LA-UR 92-805

LA-UR--92-805

DE92 011372

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: **ON THE ABSTRACTED DATAFLOW COMPLEXITY OF FAST FOURIER TRANSFORMS**

AUTHOR(S): Anton P. W. Bohm
Robert E. Hiromoto
Kathleen A. Kelly
J. M. Ashley

SUBMITTED TO: Sixth ACM International Conference on Supercomputing
Washington, D. C.
July 1992

DISCLAIMER

Los Alamos National Laboratory
Los Alamos New Mexico 87545

On the Abstracted Dataflow Complexity of Fast Fourier Transforms

A.P.W. Böhm
Computer Science Department
Colorado State University

R.E. Hiromoto and K.A. Kelly
Computer Research and Applications
Los Alamos National Laboratory

J.M. Ashley
Department of Computer Science
Indiana University

1 Introduction

The notion of a high level parallel programming language that automatically extracts the available parallelism of a program has been of research interest to a number of functional language groups around the world. The development and study of functional languages have been justified not only as a means to increase the programmer's productivity but also to increase computational performance by the extraction and execution of fine grain parallelism. The advantage of implicitly expressing parallelism precludes the need for encoding explicit parallel constructs and eliminates the hazards of introducing subtle and unpredictable synchronization errors.

Although parallel architectures have been studied and the properties of prototype systems embodied only as paper machines, significant development

with in the last few years may now allow us to realistically examine the claims made by the functional language community. Currently functional language and compilation research has resulted in several very powerful, inherently parallel programming languages. Notable among these development is Id, the work of Arvind's Computation Structures Group at MIT [11]. Based on the dataflow computational model, Id has a functional and deterministic subset, yet is a completely general purpose language supporting synchronizing data structures, and side-effects. Few fine grain dataflow computer systems have been developed in the past. One of the first research prototypes was designed and built at the University of Manchester [5]. This project resulted in identifying many important architectural issues in the design of support hardware for the dataflow execution model. The Manchester group used the purely functional, single assignment language SISAL [9] and produced a SISAL compiler generating efficient dataflow code [3].

A more ambitious purely fine grain dataflow project has resulted in the Sigma 1, a large dataflow system with 128 processing elements built at the Electro Technical Laboratory (ETL) in Tsukuba, Japan [15]. From this and other dataflow experiences, a multithreaded execution model within the frame work of dataflow has emerged and has many researchers very hopeful of its success. Today several research groups are seriously involved with building prototypes of these multithreaded architectures such as ETL's EM 4[13] and EM 5[14], Motorola and MIT's Monsoon [17] and Sandia's Epsilon 2 [7].

With dataflow languages, compiler technology and hardware quickly maturing and becoming available for use, the opportunity to critically evaluate the advantages claimed by functional language and dataflow advocates now appears to be a feasible task.

[illegible] codes for the Fast Fourier Transform written in Id and targeted for execu-

tion on Motorola's dataflow machine Monsoon. The FFT application is of interest because of its computational parallelism, its requirement for global communications, and its array element data dependences. We use the parallel profiling simulator Id World to study the dataflow performance of various implementations. Our approach is *comparative*. We study two approaches, a recursive and an iterative one, and in each version we examine the effect of a variety of implementations. We contend that only through such *comparative* evaluations can significant insight be gained in understanding the computational and structural details of functional algorithms.

2 Parallel Complexity Measures

The attractiveness of the dataflow model of computation is that all forms of parallelism can be expressed in it (as opposed to say SIMD). This makes dataflow an ideal environment for the analysis of parallel algorithms, their sequential threads, and their resource requirements [8,16]. Initial evaluation of a dataflow program is performed using the *Id World* simulator which collects statistics while it executes the code [10]. Simulation proceeds in discrete *time steps* during which all enabled instructions (instructions for which all data is available) are executed. It is assumed that every instruction executes and sends its resulting data to its successor instructions in exactly one time step: Id world simulates the behaviour of the "Tagged Token Dataflow Architecture" (TTDA) [2]. Two time related measurements are recorded: the *total work* S_1 is the total number of instructions executed; and the *critical path length* S_∞ is the total number of time steps required. The parallelism of the program is displayed in an *(ideal) parallelism profile* by plotting the number of executed instructions at each time step. A third measure S_p gives the total number of time steps if at most p instructions [illegible] *allelism profiles*. Other measures are space related and are also recorded per

time step: the total number of tokens waiting to be matched, and the total amount of data structure storage required. Important machine level phenomena such as the effect that global communication time may have on the computation are not addressed when executing programs on the emulator.

In this paper we will restrict ourselves to the time related measures S_t, S_∞, and S_p.

3 The Recursive Fast Fourier Transform

Figure 1 shows a first version of a recursive formulation of FFT's, [illegible] from the mathematical definition.

```
[illegible] =
  (_,SizeV) = bounds V  in
 if (SizeV == 1) then V  else
 {(OddV,EvenV) = shuffle V ;
  fftO = fft OddV ; fftE = fft EvenV ;
  Mid   = SizeV / 2 ; X  = TwoPi / SizeV ;
  Coeff = {array (1, Mid) |
           [i] = Cmplx (cos(X*(i-1))) (-sin (X*(i-1))) ||
                 i <- 1 to Mid };
  Prod  = { array (1, Mid)|
           [i] = Cmplx_Mul Coeff[i] fftE[i]            ||
                 i <- 1 to Mid }
 }  in  %butterfly
 {array (1, SizeV)
   | [i] = Cmplx_Add fftO[i] Prod[i] ||
           i <- 1 to Mid
   [illegible] ||
           i <- 1 to Mid };
```

```
};

def shuffle V =
{(_, SizeV) = bounds V ; Mid = SizeV / 2  in
  ({ array (1, Mid) | [i] = V[(i*2)-1] || i <- 1 to Mid },
   { array (1, Mid) | [i] = V[i*2] || i <- 1 to Mid })
}
```

Fig. 1. Recursive FFT.

The only Id feature that needs some explanation is *array comprehension*:

array ‹ *bounds* › ‹ *target* › ‹ *expression* › ‹ *generator* ›

In *bounds* the extent of each dimension of the resulting array are declared. Each *target*, *expression*, *generator* triple creates a loop, *for generator array*[*target*] = *expression*.

Note that in *fft* recursive invocations are applied to the odd and even elements of V until the size of V is one. As shown in Fig. 2, the data dependencies occurring in the recombination of smaller results into larger ones form "butterfly" patterns. In the program text in Fig. 1, the definitions of the arrays *Coeff*, *Prod*, and the result of *fft* implement this recombination.

In the recursive algorithm, $O(SizeV)$ arrays are allocated, 2 of size $SizeV$, 4 of size $SizeV/2$, 8 of size $SizeV/4$, etc. The total size of the allocated arrays is $O(SizeV\,(log(SizeV)))$, because there are $O(log(SizeV))$ stages each allocating k arrays of size $SizeV/k$. For each array element that is allocated, there is a constant amount of work to calculate its value, so there is $O(SizeV\,(log(SizeV)))$ total work. Figure 2 suggests a critical path length of $O(log(SizeV))$ and no sequential stretches.

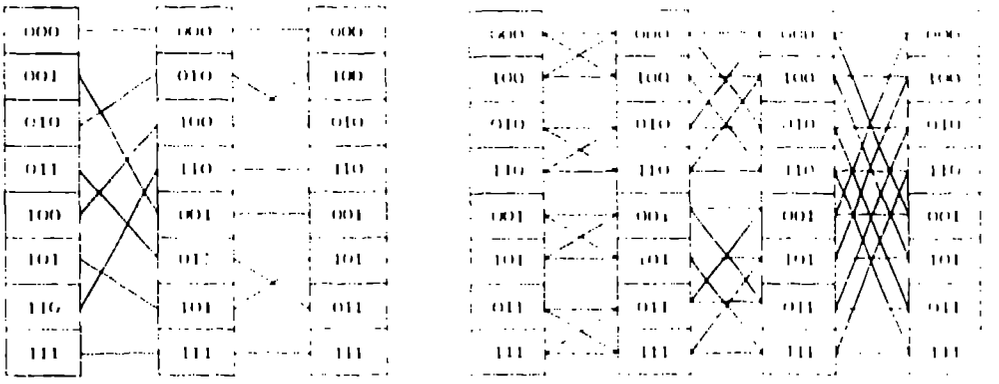

Fig. 2. Data dependence of FFT.

The above measures when applied to steps 2 and 3 provide an *Abstracted Complexity* of the algorithm that is free from the issues concerning language implementations, computational models, and the architecture. This is an idealized computational complexity model that will be used as a yard stick for the actual complexity measures obtained from running the program. This method is useful in incrementally evaluating the mapping from the language to the computational model which in turn is mapped to the machine architecture.

The parallelism profile seen in Fig. 3 is the first incremental step of running the above program under the Id world simulator with *Size*(V) = 128.

The profile is far from what is expected from the abstracted complexity measures defined above. Observe that first there is explosive divide and conquer parallelism (A) peaking at 1900, followed by (B) a stretch of low parallelism of about 20. A second less significant burst of parallelism (C) [illegible] problems the two sequential stretches (B and D) are observed to dominate more

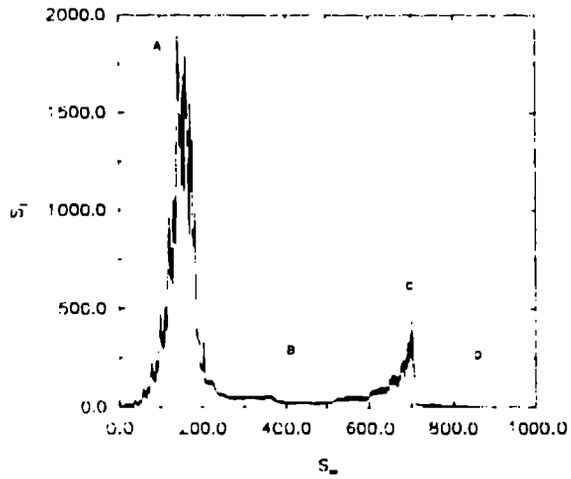

Fig. 3. Ideal parallelism profile for unoptimized recursive FFT, Size-V = 128.

and more. This parallelism profile is disappointing since the computational parallelism is known to be very large. We know that the FFT program take $O(log(Size\,V))$ parallel steps to unfold all *fft* and *shuffle* functions, which accounts for the first burst (A) of the divide and conquer parallelism. Once the functions have been unfolded, the loops in the array comprehensions dictate the parallelism and consequently the speed of the computation. Therefore, in order to understand and improve the program behavior we need to study the dynamics of loops.

4 Analysis of Loops and Double Recursion

Consider the following functions and their respective simulated performance found in Table 1.

```
def w n = {s=0 in {while s<n do next s=s+1; finally s};
def ww m n = {s=0; r=0 in {while s<m do next s=s+1;
```

```
                                    next r=r+w n finally r}};
def a n = {array (1,n) | [i] = i || i <- 1 to n};
def d n = if (n == 1) then 1 else d (n/2) + d (n/2);
```

function	m	n	S_1	S_∞
w		1	27	16
		2	41	21
		3	41	26
ww	1	10	140	80
	2	10	251	85
	3	10	362	90
	4	20	250	130
	2	20	391	135
	3	20	572	140
		50	923	691
		100	1,703	1,341
		200	3,291	2,641
		256	2,563	1,891
d		1	13	7
		2	56	19
		4	112	31
		8	314	43
		16	658	55

Table 1: Behavior of loops and recursion

Analysis of S_1 and S_∞ for these programs brings us to the following observation. In the mapping from Id to TDDA, it takes 5 steps along the critical path to spawn a loop, no matter whether it is an inner loop, an outer loop, or a loop in an array comprehension. We call the number of steps along the critical path to spawn a loop the *loop rate*. The loop rate plays an important role in the parallelism of a program. A high loop rate decreases the parallelism of a program. Take *ww* as an example. The inner loop is almost sequential and its critical path length can be varied by varying

inner loops are skewed on top of each other as in Fig. 4. The number of

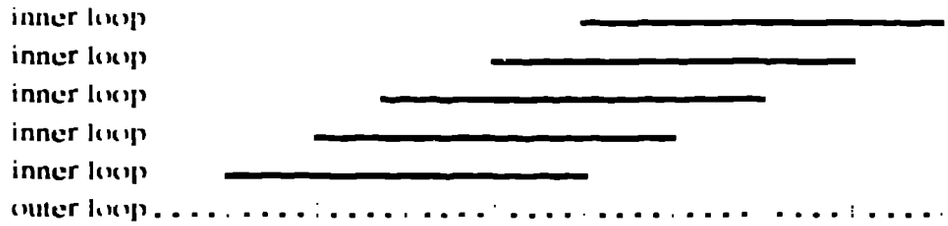

Fig. 4. Effect of loop rate on parallelism.

inner loops that run in parallel is $S_{\cdot}(\mathit{innerloop})\cdot\mathit{looprate}$. Note that this is independent of the total number of inner loops.

Divide and conquer programs do not suffer from this slowing effect, as exemplified by the function d, where $S_{\cdot}$ grows logarithmically with n and the parallelism is $O(n)$.

We are now in a position to explain the parallelism profile in Fig. 3. While the dynamic call tree unrolls along the lines of Fig. 2 in $O(\log(Size\,V))$ time (phase A), the ***shuffle***s in the first ***fft*** produce one element every ***loop rate*** time steps. This results in a producer/consumer mismatch. That is, the array elements are not all available at the moment the dynamic call tree is ready to manipulate them. The elements are put in place during the $O(\log(Size\,V))$ shuffle stages without data dependence problems. In the $O(\log(Size\,V))$ butterfly stages, the elements must wait to combined with their corresponding elements that are not yet available. Phase B starts after the call tree is unrolled, and the program behaves very much like the *inr* function with an inner loop of length $O(\log(Size\,V))$. In this case the "inner loop" is spread over a number of butterfly stages. The parallelism in phase B is therefore $O(\log(Size\,V))$ instead of the expected $O(Size\,V)$. Phase B

stages in the butterfly can now be done in divide and conquer fashion (phase

function	n	S_1	S_∞
ab	16	277	118
	32	501	123
	64	978	133
	256	3,682	198

Table 2: S_1 and S_∞ of a strip-mined loop

C1. The sequential tail (phase D) is caused by the array comprehension in *fft* that generates the final array.

The solution to this problem is to spawn loops fast enough so that they don't cause unnecessary delay. This has been recognized by a number of dataflow groups, who invented special instructions to map scalar to vector instructions, to rapidly create parallel workload, especially in loops. Examples of these instructions are iterative instructions in the Manchester Dataflow Machine [3] and the CSIRAC II Dataflow Machine [6], and the repeat mechanism in the Epsilon-2 machine [7]. Neither the tagged token dataflow machine nor the Monsoon machine [17] have this type of instructions. This was a design decision based on a RISC argument that these instructions cause pipeline bubbles [3]. Still the loop delay problem can be address by optimizing compilation, such as loop unrolling. Therefore, it is still possible, although at a higher cost in terms of instruction counts, to create array elements at a higher rate. As stated, this should and can best be done by an optimizing compiler. However, as this optimization is not available in the current Id compiler, we resort to a rather inelegant programming trick very similar to *strip-mining* [18] in a vector context. As an example, compare the function *a* in Table 1 to a strip mined version *ab* in Table 2 where the depth of the loop unrolling is defined by *chunk* = 16.

```
def ab n chunk = {array(1,n) | [i] = i ||
```

```
j <- 1 to n by chunk &
i <- j to j+(chunk-1)};
```

Where in the procedure *a* defined above, the array elements are created at the loop rate; in *ab* the elements are created at the rate of *chunk* per *loop* ... Because we have strip-mined by hand, this comes at the cost of a considerably higher S_1 value than necessary. When we apply strip-mining to all loops in the fft and shuffle functions program, the parallelism profile, as shown in Fig. 5, is satisfactory. The critical path length is now logarithmic, and the two sequential stretches B and D have disappeared.

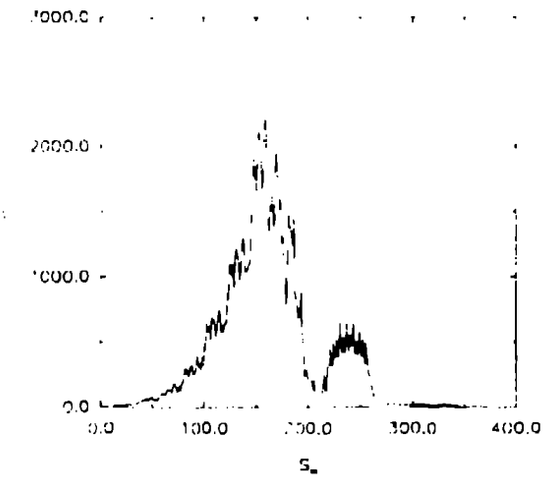

Fig. 5. Ideal parallelism profile for optimized recursive FFT with loop strip-mining, SizeV = 128.

Although the array comprehensions in fft and shuffle (Fig. 1) make for an elegant and functional programming style, they lack expressive power: it is impossible to derive two or more values in the expression part of the comprehension and assign these to two or more targets in the array. This ... In order to avoid recomputation of the operands of the butterfly combine

operation, we are forced to put intermediate results in array *Prod*. We can avoid doing this extra work by rewriting the butterfly part of *fft* using a loop instead of an array comprehension. A standard trick in the FFT algorithm is to table the roots of unity once in the main function calling *fft*. The following version of *fft* is strip mined with a chunk size 16, it performs the recombination in a loop instead of an array comprehension, and retrieves the roots of unity from a table *RofU*.

```
def fft V RofU =
( (_,SizeV) = bounds V; (_,OrgSize) =  bounds RofU in
 if (SizeV == 1) then V  else
 {(OddV,EvenV) = shuffle V ;
  [illegible]
  m = [illegible]
  R = 1D_I_array 1, SizeV) in            %butterfly
   [illegible]
   {for j <- 1 to m by 16 do    {for i <- j to j+15 do
    prod  =  Cmplx_Mul RofU[((i-1)*step)+1] fftE[i];
    R[i] =  Cmplx_Add fft0[i] prod;
    R[m+i] = Cmplx_Sub fft0[i] prod}
   finally R}
  else
   {for i <- 1 to m do
    prod  =  Cmplx_Mul RofU[((i-1)*step)+1] fftE[i];
    R[i] =  Cmplx_Add fft0[i] prod;
    R[m+i] = Cmplx_Sub fft0[i] prod
   finally R}
 } }
```

5 Strip Mining and Operator Strength Reduction

In terms of total work, often there are two extreme ways to compute a certain function: *parallel but costly* versus *sequential but cheap*. An example is the computation of the roots of unity. The parallel way is to compute each root independently using *sin* and *cos* functions. The sequential way is to derive a recurrence relation, expressing the (n+1) th root as a function of the n-th root, a technique called *operator strength reduction*. A data dependence is introduced but the resulting implementation is more efficient in terms of S_1. Ideally we would like to be able to balance the amount of parallelism against S_1, which can be achieved by strip mining: the outer loop uses the parallel data independent method to start off inner loops which use the recurrence relation. The following code computes the root of unity in this manner.

```
h = n/2;
theta = -TwoPi/ n;
RofU = 1D_I_array (1, h);
seed = cmplx(-2.0*((sin(0.5*theta))^2))
            ( sin theta);
  {for j <- 1 to h by chunk do
   { RofU[j] = cmplx (cos (-theta*(j-1)))
               (-sin (-theta*(j-1)))
   in
   {for i <- j+1 to j+chunk-1 do
   RofU[i] = Cmplx_Add (Cmplx_Mul seed RofU[i-1]) RofU[i-1]
    }}
  }
```

Again for Sr (V = 128, Fig. 6 shows the resulting parallelism profile.

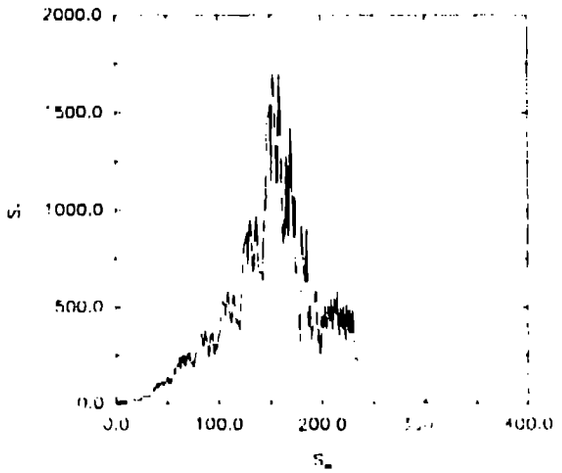

Fig. 6. Ideal parallelism profile for optimized recursive FFT with loop strip mining and operator strength reduction. Size = 128.

6 Limited Machine Parallelism

The ideal parallelism profiles in both Figs. 4 and 5 show a peak parallelism much higher than the parallelism available in say a 16 processor dataflow machine. Some of this excess parallelism is needed to hide latencies caused by remote memory references. Still, one could ask whether it is necessary to enhance the parallelism in these programs. We claim that this is the case because the *stretches of low not high parallelism govern the machine behavior*. This is nothing more than Amdahl's law. To exemplify this, Figs. 7 shows the limited parallelism profiles of the original recursive FFT algorithm (left) and the strip mined version (right) for *Size* = 128 and $p = 128$, respectively. Notice that the critical path (X axis) in the improved version is shorter than the unoptimized version. However, both critical path lengths do increase as the available parallelism is decreased (see Figs. 3 and

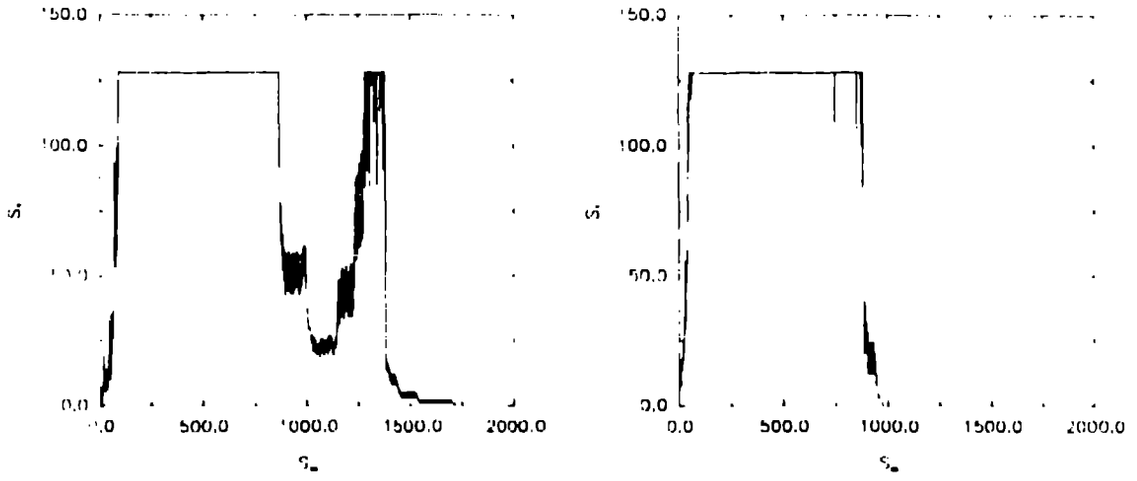

Fig. 7. 128 fold limited parallelism profile for unoptimized and optimized recursive FFT, SizeV = 128.

Also notice the serious dent in parallelism in in the left profile but which has all but disappeared in the improved program.

7 Iterative FFT

In the iterative FFT algorithm, bit reversal of the index, instead of repeated shuffling, puts the array elements in the required place. We make bit reversal parallel and efficient by creating a nested loop, where the outer loop provides the parallelism and the inner loop the efficiency. The outer loop starts a sequence by just reversing the bits of the index: $B[bitrev J] = A[J]$ and the inner loop uses a recurrence: $bitrev(J+1) = (bitrev J) +' 1$, where $+'$ increments a bit pattern going from left to right [12], as in the following function *itershuffle*.

```
def iter_shuffle A = {
    (_,n) = bounds A; J = 1; B = 1D_I_Array (1,n) in
```

```
    {for k <- 1 to n by chunk do {
     j = bitrev k n in
     {for i <- k to (k+chunk-1) do
       B[i] = A[j];
       next j = { m = div n 2 in
                  if ((m>=2) and (j>m))
                  then { while ((m >=2 ) and ( j>m )) do
                         next j = j-m; next m = div m 2
                       finally j+m }
                  else j+m }
     }}
    finally B}
}
```

The iterative FFT differs from the recursive one in another aspect: the intermediate values are kept in *log(Size)* equal sized arrays. The following program is, apart from the strip mining, an Id translation of the code in [12].

```
def fft A RofU = {
   (_,n) = bounds A; mmax = 2
     in
   { while n >= mmax do
      next A  =
      {h = div mmax 2;  k = div n mmax; B = 1D_I_Array(1,n)
       in
       if (h > chunk) then
        {for l <- 1 to h by chunk do
         {for ll <- l to l+chunk-1 do
         /index = (ll-1)*k+1 in
          {for jj <- 0 to (k-1) do
           i = ll+jj*mmax; j = i+h;
```

```
          temp =  Cmplx_Mul A[j] RofU[index];
          B[j]=  Cmplx_Sub A[i] temp;
          B[i]=  Cmplx_Add A[i] temp;
 }};
        } finally B }
      else
       { for ii <- 1 to h do
         {index = (ii-1)*k+1 in
         { for jj <- 0 to (k-1) do
    i = ii+jj*mmax; j = i+h;
          temp =  Cmplx_Mul A[j] RofU[index];
  B[j]= Cmplx_Sub A[i] temp;
  B[i]= Cmplx_Add A[i] temp;
 }} finally B }
       },
     next mmax = mmax * 2;
    finally A }
}
```

Table 3 gives S_1 and S_∞ values of the recursive and iterative FFT algorithms. The first column shows the figures for the initial recursive algorithm, the second and third columns show the results of the optimized recursive and iterative versions of the algorithm where the chunk sizes for strip mining have been optimally chosen. Surprisingly, there is no clear winner between the optimized cases. The optimized recursive algorithm has the shortest critical path, whereas the optimized iterative algorithm executes the smallest number of instructions. The instruction counts in this paper are those of the MIT Tagged Token Dataflow Architecture, the relative of

Method	recursive (unoptimized)		recursive		iterative	
SizeV	S_1	S_∞	S_1	S_∞	S_1	S_∞
16	10,800	226	10,048	202	9,072	251
32	25,588	410	22,613	262	20,159	293
64	59,060	557	50,046	298	44,070	352
128	133,796	1,044	109,716	330	96,066	502

Table 3: S_1 and S_∞ of FFT algorithms

8 Conclusion

We have introduced the notion of an *abstracted complexity* of a parallel algorithm and used this to evaluate the actual complexity of two FFT algorithms. By applying this comparative analysis, we were led to transform our initial FFT algorithms using well understood conventional techniques such as strip mining and operator strength reduction of loop behavior in Id. Amdahl's law states that the speedup behavior of a parallel program is governed by its *sequential* parts, not its parallel parts. This means that even if there is a lot of parallelism in a program, or if it is "intuitively clear" how to parallelize a program, as it is in the case of FFT, we need to analyze it, study its sequential threads, and learn how to remove these. The Id World tools make it possible to identify and measure the sequential threads of a program. The results presented in this paper indicate that important improvements in the parallelism and total work efficiency can be achieved by striving for an actual parallelism profile that is in accordance with the abstracted complexity of the parallel algorithm.

References

[1] Arvind, D.E. Culler and K. Ekanadham, "The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures," C.R. Jesshope and K.D. Reinartz (eds.), CONPAR 88, Cambridge University Press (1989) pp. 541-555.

[2] Arvind, R.A. Iannucci, *Instruction Set Definition for a Tagged Token Dataflow Machine* LCS, MIT, 1983

[3] A.P.W. Böhm, J.R. Gurd, *Iterative instructions in the Manchester Dataflow Computer*, IEEE, Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp. 129-139.

[4] A.P.W. Böhm, J. Sargeant, *Code Optimisation for Tagged Token Dataflow Machines*, IEEE Transactions on Computers, Vol 38, Num 1, January 1989.

[5] A.P.W. Böhm, J.R. Gurd, C. C. Kirkham, *The Manchester Dataflow Computing System*, in: J. ongarra (ed), *Experimental Parallel Computing Architectures*, North Holland, 1987, pp 177-219.

[6] G.K. Egan, N.J. Webb, A.P.W. Böhm, *Some Architectural Features of the CSIRAC II Dataflow Computer*, in J.L. Gaudiot and L. Bic, *Advanced Topics in Data-Flow Computing*, Prentice Hall, 1990.

[7] V. Grafe et al., *The Epsilon Project* in J.L. Gaudiot and L. Bic, *Advanced Topics in Data-Flow Computing*, Prentice Hall, 1990.

[8] J.R. Gurd, A.P.W. Böhm and Y.M. Teo, *Performance Issues in Dataflow Machines*, Future Generation Computer Systems, 3, 1987, pp. 285-297.

[9] J.R. McGraw *et al.*, SISAL – Streams and Iteration in a Single-Assignment Language, Lawrence Livermore National Laboratory, M-146 (January, 1985) 93 pp.

[10] D. R. Morais, *ID World: An Environment for the Development of Dataflow Programs Written in ID*, MIT LCS TR-365, may 1986.

[11] R.S. Nikhil, *Id (version 90.0)* Reference Manual, TR CSG Memo 284-1, MIT LCS 1990.

[12] W.H. Press et al., Numerical Recipes, the art of Scientific programming, Cambridge University Press.

[13] S. Sakai et al., "An Architecture of a Dataflow Single Chip Processor", Proceedings of the 1989 International Symposium on Computer Architecture, Filat, ACM (1989) pp 46-53.

[14] S. Sakai, Y. Kodoma, Y. Yamaguchi, *Architectural Design of a Parallel Supercomputer EM-5*, JSPP91, may 1991.

[15] T. Shimada *et al.*, "Evaluation of a Prototype Data Flow Processor of the SIGMA-1 for Scientific Computations", Proceedings 13th International Symposium on Computer Architecture (June, 1986) pp. 226-234.

[16] Y.M. Teo, A.P.W. Böhm, *Resource Management in Dataflow Computers with Iterative Instructions*, in J.L. Gaudiot and L. Bic, *Advanced Topics in Data-Flow Computing*, Prentice Hall, 1990.

[17] K. R. Traub, G. M. Papadopoulos, M. J. Be kerle, J.E. Hicks and J. Young, *Overview of the Monsoon Project*, ICCD91, IEEE, oct 1991, pp 150-155.

[18] M. Wolfe, *Optimizing Supercompilers for Supercomputer*, MIT Pres., 1989.