithms is compared with that of the standard ones.

Algorithm derivation

Calculation of the Fourier Transform of a signal involves the evaluation of the following integral:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) * e^{-j\omega t}\, dt \qquad (1)$$

In many cases the signal of interest is not a continuous time signal, but a discrete time signal. Discrete time signals are therefore sequences of numbers and therefore lend themselves for implementation and analysis on digital computers.

In the case of discrete signals the integral above becomes a summation:

$$F(n) = \sum_{k=0}^{N-1} f(k) * e^{-j2\pi kn/N} \quad n = 0,1...N-1 \qquad (2)$$

The above summation is called the Discrete Fourier Transform (DFT).

Calculation of the DFT is computationally expensive requiring order N^2 multiplications and a similar number of additions.

Many approaches for improving the computational efficiency of the DFT, rely on the properties of the $e^{-j2\pi kn/N}$ term, which is periodic and symmetric. Exploitation of these properties has led to several fast algorithms for evaluating the DFT.

Although there are several FFT algorithms, most are based on the principle of decomposing the computation into successively smaller DFT computations, this method was re-discovered by Cooley and Tuckey [CT65].

One common FFT algorithm is called the Cooley-Tuckey Radix Two Fast Fourier Transform. This algorithm is based on the successive partitioning of the data sequence into even and odd indices (note N must be a power of two hence the name Radix two).

The Radix Two algorithm can be derived by either separating the input sequence f(k) into two N/2 point sequences, Decimation in Time (DIT), or by dividing the output sequence F(n) into two N/2 point sequences, Decimation in Frequency (DIF). The number of operations in each algorithm is the same, however the DIF algorithm accepts data in natural order, and outputs the data in scrambled order, while the DIT algorithm accepts data in scrambled order, and outputs it in natural order. These features can be advantageous in convolution or correlation, as unscrambling (bit reversing) can be avoided by using an DIF algorithm to transform to the discrete frequency domain and a DIT to transform back to the discrete time domain.

As previously stated the Radix 2 DIT algorithm, for $N=2^m$ can be obtained by splitting the input sequence into two N/2 sequences consisting of the even elements and odd elements of the input sequence. The Radix 2 Cooley-Tuckey algorithm can be conveniently expressed in tensor notation [BSS92] as:

$$F_{2k} = \prod_{i=1}^{k} \{(I_2k\text{-}i \otimes F_2 \otimes I_2i\text{-}1)(I_2k\text{-}i \otimes T^{2^i})\} R(2^k) \quad (3)$$

where: $\otimes$ denotes the tensor product.

$R(2^k)$ is the bit reversal permutation.

T^{2^i} represents the twiddle factors.

$F_2 \otimes I_2i\text{-}1$ is a two point transform (butterfly).

Alternatively the FFT algorithm can be conveniently expressed as a signal flow graph as shown in figure 1. The signal flow graph has an advantage in that it shows up possible concurrency, and aids in the coding of the algorithm This algorithm is termed fast since it requires order $N\log_2 N$ multiplies rather than order N^2 multiplies to calculate the DFT of a sequence.

As evident from the signal flow graph this FFT algorithm has a high level of symmetry as well as potential concurrency, this was recognised early by Pease [Pea68] and Gold [GB73]. The concurrency can be observed by noting that the input data to each butterfly in a stage depend only on the previous butterfly or the input. Since there are no data dependencies for each butterfly in a particular stage, the calculation of all these butterflies could proceed concurrently. Although the signal flow diagram of the FFT is quite elegant, coding the algorithm to exploit the available concurrency is not as simple as it would seem. One must partition the FFT graph into segments and assign each of these to a processor. Proper synchronisation must be assured so that the data at the end of each stage is valid..

Sequential algorithm

A FORTRAN program to implement of the above FFT, due to Cooley, Lewis and Welch, adapted from [RG75], is shown in figure 2.

The program is divided into two sections; the first part is devoted to performing the bit reversal on the input sequence, such that it is in the order required for the FFT. Note bit reversal is not essential in some cases, such as computing a convolution, thus it will not be considered further. The second part of the program is concerned with the computation of the FFT. This part consists of three nested loops, the most outer loop steps through each stage of the signal flow graph, another loop performs the indexing on the powers of W as required by the butterflies, while the third loop keeps track of which butterfly calculation is being performed.

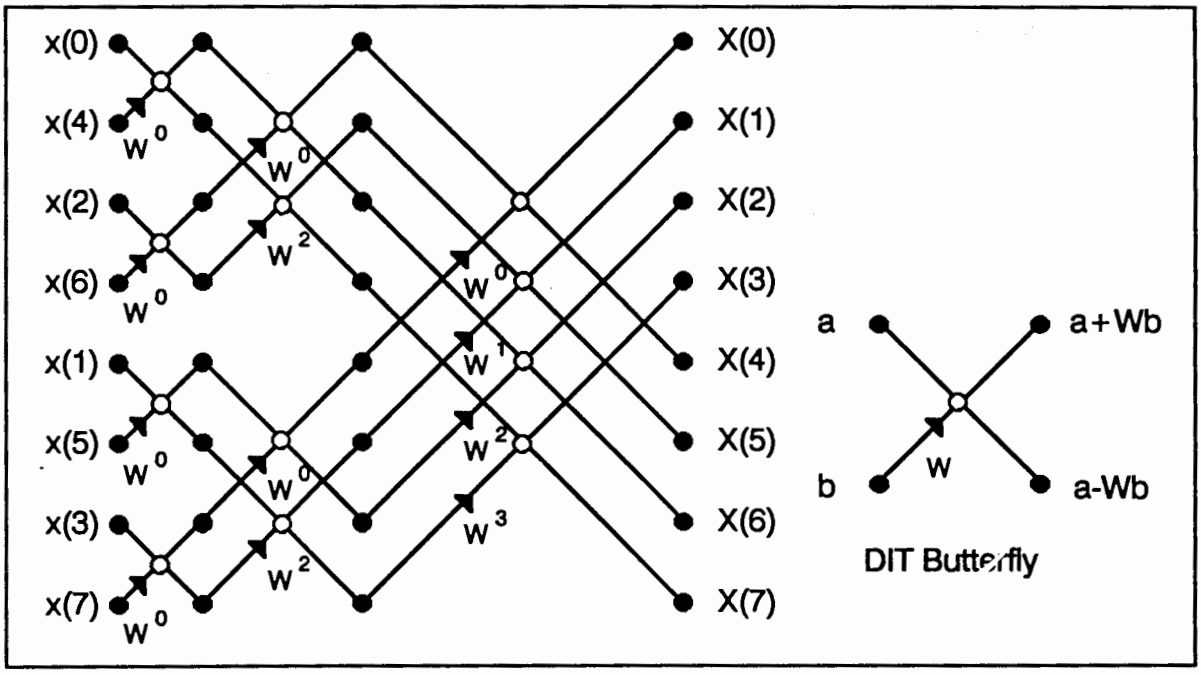

Figure 1 DIT signal flow graph for 8 point FFT.

```
      SUBROUTINE FFT(A,M,N)
      COMPLEX A(N),U,W,T
      N=2**M
      NV2=N/2
      NM1=N-1
      J=1
      DO 7 I=1,NM1                          Bit Reverse input
       IF(I.GE.J) GO TO 5
       T =A(J)
       A(J)=A(I)
       A(I)=T
    5  K=NV2
    6  IF(K.GE.J) GO TO 7
       J=J-K
       K=K/2
       GO TO 6
    7 J=J+K
      PI=3.141592653589793
      DO 20 L=1,M                           For each stage
       LE=2**L
       LE1=LE/2
       U=(1.0,0.)
       W=CMPLX(COS(PI/LE1),SIN(PI/LE1))     Calculate the new e^-j2πkn/N term
       DO 20 J=1,LE1                        Do each butterfly
        DO 10 I=J,N,LE
         IP=I+LE1
         T=A(IP)*U                          Computation of the butterflies
         A(IP)=A(I)-T
   10    A(I)=A(I)+T
   20 U=U*W                                 Update the e^-j2πkn/N term
      RETURN
      END
```

Figure 2 FORTRAN Radix 2 FFT Decimation in Time Algorithm.

The program of figure 2 if compiled to run on a shared memory multiprocessor, such as the Encore, shows no speed-up because of the way the "Twiddle Factors", ($e^{-j2\pi kn/N}$) terms are calculated, that is an initial cosine and sine term is computed, then on each iteration of the outer loop (label 20) the twiddle (W term) is updated by a recursion relation, this method of calculation is economical in terms of machine instructions, but it makes the program sequential. Since the new W value depends on the its value on the previous iteration, a data dependency exists which is not evident in the signal flow graph. The translation from FORTRAN to SISAL is quite straight forward since most of the FORTRAN control structures map directly to SISAL. One major problem in the translation is that SISAL lacks a complex number type and complex operations. This deficiency was overcome by defining a set of functions for handling complex numbers, and declaring a "type complex" as a:
Record[Realp, Imaginaryp : Double_Real].

Although this record construct overcomes the problem it leads to clumsy programming, since all operations involving this data type must be performed explicitly.

A pre declared type complex is essential for signal processing as complex data is often manipulated. The need for the complex type has previously been noted by Chang [CED90] and will be implemented in SISAL 2 [COBGF].

As expected, the SISAL program shows no speed-up. The main outer loop is sequential as can be seen from the signal flow graph, however the computation of the butterflies is also sequential, while the signal flow graph suggests that this process could be parallel. There are two reasons that make this process sequential. The first is the way the W terms are calculated. To remove this loop iteration data dependency all the W terms can be precalculated and stored in an array for access by the program. The second reason for the sequential behaviour of the loop is in the way SISAL exploits parallelism from loops. SISAL does not allow one to express a sequential process in a parallel form, since it cannot be written as a *product for* form loop. One expects that the loop which would step through each

butterfly calculation can be expressed in a parallel fashion, since there are no data dependencies between butterflies in the same stage.

When the data operated upon is stored as an array, then SISAL requires that the elements of that array are processed in order, if the product form for loop is to be used. Thus SISAL can only slice loops such as the following array build statement

```
A:=for i in 1,10
    returns array of i
    end for
```

This loop is considered concurrent by OSC since the elements of A are created in order, ie A[1],A[2]...A[10], thus the concurrency of this loop can be exploited by loop slicing. Note OSC achieves concurrency by loop slicing, on a shared memory multiprocessor.

The Cooley-Tuckey FFT algorithm is not easily expressed in SISAL in a way that achieves useful speed-up. The problem with this algorithm is that the elements of the array output from each stage are not produced in order. An alternative algorithm is required which can be expressed in SISAL in a parallel form.

Constant geometry algorithm

As seen from the signal flow graph the output vector of each stage of the FFT is not produced in order, but the elements of the vector come out in a different order for each stage. This implies that the sequential SISAL non product *for* loop must be used. To overcome the limitation of arrays in SISAL being built in a strict sense for parallel loop constructs, the FFT algorithm must be modified so that the elements of the output of each FFT stage, are produced in order. The reason for the elements being produced out of order is because the Cooley-Tuckey FFT program is based on a so called "In-place algorithm". This means that each butterfly output is put back into the index where it came from. This is desirable since it means that only one array is required to implement the program. In a SISAL implementation this is not an advantage since a copy of the old array may exist, (*old*) when using a non product *for* loop. If the restriction of in-place computation can be relaxed then the indexing can be kept constant from stage to stage and in order. This allows the inner loop, which performs each butterfly for a given stage, to be expressed in the product *for* form loop. This algorithm is termed "Constant Geometry" [RG175] and the signal flow diagram is shown in figure 3. Note the signal flow graph below represents a DIF algorithm, the DIT version would have the powers of W in the bottom wing of the input to the butterfly. Since each stage of the constant geometry algorithm is the same, refer to signal flow graph, the program is simple to express.

Pipeline algorithms

The FFT algorithms presented thus far rely on the data being fed into the algorithm in a parallel fashion, that is as an array. In practice the data would arise from sampling a signal at discrete time intervals. In this case the data would arrive in a sequential fashion.

Inspection of the signal flow graphs shows that the butterflies in a particular stage could be processed in any order. In fact it is not necessary to fully complete a stage before commencing the next stage, subject to the availability of the appropriate data for the next stage.

When using arrays, data cannot, in general, be output from each stage of the algorithm until each stage has been completed, that is all of the elements of each array must be computed. Arrays therefore restrict the amount of exploitable concurrency in the FFT algorithm.

In some situations, it is desirable to pass the data in a serial fashion to the FFT and have it output in a serial fashion. This scheme is attractive since it is possible have data output at the sampling rate, once the FFT has been primed with data. An algorithm which achieves this is a pipeline algorithm. Several pipeline Fast Fourier Transforms have been presented in the literature, some of these however are usually implemented with special purpose hardware [GB73][GW70][SJ90].

A pipeline algorithm [GB73] can be derived by considering the FFT signal flow graph of figure 4, this is the graph of an 8 point DIF algorithm, note input data are normally ordered, output data are bit reverse ordered. The DIF version is chosen as it is assumed that the input data are in natural sequential order.

It can be seen from the graph that to co ute the X(0) and X(4) output points only three butterflies need to be evaluated, as shown in figure 5. A similar situation exists for the other output points.

The input to the first butterfly stage is:

$$x(t) = x(k) \qquad k=0...N/2-1 \qquad (4)$$
$$x(b) = x(k+N/2) \qquad k=0...N/2-1$$

that is two points separated by N/2 where N is the FFT length. Therefore the first N/2 input samples to the pipeline are routed to the top arm of the butterfly while the next N/2 samples to the bottom. A delay of N/2 is used in the top arm of the butterfly to ensure that the data at the butterfly are synchronised. The appropriate W^p must be used at the butterfly output. The data leaves the butterfly in parallel pairs. The input to the second butterfly stage is:

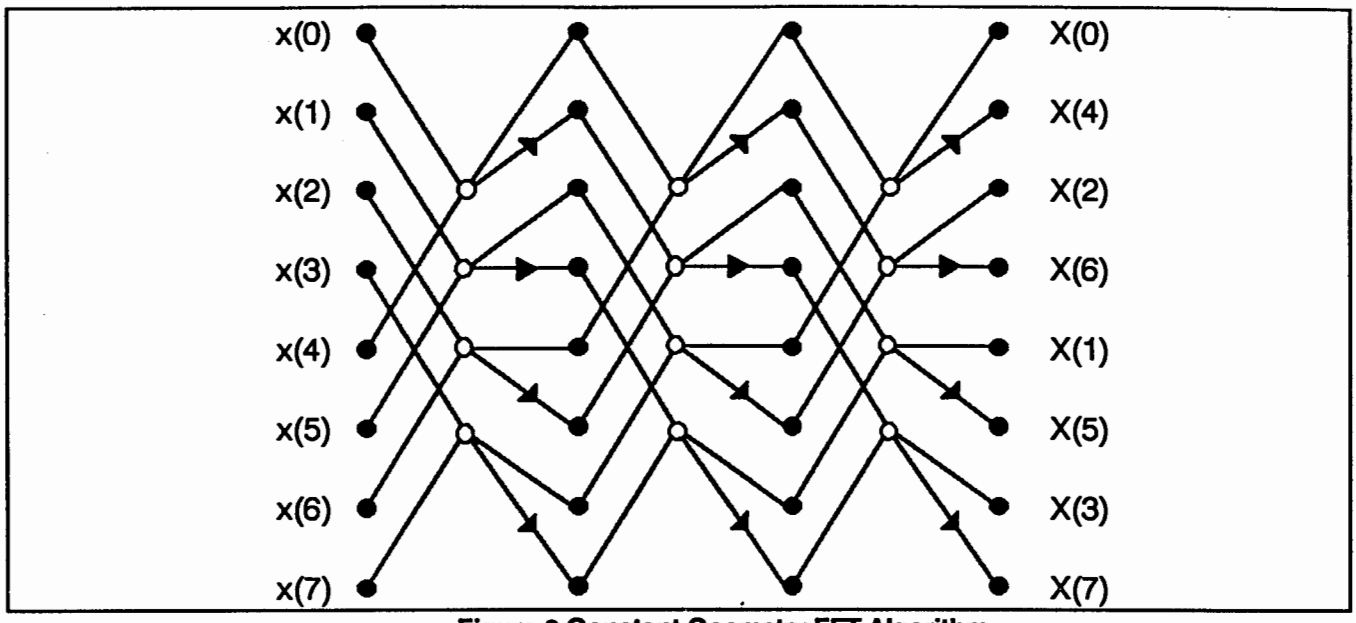

Figure 3 Constant Geometry FFT Algorithm.

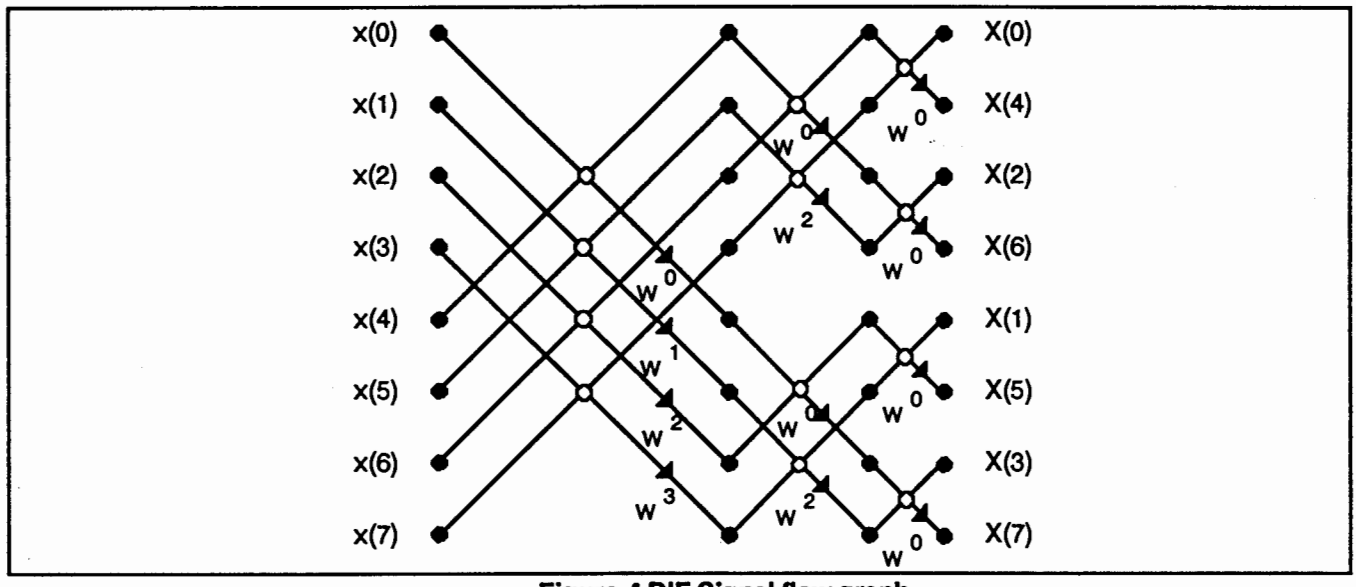

Figure 4 DIF Signal flow graph.

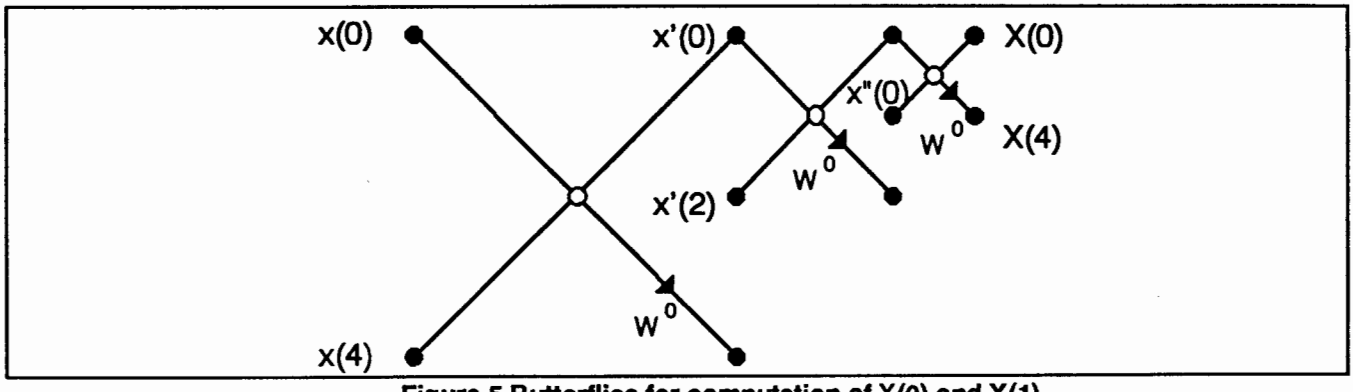

Figure 5 Butterflies for computation of X(0) and X(1).

x'(t) and x'(t+N/4) t=0...N/4-1 (5)
x'(b) and x'(b+N/4) b=0...N/4-1 (6)

where x'(t) is the output of the top arm of the first butterfly and x'(b) is the output of the bottom arm of the first butterfly. Again the appropriate W^p must be applied. The argument can be continued for the third and following stages of the pipeline and the results this shown in figure 6.

The elements of the data sequence must be routed to the appropriate arm of each butterfly, this is achieved by the switches and delay lines in the pipeline. Each switch in the pipeline switches at twice the frequency of its predecessor, and the delay lengths in a given stage are half that of the previous stage. The first switching block works as follows:

-The data samples are routed straight through for the first N/4 elements.

-The samples are crossed over for the next N/4 samples.

SISAL has an appropriate data type for pipeline algorithms, that is the type STREAM [Can89][MSA*85]. A stream is a sequence of values of uniform type that allows pipeline concurrency to be expressed directly by their use. Streams differ from arrays in that the elements of the stream can only be accessed in sequence, no subscripting or random access is possible. This form of access allows the use of elements from the stream without having to produce the complete stream as would be required with an array. By definition SISAL streams require a non-strict implementation. A problem occurs with the use of streams in OSC because OSC implements streams strictly [CO]. As a consequence of this implementation, pipeline concurrency is not exploited. In addition parallel loops are difficult to write with streams because only the first element of the stream can be accessed.

The switch blocks of the pipeline are implemented by a function *stream_switch*. This function is complicated since the switching period is stage dependent. Another problem arises because the two streams that enter the function *stream_switch* are not processed simultaneously, that is the part of top stream is processed before the bottom stream. The delays in the pipeline are meant to cope with this problem. The software implementation uses the SISAL ***when & unless*** masking clauses to filter out the unwanted values from the ***returns*** part of the *for* loop.

The pipeline algorithm could be made simpler to express if the switching block could be simplified. To achieve this aim the switching should be made independent of the stage of the pipeline. Again a constant geometry algorithm could be used to achieve this. A pipeline implementation of the constant geometry algorithm is shown in figure 7.

It can be seen that each stage of the pipeline is identical. This feature makes the expression of this pipeline algorithm straight forward.

The switch blocks for this algorithm are very simple since they are composed of two switches. The first switch switches the sample rate, while the second switch switches at a rate of N/2 samples.

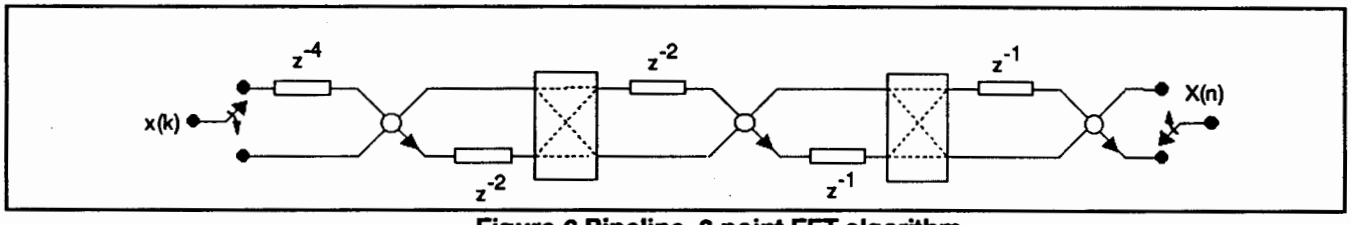

Figure 6 Pipeline 8 point FFT algorithm.

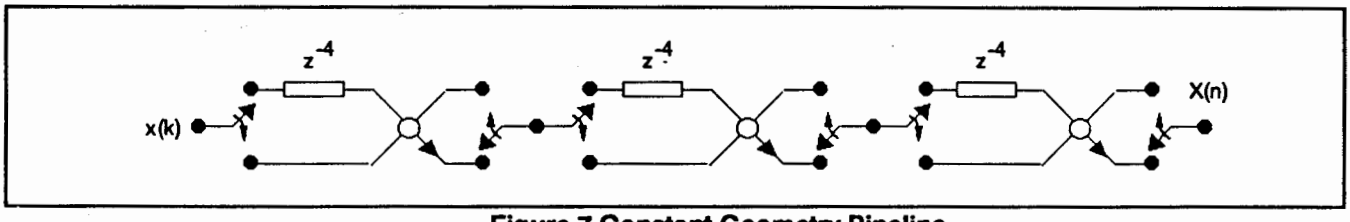

Figure 7 Constant Geometry Pipeline.

Results

The various algorithms presented in the previous section were all run on an Encore shared memory multiprocessor, using OSC as the compiler. The OSC -h500 switch was used to ensure that the compiler considered all concurrent loops for slicing. The value of 500 was arrived at by trial and error. Run-times were obtained using the SPEED-UPS routine from the OSC library. All times are in seconds and the longest wall time was chosen when multiple processors were used. Generally in cases where there was a significant difference between the wall time and the CPU time the results were discarded (CPU utilisation <95%). When multiple processor run-times showed a significant difference for each processor the results were also discarded.

All input data were generated internally by each program, therefore the run-times reflect this. Output data from the programs was suppressed by using the -z switch.

Speed-up is defined as T_1 proc / T_n procs.

Various FFT sizes were used in the study to check the performance of the algorithms against model size.

Sequential algorithm

Tables 1 and 2 present the results for run-time and speed up for the Sequential FFT algorithm.

As can be seen from table 2, the Sequential FFT algorithm shows no significant speed-up. This is to be expected since the algorithm is sequential.

Constant geometry algorithm

Tables 3 and 4 present the results for run-time and speed up for the Constant Geometry FFT algorithm.

The constant geometry algorithm achieves good speed-up, refer to table 4 and figure 8. The single processor run-times are longer than the sequential algorithm (table 1), as is to be expected because of the less efficient calculation of the indexes of the W factors used in the butterflies. However because of the speed-up obtained, this code becomes faster than the sequential algorithm.

The droop in the speed-up graph for 10 -16 processors is due to other processes competing for resources. The speed-up improves with the size of the FFT and high efficiencies are obtained for N > 4096.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.6	1.3	2.78	6.240	13.140	28.620	59.640
2	0.58	1.28	2.74	6.140	13.000	28.400	59.140
3	0.58	1.28	2.74	6.120	12.960	28.380	59.180
4	0.6	1.3	2.78	6.100	12.980	28.420	59.040
5	0.58	1.3	2.76	6.120	13.000	28.420	59.480
6	0.58	1.26	2.78	6.140	13.020	28.680	59.160

Table 1 Run-time vs FFT length for the Sequential FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.03	1.02	1.01	1.02	1.01	1.01	1.01
3	1.03	1.02	1.01	1.02	1.01	1.01	1.01
4	1.00	1.00	1.00	1.02	1.01	1.01	1.01
5	1.03	1.00	1.01	1.02	1.01	1.01	1.00
6	1.03	1.03	1.00	1.02	1.01	1.00	1.01

Table 2 Speed-up vs FFT length for the Sequential FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.82	1.8	3.9	8.700	18.000	38.100	80.860
2	0.44	0.96	2.02	4.480	9.280	19.680	41.540
3	0.32	0.64	1.36	2.940	6.200	13.140	27.820
4	0.24	0.5	1.06	2.300	4.680	9.900	20.940
5	0.2	0.4	0.86	1.800	3.780	8.020	16.860
6	0.18	0.36	0.74	1.520	3.180	6.720	14.160
7	0.16	0.32	0.64	1.340	2.780	5.800	12.560
8	0.14	0.28	0.600	1.200	2.440	5.160	11.140
16	0.1	0.2	0.440	0.740	1.440	2.920	6.200

Table 3 Run-time vs FFT length for the Constant Geometry FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.86	1.88	1.93	1.94	1.94	1.94	1.95
3	2.56	2.81	2.87	2.96	2.90	2.90	2.91
4	3.42	3.60	3.68	3.78	3.85	3.85	3.86
5	4.10	4.50	4.53	4.83	4.76	4.75	4.80
6	4.56	5.00	5.27	5.72	5.66	5.67	5.71
7	5.13	5.63	6.09	6.49	6.47	6.57	6.44
8	5.86	6.43	6.50	7.25	7.38	7.38	7.26
16	8.20	9.00	8.86	11.76	12.50	13.05	13.04

Table 4 Speed-up vs FFT length for the Constant Geometry FFT.

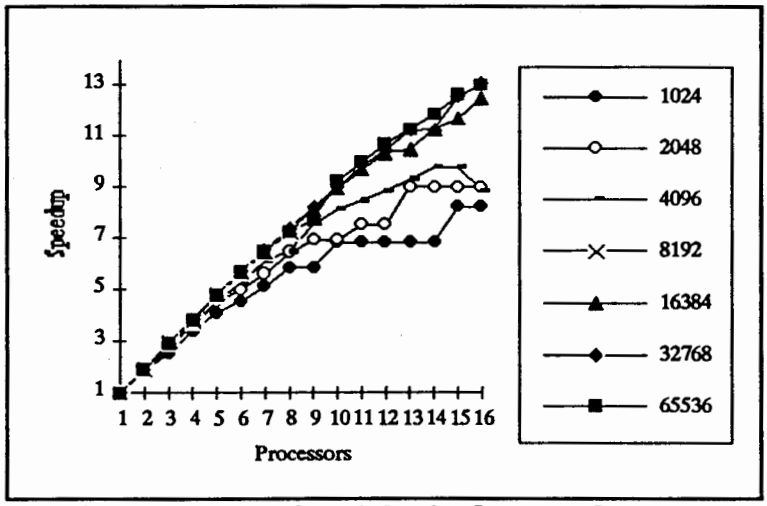

Figure 8 Speed-up vs FFT length for the Constant Geometry FFT.

Pipeline algorithms

Tables 5 and 6 present the results for run-time and speed up for the Pipeline FFT algorithm.

The pipeline algorithm shows poor speed-up, mainly because the implementation of this algorithm introduces a large amount of array copying, which severely affects its performance. The OSC compiler warns when array copying may occur.

Tables 7 and 8 present the results for run-time and speed up for the Constant Geometry Pipeline FFT algorithm.

The constant geometry pipeline code performs better than the previous pipeline algorithm, but still suffers from array copying.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.45	1.22	3.11	9.35	25.44	63.06	181.83
2	0.44	1.20	2.51	7.36	20.79	57.54	160.45
3	0.43	1.16	2.45	7.10	20.10	53.82	153.59
4	0.44	1.05	2.57	7.14	19.54	52.03	150.77
5	0.43	1.06	2.51	7.09	19.23	52.26	150.02
6	0.43	1.10	2.46	7.16	22.71	56.14	159.57

Table 5 Run-time vs FFT length for the Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.04	1.02	1.24	1.27	1.22	1.10	1.13
3	1.07	1.05	1.27	1.32	1.27	1.17	1.18
4	1.04	1.17	1.21	1.31	1.30	1.21	1.21
5	1.06	1.15	1.24	1.32	1.32	1.21	1.21
6	1.05	1.11	1.26	1.31	1.12	1.12	1.14

Table 6 Speed-up vs FFT length for the Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1.16	2.28	6.1	14.600	35.800	87.940	214.400
2	0.76	1.88	3.96	10.060	25.720	63.540	170.820
3	0.6	1.56	3.28	8.580	22.420	56.500	156.300
4	0.54	1.44	2.96	7.820	20.780	53.340	149.640
5	0.5	1.34	2.74	7.420	19.820	52.040	145.820
6	0.46	1.26	2.6	7.140	19.440	50.800	144.900

Table 7 Run-time vs FFT length for the Constant Geometry Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.34	1.34	1.45	1.44	1.36	1.32	1.30
3	1.43	1.50	1.61	1.63	1.50	1.46	1.39
4	1.58	1.60	1.73	1.82	1.54	1.48	1.39
5	1.66	1.74	1.88	1.81	1.59	1.53	1.43
6	1.60	1.86	1.84	1.86	1.66	1.55	1.46

Table 8 Speed-up vs FFT length for the Constant Geometry Pipeline FFT.

Both pipeline algorithms require a function to implement the switch that switches with a period of N/2 samples. This function must separate the input stream into two streams, one containing the first N/2 samples the other the rest of the stream. An obvious way of expressing this switching function in SISAL is as:

```
for aelement in input at i
returns stream of aelement when i<=length
        stream of aelement when i>length
end for
```

Figure 9 'Product form for loop' for switch.

This approach has two problems when OSC is used to compile it. The first problem is that OSC will not slice this loop as the compiler considers it sequential because of the *when* clause. Thus using the product for form loop is not an advantage.

The second problem with this approach is that this loop introduces copying. To overcome some of the copying, the loop can be expressed as the non product form for loop as shown in figure 10. This implementation of the loop is much faster as some array copying is eliminated. Note this was verified using the OSC *-time* switch.

```
for initial
  i:=1;
  bottom:=input
until i>length repeat
  i:=old i+1;
  bottom:=stream_rest(old bottom);
returns stream of stream_first(bottom) unless i>length
% This introduces copying
        value of bottom
end for
```

Figure 10 'Non product form for loop' for switch

The array copying which results from this loop is still significant both in terms of run-time and speed-up performance. This is particularly noticeable for large FFT lengths. If the *unless* clause is removed from the above loop, the execution time is reduced considerably, and the speed-up performance is improved, refer to table 10. Removal of the *unless* clause, is not practical since the code produces incorrect results. The speed-up improvement implies that the array copying due to the unless clause is of a sequential nature.

Processors	1024	16384	65536
1	1.04	22.52	101.4
2	0.60	12.70	57.22
4	0.38	7.70	34.84
6	0.32	6.58	27.66

Table 9 Run-time vs FFT length for the Constant Geometry Pipeline FFT without *unless*.

Processors	1024	16384	65536
1	1	1	1
2	1.73	1.77	1.77
4	2.74	2.92	2.91
6	3.25	3.42	3.67

Table 10 Speed-up vs FFT length for the Constant Geometry Pipeline FFT without *unless*.

The shared memory machine and the OSC code do not exploit the available pipeline concurrency. Therefore to better compare the pipeline algorithm with the array based (non pipeline) algorithms, it was coded using arrays rather than streams.

Performance of the array based pipeline algorithm is much better than the stream version since the array copying is eliminated. By eliminating the array copying one can then express the function **switch** as a parallel for construct, thereby improving the speed-up performance.

Run-time and speed-up results, for the array based pipeline algorithm, are given in tables 11 and 12. The results show that the use of arrays in OSC gives better performance than the use of streams. The performance of this pipeline algorithm is similar to the constant geometry algorithm, refer to tables 3 and 4.

Note a dataflow machine would possibly achieve a better result for the stream based pipeline algorithms as pipeline concurrency could be exploited assuming a non strict implementation of the streams. However at the time of writing a dataflow machine that implements SISAL streams was not available. It is expected that the CSIRAC II simulator will eventually support the stream type.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.9	1.94	4.22	9.080	19.280	41.360	86.680
2	0.48	1	2.16	4.600	9.820	20.980	43.960
4	0.26	0.54	1.12	2.380	5.000	11.100	22.520
8	0.18	0.32	0.680	1.340	2.820	5.920	12.380
16	0.16	0.26	0.520	1.000	1.960	4.060	8.540

Table 11 Run-time vs FFT length for the Constant Geometry (Array) Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.88	1.94	1.95	1.97	1.96	1.97	1.97
4	3.46	3.59	3.77	3.82	3.86	3.73	3.85
8	5.00	6.06	6.21	6.78	6.84	6.99	7.00
16	5.63	7.46	8.12	9.08	9.84	10.19	10.15

Table 12 Speed-up vs FFT length for the Constant Geometry (Array) Pipeline FFT.

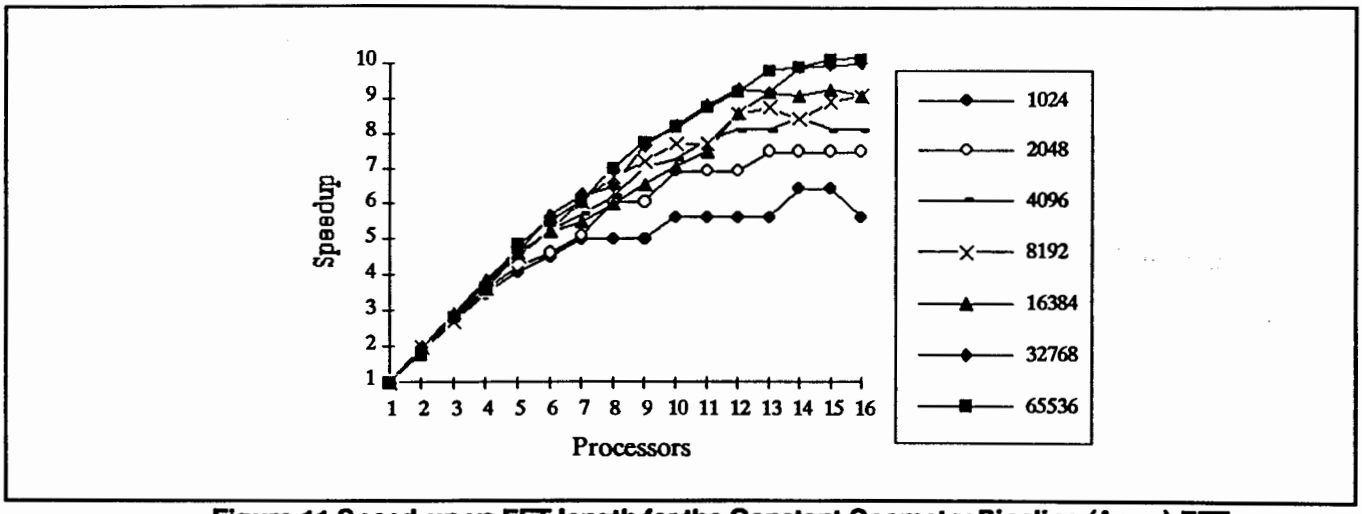

Figure 11 Speed-up vs FFT length for the Constant Geometry Pipeline (Array) FFT.

Conclusions

The paper has presented several FFT algorithms which have been coded in SISAL. The Cooley-Tuckey FFT algorithm is difficult to code in SISAL in a way that exploits the concurrency which seems to be present in the algorithm. This is due to the order in which the output elements of each butterfly stage are produced. Useful speed-up has been demonstrated, for the constant geometry FFT algorithm, without significant programming effort. However SISAL's lack of a complex number type and operations, leads to clumsy and long code.

For signal processing the data type stream is of key importance as the data in this field occurs naturally as a stream. Here SISAL could have an advantage over other languages. However the strict implementation of streams is inefficient making the writing of code difficult if copying is to be avoided. The use of strict streams limits the possibility of real time signal processing as the whole of the input data, must be in memory before processing can commence.

References

[ABC*91] Adams D.E., Bronson E.C., Casavant T.L., Jamieson L.H., Kamin R.A., "Experiments with Parallel Fast Fourier Transforms", Parallel Algorithms and Architectures for DSP Applications, pp 49-75 Kluwer Academic Publishers 1991.

[BSS92] Bollman D., Sanmiguel F., Seguel J, "Implementing FFT's in SISAL" Proceedings of the Second Sisal Users' Conference. pp 59-65, December 1992.

[Can89] Cann, D.C. "Compilation Techniques for High Performance Applicative Computation" Technical Report CS-89-108, Colorado State University, pp 12-13, May, 1989.

[Can91] Cann D.C. "Retire Fortran? A Debate Rekindled", Technical Report UCRL-JC-107018, Lawrence Livermore National Laboratory, April 1991.

[CED90] Chang P. and Egan G.K., "An Implementation of a Numerical Weather Prediction Model in SISAL" Technical Report 31-017, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology December. 1990.

[CO] Cann D., Oldehoeft R.R., "A guide to the Optimising SISAL Compiler" p32.

[COBGF] Cann D., Oldehoeft R.R., Bohm A.P.W., Grit D., Feo J., " SISAL Reference Manual, Language Version 2.0", Colorado State University and Lawrence Livermore National Laboratory, 1990.

[CT65] Cooley J.W., Tukey J.W., "An Algorithm for the Machine Calculation of Complex Fourier Series", Math. Comput., Vol. 19, pp 297-301 April 1965.

[Cve87] Cvetanovic Z., "Performance Analysis of the FFT Algorithm on a Shared-Memory Parallel Architecture", IBM J. Res. Develop. Vol. 31 No. 4, pp 435-451 July 1987.

[GB73] Gold B., Bially T., "Parallelism in Fast Fourier Transform Hardware", IEEE Trans. Audio Electroacoust., Vol. AU-21 No 1, pp 5-16 February 1973.

[GW70] Groginski H.L., Works G.A., "A Pipeline Fast Fourier Transform", IEEE Trans. Comput. Vol. C-19, pp 1015-1019 November 1970.

[MSA*85] McGraw J., Skedzielewski S., Allen S., Oldehoeft R., Glauert J., Kirkham C., Noyce B. and Thomas R., "SISAL: Streams and Iteration In a Single Assignment Language, Language Reference Manual, Version 1.2", Lawrence Livermore National Laboratory, March 1985.

[NS87] Norton V.A., Silberger A., "Parallelization and Performance Prediction of the Cooley-Tuckey Algorithm for Shared Memory Architectures", IEEE Trans. Comput. Vol. C-36, No 5, pp 581-591 May 1987.

[Pea68] Pease M.C., "An Adaptation of the Fast Fourier Transform for Parallel Processing", J. Ass. Comput. Mach., Vol 15, pp 252-264 April 1968.

[RG75] Rabiner L.R., Gold B., "Theory and Application of Digital Signal Processing", Prentice-Hall 1975 p 367.

[RG175] Ibid, pp 575-576.

[SJ90] Sapiecha K., Jaroki R., "Modular Architecture for High Performance Implementation of the FFT Algorithm", IEEE Trans. Comput. Vol. C-39, No 12, pp 1464-1468 December 1990.

Acknowledgements

The authors thank the members of the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, for their assistance in this research.

The authors thank the Royal Melbourne Institute of Technology (RMIT) for the use of the Encore facility.

The Laboratory for Concurrent Computing Systems is funded under a special research infrastructure grant for parallel processing research by the Australian Commonwealth Government.

Note all FFT codes are available from the principal author.

---

* A.L. Cricenti is a member of Faculty in the School of Electrical Engineering and a Researcher in the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, John Street, Hawthorn 3122, Australia, Phone:+61 3 819 8516, E-mail: alc@stan.xx.swin.oz.au.

G.K. Egan is Professor of Computer Systems Engineering and Director of the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, John Street, Hawthorn 3122, Australia, Phone:+61 3 819 8516, E-mail: gke@stan.xx.swin.oz.au.