LA-UR -91-2440

LA-UR--91-2440

DE91 016300

TITLE

DEVELOPING DATAFLOW ALGORITHMS

AUTHOR(S) Robert E. Hiromoto
Anton P. W. Bohm

SUBMITTED TO International Association for Mathematics and
Computers in Simulation
July 22-26, 1991
Dublin, Ireland

MASTER

[illegible]

DEVELOPING DATAFLOW ALGORITHMS[1]

Robert E. Hiromoto
Computer Research & Applications
Los Alamos National Laboratory
Los Alamos, NM 87545 U.S.A.

AND

Anton P.W. Bohm
Computer Science Department
Colorado State University
Fort Collins, CO 80523 U.S.A.

INTRODUCTION

The design and validation of parallel algorithms is a rewarding and satisfying experience once the implementation has been completed and debugged. It is this latter task which can be extremely frustrating when dealing with a general purpose multiple instruction multiple data (MIMD) computer system. Errors in expressing parallel constructs give rise to unpredictable execution behavior, affecting both the resulting answer and the sanity of the programmer.

The notion of a high-level parallel programming language that insulates the programmer from the perils of asynchronous bugs and booby traps has been the goal of many researchers in the functional language community. In the last few years, significant progress has been made in this area. Language and compilation research has resulted in several very powerful, inherently parallel programming languages. Notable among these is Id, the work of Prof. Arvind's Computation Structures Group at MIT. Id has a functional and deterministic subset, yet is a completely general purpose language supporting synchronizing data structures, and side-effects. Compilation research is also being carried out at Yale University, Chalmers University of Technology, and the functional programming group at the University of Glasgow.

Up until now few dataflow computer systems have been developed for wide use. One of the first was designed and built at the University of Manchester. 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, strict, single assignment language SISAL for writing their applications and produced a compiler generating highly efficient dataflow code for this language. A similar dataflow project was initiated at the Electro-Technical Laboratory (ETL) in Tsukuba, Japan. Here a large dataflow system with 128 processing elements was designed and built. Unfortunately the lack of programming software has prevented the system from being fully tested on substantial application codes. Still from these and other experiences, a multithreaded execution model with in 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 (e.g., Sandia National Laboratories' (Albuquerque, New Mexico) Epsilon-2, Motorola and MIT's Monsoon, IBM's Empire, and ETL's EM-4).

We are now at a very exciting moment when language, compiler technology, and hardware are quickly maturing and becoming readily available for use. This moment also offers us the important opportunity to critically assess the advantages claimed by functional language and dataflow advocates.

Our approach is to study the performance of a collection of numerical algorithms written in Id which is available to users of Motorola's dataflow machine Monsoon. We will study the dataflow performance of these implementations first under the parallel profiling simulator Id World, and second in comparison with actual dataflow execution on the Motorola Monsoon. This approach will allow us to follow the computational and structural details of the parallel algorithms as implemented on dataflow systems. When running our programs on the Id World simulator we will examine the behaviour of algorithms at dataflow graph level, where each instruction takes one timestep and data becomes available at the next. This implies that important machine level phenomena such as the effect that global communication time may have on the computation are not addressed. These phenomena will be addressed when we run our programs on the Monsoon hardware. Potential ramifications for compilation techniques, functional programming style, and program efficiency are significant to this study. In a later stage of our research we will compare the efficiency of Id programs to programs written in other languages. This comparison will be of a rather qualitative nature as there are too many degrees of freedom in a language implementation (on language, compiler, and target machine level) for a quantitative comparison to be of interest.

We begin our study by examining four routines that exhibit different computational characteristics. These routines and their corresponding characteristics are listed below:

(1) **Fast Fourier Transforms** (FFT)
- *computational parallelism*
- *data dependences between the butterfly shuffles*

(2) **Adaptive Quadratures**
- *dynamic unrolling of recursively adaptive grid refinements*

(3) **Eigenvalue/vector Solvers**
- *application of cyclic rotations (incremental array updates)*

(4) **Stochastic Simulations**
- *data accumulation*

Details of our implementation and performance analysis will be presented during the session's presentation. For economy, we chose to describe only one of the four routines that have been analyzed at this time.

FFT

A fast fourier transform (FFT) routine was written in Id by J. Michael Ashley, a former summer student at Los Alamos National Laboratory. The FFT exhibits both divide and conquer and loop parallelism. The relevent Id program segments are:

```
Def Main_fft Size_of_V =
{ C = { array (1, Size_of_V) | [i] = Cmplx (i * 1.0) 0.0
                 || j <- 1 to Size_of_V } ;
In FFT C } ;

Def FFT V = { (_,Size_of_V) = bounds V ;
In {if (Size_of_V == 1) then V
  else
  {(Left_V, Right_V) = Shuffle V ; FFT_L = FFT Left_V ;
   FFT_R = FFT Right_V ; Mid   = fix (Size_of_V / 2) ;
   X    = Two_Pi / Size_of_V ;
   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] FFT_R[i]
        || i <- 1 to Mid } ; } ;
   In { array (1, Size_of_V)
     | [i] = Cmplx_Add FFT_L[i] Prod[i] || i <- 1 to Mid
     | [Mid+i] = Cmplx_Sub FFT_L[i] Prod[i] || i <- 1 to Mid } } },
```

[1]This work was done under the auspices of the U. S. Department of Energy contract No. W-7405-ENG-36.

```
Shuffle V = { ( _, Size_of_V) = bounds V ; Mid = fix (Size_of_V / 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 }) } ;
```

Note that in FFT a recursive invocation of FFT is applied to Left_V and Right_V (the odd and even elements of V, respectively) until the butterfly shuffle on V has been completed. The array element data dependences occurring in the recombination of smaller results into larger ones are expressed in three array comprehensions defining the values of the arrays Coeff, Prod, and the result of FFT. Running the above program under the Id world simulator for a Size_of_V of 128 gives us the following parallelism profile:

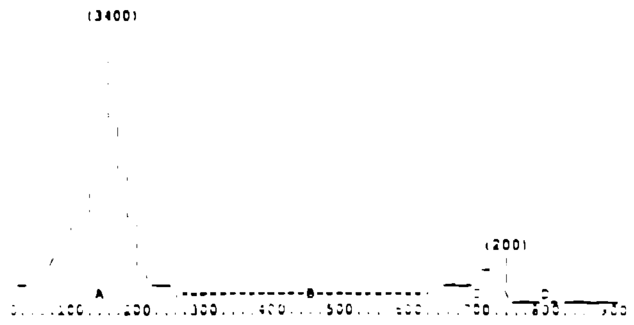

Fig. 1. Parallelism profile of FFT 128.

ANALYSIS

The simulator assumes that each instruction takes one timestep and that the results of an instruction execution are available the next timestep. This approximation leads to an idealised graph interpretation in which maximally parallel execution proceeds along a critical path via a sequence of indivisible timesteps. The graph in figure 1 plots the number of instructions executed at each timestep, and consequently pictures the ideal parallelism profile of FFT 128.

When studying Fig. 1, we observe the following. First there is explosive divide and conquer parallelism (A), followed by (B) a stretch of low parallelism. A second less significant burst of parallelism (C) follows which dies down to an almost sequential tail (D). For larger problems the two sequential stretches (B and D) are observed to dominate more and more. The parallelism profile drawn in Fig. 1 is very disappointing since the computational parallelism is known to be very large. To begin to understand this, we know that the FFT program takes O(log(n)) parallel steps to unfold all FFT and Shuffle functions. This accounts for the first burst (A) of the divide and conquer parallelism. Once the functions have been unfolded, the loops (array comprehensions) dictates the parallelism and consequently the speed of the computation. In the first instance the dominant loop is the array comprehension in the main function creating the original function values to be transformed.

Since a considerable amount of work may go into the activation of a loop-body, a loop analysis is performed on two simple loops:

```
Def WW n m = { s = 0;  r = 0;
In {while (s < n) do next s = s + 1; next r = r + W m; finally r} };
```

where

```
Def W n = { s = 0;
In {while (s < n) do next s = s + 1; finally s} };
```

Through an analysis of these two loops, it is found that WW (a doubly nested loop) requires five (5) steps on the critical path to instantiate each inner loop-body, that is, every five parallel steps a new inner loop is spawned off. As the inner loop bodies are skewed on top of each other, the number of them that run in parallel is equal to the critical path required to execute an inner loop divided by the rate at which the loop can spawn off the next inner loop. Figure 2 sketches a parallelism profile for WW 64 20:

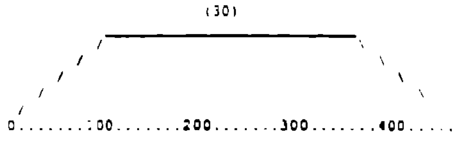

Fig. 2. Parallelism Profile for a nested loop.

Clearly, the loop rate plays an important role in the parallelism of a program. Phase B in Fig. 1 show this loop behavior where the dominant loop is the array comprehension in the main function creating the original function values to be transformed. Once every 5 parallel steps an array element is created and sent through the log(n) stages of the FFT.

The completion of the FFT is dependent on the loop rate and the creation of the last two elements of the original array. When the last two elements go into the FFT (phase (C) in Fig. 1), the remaining stages of the computations can be done with divide and conquer parallelism.

The last sequential tail (phase (D) figure 1) is caused by the array comprehension in FFT that generates the resulting array.

SOLUTION

The solution to this problem is to unroll loops fast enough so that they don't cause unnecessary delay. This has been recognized by a number of dataflow teams, who invented special instructions (very similar to vector instructions) to rapidly create parallel workload especially in loops (iterative instructions, repeat mechanism). Id does not have this type of instructions, but it is still possible, although at a consirably higher cost, to create array elements at a higher rate in a nested loop. Take the following array comprehensions:

```
Def A n  = { array (1, n) | [i] = i || i <- 1 to n };

Def Ab n = { array (1, n) | [i] = i || j <- 1 to n by 16 & i <- j to j+15 };
```

Some statistics:

A	n	S1	Sinf	Ab	n	S1	Sinf
	16	223	101		16	277	118
	32	399	181		32	504	123
	64	751	341		64	958	133
	256	2863	1301		256	3682	198

In the above table, S1 stands for the number of instructions executed and Sinf stands for the total critical pathlength. Where array elements are created at the loop rate (5) in A, they are created at a rate of 16 elements in 5 parallel steps in Ab (i.e., an element production rate of 16/5). This comes at the cost of higher S1 figures. This trick can be employed to make all loops in the FFT program go at a higher element production rate. Applying this idea to Main_fft, we have:

```
Def Main_fft Size_of_V =
{ C = { array (1, Size_of_V) | [i] = Cmplx (i * 1.0) 0.0
                        || j <- 1 to size by 16 & i <- j to j+15 } ;
In FFT C } ;
```

To apply this technique to FFT, requires keeping track of the shrinking vector lengths. The application of this technique to the array Coeff in FFT results in:

```
if Mid > 16 then
{ Coeff = { array(1, Mid)
        | [i] = Cmplx (cos (X * (i-1))) (-sin (X * (i-1)))
        || j <- 1 to Mid by 16 & i <- j to j+15} ;
  In { array (1, Size_of_V)
        | [i]  = Add_c FFT_L[i] Prod[i]
        || j <- 1 to Mid by 16 & i <- j to j+15 } }
else
{ Coeff   = { array(1, Mid)
        | [i] = Cmplx (cos (X * (i-1))) (-sin (X * (i-1)))
        || i <- 1 to Mid } ;
In { array (1, Size_of_V)
        | [i]  = Add_c FFt_L[i] Prod[i] || i <- 1 to Mid } } }
```

Its parallelism profile is shown in Fig. 3.

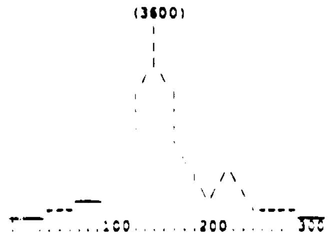

Fig. 3. Parallelism of unrolled loops in FFT 128.

The critical pathlength is about 30 percent of that of the original program, and the two ugly sequential stretches B and D have disappeared. Now the parallelism of the program is satisfactory.

It turns out that the array comprehensions are rather inefficiently implemented in Id. In the talk we will address the efficiency issue in some detail.