PARALLELISATION AND PERFORMANCE OF THE BURG ALGORITHM ON A SHARED MEMORY MULTIPROCESSOR

A.L. Cricenti and G.K. Egan·

Abstract

This paper describes the implementation of a signal processing algorithm, specifically the Burg Algorithm, using both a high level al parallel language SISAL and Encore Parallel FORTRAN. The Burg Algorithm is an estimation technique for fitting an autoregressive model to a time series data set. This algorithm contains a time shift/inner product operation which is used in a number of other important signal processing algorithms such as convolution. The paper describes the results obtained using both the high level parallel language SISAL and EPF, on both an ENCORE Multimax multiprocessor machine, and a single processor IBM RS6000/530 machine.

1. Introduction

Signal Processing Algorithms are widely used and of vital importance in areas such as biomedical engineering, seismic data analysis, speech analysis and spectral estimation. The demand that signal processing algorithms place on computing system performance is increasing as more complicated algorithms are made to function in real time. As the limitations of current uniprocessor systems are being reached, many computer manufacturers are turning to multiprocessor configurations to obtain increased performance. In addition to hardware limitations, current computer languages must be evaluated for performance and ease of use with reference to their suitability for parallel machines.

The purpose of this paper is to describe the implementation of a signal processing algorithm, in this case the Burg Algorithm[1], using both a high level parallel language SISAL (Stream and Iteration in a Single Assignment Language)[2] and EPF (Encore Parallel FORTRAN). The Burg Algorithm is an estimation technique for fitting an autoregressive model to a time series data set. This algorithm contains a time shift / inner product operation which is used in a number of other important signal processing algorithms such as Convolution.

It is claimed that the optimising SISAL compiler (OSC) from Colorado State University yields performance competitive with FORTRAN [5][8]. Also the maximum concurrency of a SISAL program is theoretically only limited by the data dependencies. However, the OSC compiler on a shared memory machine, only exploits parallelism from the parallel loop construct.

The results obtained using an optimising SISAL compiler are compared to those obtained using the EPF (parallel FORTRAN) annotator. The comparison is made both on a 6 XPC processor (4 Mips per processor) based Encore Multimax Multiprocessor machine and a

---

·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 Institute of Technology, John Street, Hawthorn 3122, Australia, Phone: +61 3 819 8322, 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 Institute of Technology, John Street, Hawthorn 3122, Australia, Phone: +61 3 819 8516, E-mail: gke@stan.xx.swin.oz.au.

single processor IBM RS6000/530 (30 Mips) system. The performance of the IBM processor is representative of processors in next generation medium cost multiprocessors.

2. The Burg Algorithm

The Burg Algorithm is a method of generating an autoregressive model from a set of data samples, that is it gives estimates for A(z) in:

$$H(z) = \frac{1}{A(z)}$$

There are several ways of obtaining an AR model, the Burg algorithm is based on minimising the forward and backward prediction errors, assuming a lattice filter structure as shown in figure 1.

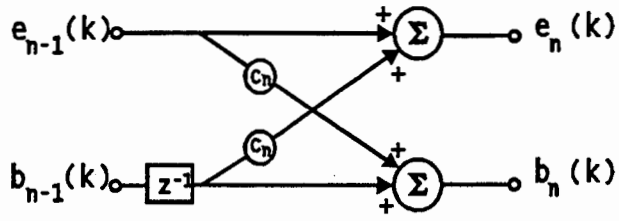

Figure 1 Lattice filter structure

where:

$$e_n(k) = e_{n-1}(k) + c_n\, b_{n-1}(k-1) \quad \text{forward prediction error} \qquad (1)$$

$$b_n(k) = c_n\, e_{n-1}(k) + b_{n-1}(k-1) \quad \text{backward prediction error} \qquad (2)$$

and c_n are called the reflection coefficients.

The Burg Algorithm involves the choice of reflection coefficients such that the error energy is minimised, when only a finite number of data samples is available.

The optimum value of the reflection coefficients can be easily derived[7] and is given by:

$$c_n = -2\frac{\sum_{k=n}^{M} e_{n-1}(k)*b_{n-1}(k-1)}{\sum_{k=n}^{M} e^2_{n-1}(k)*b^2_{n-1}(k-1)} \qquad (3)$$

where M is the number of data samples.

The autoregressive coefficients can then be estimated from:

$$a_n = c_n \qquad (4)$$

$$a_n(j) = a_{n-1}(j) + c_n a_{n-1}(n-j) \quad \text{for } j=1..n-1 \qquad (5)$$

A sequential implementation of the Burg algorithm is outlined in [1] and is reproduced in figure 2a with individual tasks labelled $T_{in}(1)$, $T_n(1)$, $T_{nn}(2)$, $T_{in}(2)$, $T_{in}(3)$. Tasks $T_{in}(1)$ are computations of the inner products in both the numerator and denominator of (3) above. $T_n(1)$ is the calculation of the division needed to compute c_n. This task cannot proceed in parallel, since it depends on the results of task $T_{in}(i)$. This data dependency can be also be seen from the maximally parallel graph for **m**=5 and **max**=3 reproduced in figure 2b. $T_{in}(2)$ updates the autoregressive coefficients and task $T_{in}(3)$ updates the forward and backward prediction errors (equations 1 and 2), the graph of figure 2b shows that each of the loops corresponding to tasks $T_{in}(1)$, $T_{in}(2)$, $T_{in}(3)$ can be computed in parallel.

```
1. INITIALIZATION

   FOR i=1 TO m DO
      e(i)=x(i)
      b(i)=x(i)

2. THE MAIN LOOP

   FOR n=1 TO max DO
      s1=s1+e(i)*b(i-n)                    Tin(1)
      s2=s2+e(i)**2+b(i-n)**2
   c(n)=-2.0*s1/s2                         Tn(1)
   IF n>1 THEN DO
      FOR i=1 TO n-1 DO
      a1(i)=a(i)+c(n)*a(n-i)               Tin(2)
      FOR i=1 TO n-1 DO
         a(i)=a1(i)
   a(n)=c(n)                               Tnn(2)
   FOR i=n+1 TO m DO
      temp=e(i)+c(n)*b(i-n)                Tin(3)
      b(i-n)=b(i-n)+c(n)*e(i)
      e(i)=temp
```

Figure 2a Sequential Burg Algorithm for m data points and max reflection coefficients from [1]

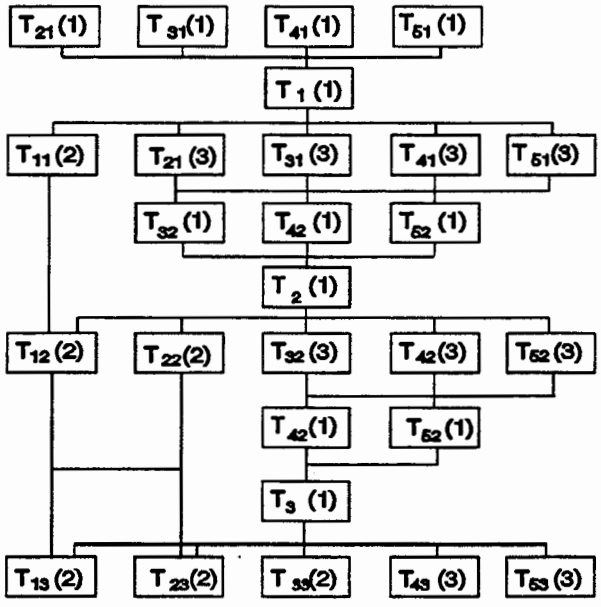

Figure 2b Maximally parallel graph for m=5 and max=3 from [1]

3. Language Considerations

3.1 Encore Parallel FORTRAN

The Encore Parallel FORTRAN compiler (EPF) is the UMAX f77 implementation of FORTRAN with enhancements which allow parts of a program to be executed in parallel. These statements are PARALLEL, DO ALL, CRITICAL SECTION, BARRIER, LOCK WAIT, LOCK SEND, and EVENT.

The EPF compiler consists of analysis and transformation tools, a parallelising compiler, parallel runtime libraries, and a code generator. Whilst programs can be written directly in EPF, EPF can also be used to convert a standard FORTRAN program into a source which is annotated with the parallel statements outlined above. During compilation, EPF first detects possible concurrent parts of the source programs, these are shown in a .LST file. The annotator then generates the EPF program , .E file, by inserting the appropriate EPF statements.

The EPF annotator may require user intervention to produce the most efficient code for a particular program; useful speedup can be achieved by fine tuning output of the annotator. However, for the simple code presented here, it is sufficient to rely on the annotator alone.

3.2 SISAL

SISAL is a functional language which has been targeted at a wide variety of systems including current generation multiprocessors such as the Encore Multimax and research dataflow machines[2][3][4]. The textual form of SISAL, in terms of control structures and array representations, provides a relatively easy transition for those familiar with imperative languages, since it has a PASCAL like syntax. The advantage of SISAL is that codes written in SISAL are portable to a variety of parallel architectures. SISAL also insulates programmers from the underlying machine architecture, and allows concurrency to expressed implicitly, thus removing the burden of processor synchronization and job scheduling.

4. Parallel Implementation

4.1 EPF

The simplest parallel implementation of the Burg algorithm is obtained by coding the sequential algorithm in FORTRAN and then using the Encore Parallel FORTRAN compiler (EPF) to produce the parallel code suitable for the Encore Multimax Multiprocessor. This process requires that the programmer knows very little about the underlying architecture of the machine, thus code may be generated very easily. This method is also attractive since it allows existing software, written in FORTRAN, to be implemented on parallel machines without any translation. There are two main disadvantages of this method. Firstly the optimum speedup is usually not obtained, since the original program may be sequential in nature. Secondly the annotated code produced by the EPF compiler is machine dependent.

The annotator has identified that all loops, in figure 2a, except the outer loop can be parallelised. Thus the annotator can successfully identify the parallel loops. The maximally parallel graph suggests that tasks $T_{in}(2)$, $T_{nn}(2)$ and $T_{in}(3)$ could be performed at the same time; unfortunately EPF can only slice loops, and since the outer loop is sequential, due to task $T_n(1)$, EPF cannot make these tasks parallel.

4.2 SISAL

The second implementation of the algorithm is in SISAL. The "disadvantage" here is that existing codes need to be re-expressed in the SISAL language, to expose the possible parallelism. The SISAL implementation of the Burg Algorithm as presented in [1], was directly transliterated from the FORTRAN version, excepting the loop which updates the autoregressive coefficients shown in figure 3a, which was transformed into a parallel format to ease the coding.

```
%calculate auto regressive coefficients

a:=  for k in 1,old n
 returns array of
      if k = old n then c
      else old a[k]+c*old a[old n-k]
      end if
 end for;
```

Figure 3a Implementation of the calculation of the autoregressive coefficients in SISAL.

The loop which updates the forward and backward errors was also changed, figure 3b. The original FORTRAN loop has been split into two loops. This was done so that the indexes of **b** change in manner which is suitable for the parallel *for* loop.

```
e:=
 for l in old n+1,m
 returns array of
     old e[l]+c* old b[l-old n]
 end for;
b:=
 for j in 1,m - old n
 returns array of
      old b[j]+c*old e[j+old n]
 end for;
```

Figure 3b Implementation of the updating of the forward and backward errors, in SISAL.

SISAL expresses concurrency naturally, therefore it is not possible to write a sequential loops in the parallel form. In order to achieve useful speedup, it is necessary to reorganise the computation, and rethink the algorithm in a parallel manner.

5. Results

The SISAL and FORTRAN versions of the program were run on both an Encore Multimax and IBM RS6000/530 system using the standard f77 FORTRAN compiler, the EPF compiler, where appropriate, and the optimising SISAL compiler (OSC v12.7).

For comparison purposes the number of data points was set to **m**=10000 and the model size to **max**=100. This model size was chosen so as to obtain run times which could be measured accurately.
The run times obtained for both the FORTRAN and SISAL implementations of the algorithm on the Encore Multimax multiprocessor with six XPC processors are summarised in table 1.

Note that $\text{Speedup} = \frac{T_{1\ \text{proc}}}{T_{n\ \text{procs}}}$ and $\text{Efficiency} = \frac{\text{Speedup(n)}}{n}$.

Processors	Time (s)	Speedup	Efficiency
1	29.6	1.00	1.00
2	17.2	1.72	0.86
3	12.7	2.33	0.78
4	9.7	3.05	0.76
5	8.2	3.61	0.72
6	7.8	3.79	0.63

Encore Parallel FORTRAN

Processors	Time (s)	Speedup	Efficiency
1	31.02	1.00	1.00
2	15.49	2.00	1.00
3	13.46	2.30	0.77
4	11.03	2.81	0.70
5	9.43	3.29	0.66
6	8.32	3.73	0.62

SISAL

Table 1 Experimental results on the Encore Multimax

As can be seen from the run times, speedup is achieved with the EPF compiler without significant programming effort. The EPF compiler converts DO LOOPS to parallel code. However, in some cases the annotator is fairly conservative, and further speedup may be obtained, in some instances, by manually annotating programs. In this study no manual annotation was performed as the speedup obtained for this simple code was satisfactory.

As can be seen, from the results, the FORTRAN and SISAL implementations achieve similar speedup, but the FORTRAN implementation has a lower run time for the single processor run, refer to table 1 and figure 4a.

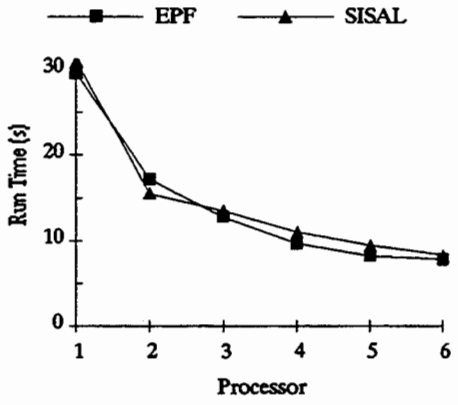

Figure 4a Run time vs processors

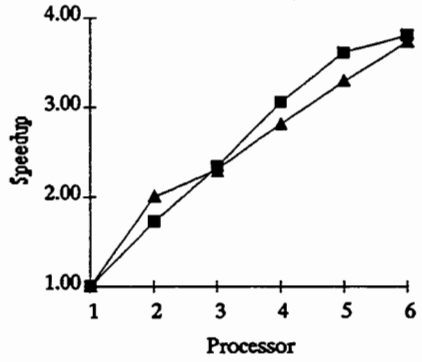

Figure 4b Speedup vs processors

It should be noted that initially no speedup was achieved with the SISAL implementation of the Burg Algorithm. Speedup was achieved by forcing the SISAL compiler to slice all **for** loops, by setting the **-h** pragma to 500; the cost estimator in SISAL had deemed the low complexity loops not worth slicing. The value of 500 was arrived at by trial and error. As this pragma is applied globally in the current version of the SISAL compiler there may be a risk of over parallelisation of some loops [6].

Speedup for the SISAL implementation is dependant on the size of the model, as can be seen from the graph shown in figure 5. Speedup increases as the number of data points increases, this is because for larger amounts of data, the processors spend relatively more time on useful computation, than on overheads computation.

The droop in the speedup curve, of figure 5, for six processors is due to other processes competing for the limited machine resources.

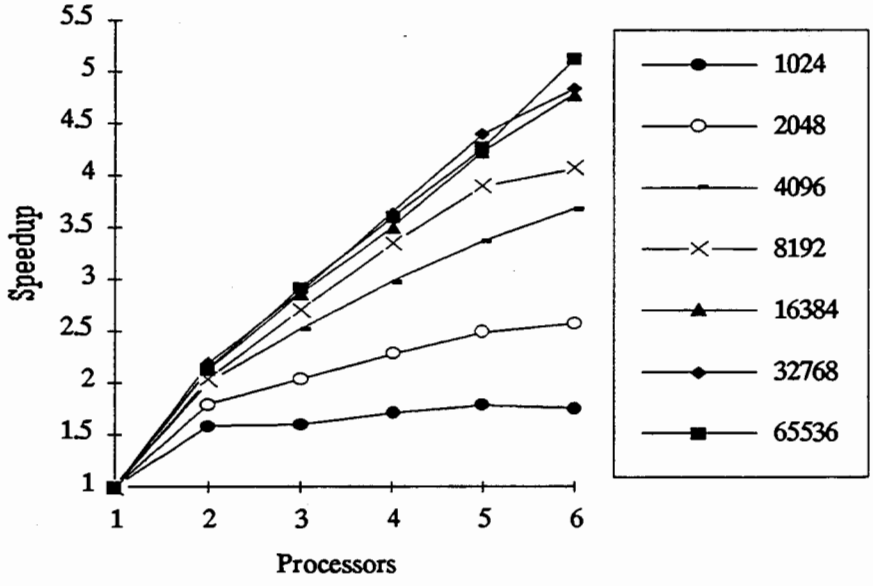

Figure 5 Speedup vs number of data points.

Speedup for the SISAL implementation increases beyond six processors as can be seen from the speedup curve shown in figure 6. These results were obtained on a slower 20 APC processor Encore Multimax. Only 16 processors were used so that interference from other users was minimised.

The graph shows the effect of Amdahl's law that forces the tail of the speedup curve to flatten. This limitation is due to the sequential part of the algorithm but good speedup is still achieved. The parallel FORTRAN (EPF) compiler was not available on this system.

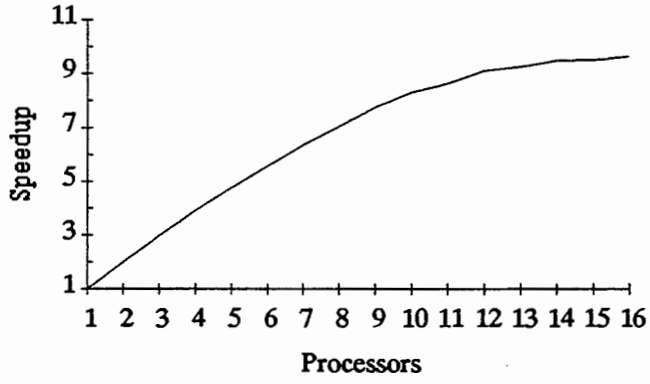

Figure 6 Speedup vs processors

The run times for a single processor (RS6000) machine are summarised in table 2. These times are for a model size of **m**=10000 and **max**=100. The time for the SISAL implementation is comparable with the FORTRAN[1] implementation.

	User +System (s)
SISAL	1.05
FORTRAN	4.96
FORTRAN -O	0.97

Table 2 Times for the IBM RS6000/530 (m=10000, max=100)

The results for the IBM RS6000/530 and Cray Y-MP are shown in table 3 for comparison with the results from [1], these results were obtained by tailoring the algorithm to the architecture of the target machine. The parameters for this study were **m**=16384 data points and model size **max**=10.

Machine	*HEP*	*iPSC/2*	*MPP*	*X-MP/48*	RS6000	Y-MP
Execution Time (s)	*1.679*	*0.24*	*0.5522*	*0.016887*	0.19	0.009 (1p)

Times from [1]

Table 3 Comparison of Burg Algorithm execution time (m=16384, max=10). Note the times for the Y-MP and RS6000 are for SISAL implementations.

6. Conclusions

The Burg filter was implemented both in FORTRAN and SISAL. Significant speedup is achieved with the SISAL implementation, suggesting that SISAL may be a useful language for signal processing algorithms. SISAL is useful since it allows parallelism to be expressed without considering processor synchronization and the underlying machine architecture. FORTRAN annotators such as the EPF annotator are useful in that speedup is obtained for little effort, and existing FORTRAN implementations of some simple algorithms need not be recoded, this is desirable since several digital signal processing FORTRAN programs already exist. Run times on modern single processor machines such as the IBM RS6000/530, are comparable to some existing parallel architecture machines, and give an indication of possible computation speeds of future generation multiprocessors.

1 XLF RS6000 FORTRAN Compiler

Acknowledgements

The authors thank the members of the Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology, in particular P.S. Chang, for their assistance in this research.

The authors thank Cray Research Australia for the use of the Cray Y-MP 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 both the EPF and SISAL codes for the Burg algorithm are available from the principal author.

References

[1] N.M. Sammur and M.T. Hagan. "Mapping Signal Processing Algorithms on Parallel Architectures." Journal of Parallel and Distributed Computing, Issue no.8 1990 pp180-185.

[2] McGraw et al., "SISAL: Streams and Iteration in a Single Assignment Language." Language Reference Manual, M146, Lawrence Livermore National Laboratories.

[3] A.P.W. (Wim) Bohm and J. Sargeant, "Efficient Dataflow Code Generation of SISAL", Technical Report UMCS-85-10-2, Department of Computer Science, University of Manchester, 1985.

[4] G.K. Egan, N.J. Webb and A.P.W. (Wim) Bohm, "Some Features of the CSIRAC II Dataflow Machine Architecture", in Advanced Topics in Data-Flow Computing, Prentice-Hall 1990,

[5] D.C. Cann, "High Performance Parallel Applicative Computation", Technical Report CS-89-104, Colorado State University, Feb.1989.

[6] P.S. Chang, and G.K. Egan "Analysis of a Parallel Implementation of a Numerical Weather Model in the Functional Language SISAL" Technical Report 31-012, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, March 1990.

[7] R.A. Roberts and C.T. Mullis, "Digital Signal Processing" Addison - Wesley 1987

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