The Scalability of Linear Filters on Hypercube Concurrent Computers

C. H. Dick
Department of Electronic Engineering
La Trobe University
Victoria, 3083, Australia

Abstract

The general concept in solving a problem on a parallel machine requires subdividing a large problem into processes, each of which can be computed on a single processor. This can be a formidable task requiring substantial communication and control overhead. It is easy to speculate what may happen when one implements various applications on such a system. But nothing teaches one as surely as actually implementing real problems with real software on real hardware. This paper reports on the implementation and performance of a hypercube arrangement of VLSI DSP processors (TMS320C30) for FIR filtering. Two approaches are considered. A direct partitioning of the standard FIR filter structure, and a decomposition of the fast FIR algorithm described in [1]. Two interprocessor communication strategies are described. Models constructed from measurements on a prototype machine are used to calculate the performance of filters on higher dimensional hypercubes.

1 Introduction

Our aim in this paper is to provide an analysis of the use of hypercube arrangements of VLSI DSP processors for real-time FIR filtering. The arithmetic core of VLSI DSP processors is optimised for multiply-accumulate operations, therefore we consider parallelisations of FIR filter structures that retain the multiply-accumulate structure. Two approaches to parallelising the filter are considered. The first method is a direct decomposition of the standard FIR filter structure. The second technique utilises the fast FIR filtering algorithm described in [1].

This paper is organised as follows. Section 2 gives an overview of the hypercube concurrent computer developed in the Signal Processing Laboratory at La Trobe University. Section 3 describes the parallelisation of the standard FIR filter structure and the fast FIR filter algorithm described in [1]. The arithmetic and communication requirements of the algorithms are described. Based on measurements on our prototype hypercube, the sample rate and speedup of filters on higher dimensional hypercubes is presented.

2 System Architecture

DspCube is a hypercube arrangement of TMS320C30 based processing nodes. Nodes communicate over 80 Mbit/s duplex serial links, although the *real* data communication bandwidth is lower due to the link communication protocol and message handling overheads. One node is connected to a host system. Each node has 64 KBytes of fast memory on each of the two TMS320C30 busses, and can support upto 32 MBytes of DRAM. A server process running on the host provides DspCube with disk I/O and graphics facilities. The processing nodes communicate by passing messages. Each processor executes its own instruction stream and operates on a sub-set of data of the overall problem. Figure 2 shows the DspCube system architecture.

3 Parallel FIR Filter

A length N FIR filter is described by the difference equation

$$y_n = \sum_{i=0}^{N-1} x_{n-i} h_i \quad n = 0, 1, \ldots \tag{1}$$

where the y_n's are the filter outputs, the x_i are the filter input samples and h_i $i = 0, \ldots, N-1$ are the filter coefficients.

1058-6393/95 $4.00 © 1995 IEEE

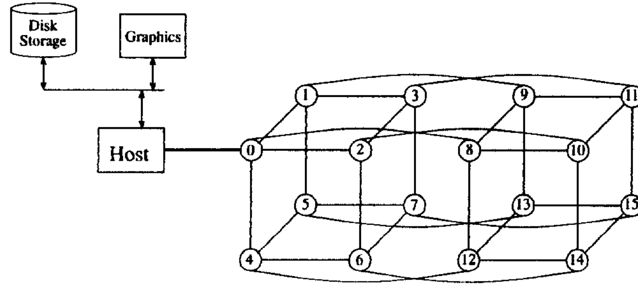

Figure 1: The architecture of DspCube. The current processing node hardware supports implementation of a 4-dimensional hypercube. The processing nodes are based on the TMS320C30 processor.

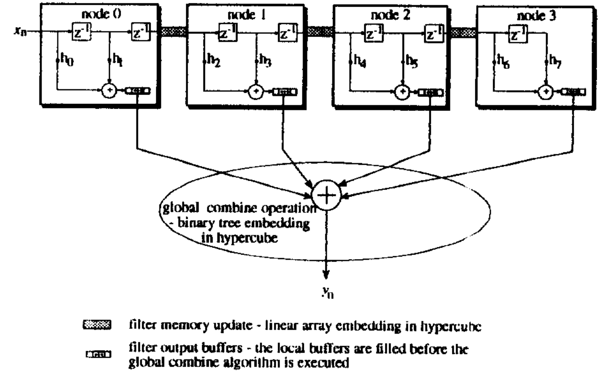

Figure 2: Partitioning a length 8 FIR filter for a 2 dimensional hypercube.

The FIR filter computation can be partitioned for a hypercube by decomposing the inner-product calculation into a number of sub-filter sections whose outputs are combined to form the final filter output. Consider a hypercube processor with M processing nodes, and assume that the number of filter coefficients is divisible with no remainder by the number of processing nodes. The extension to the case where this assumption is not true is relatively straightforward. Partition Eq.(1) into M sub-filter sections. The kth sub-filter is given by

$$y_n^k = \sum_{i=k\frac{n}{M}}^{(k+1)\frac{n}{M}-1} x_{n-i}h_i \qquad (2)$$

The final filter output y_n is the sum of the sub-filter partitions

$$y_n = \sum_{k=0}^{M-1} y_n^k \qquad (3)$$

Each sub-filter is allocated to one node of the hypercube. An example of computing a length 8 filter on a 2 dimensional hypercube is shown in Figure(2).

Computation of each sub-filter is performed in parallel with the final result obtained by combining each partial result. Combining the partial results requires inter-processor communication, as does the filter memory update required after each new filter output is formed. The extensive graph embedding properties [2] of the hypercube graph make this architecture well suited to implementing the required communication. A linear array embedding is employed for the filter memory updates, and a mesh of trees embedding is used in the global combine operation needed to sum the sub-filter outputs. It is not necessary to accumulate the final filter output in each node in the system, only in the node that will output the sample to the DAC. A modified version of the *combine* algorithm described in [3] is used for summing the sub-filter outputs. Assume that the sum of the sub-filter outputs is to be accumulated in node 0. First all nodes communicate across channel 0, the node with lowest node number receiving data, and the node with the higher node number sending data. The node receiving data adds the received sub-filter output to its local sub-filter output. This process is then iterated over all dimensions of the hypercube. Figure (3) illustrates this process for a 3 dimensional hypercube.

The baseline used for evaluating the performance of the parallel FIR filter is the fastest FIR filter that executes on one node of the parallel machine. The parallel FIR filter was implemented on 2 and 3 dimensional hypercubes. The execution times are listed in Table (1). The definition for speedup S used in this analysis is that described in [3] and defined as

$$S = \frac{T(1)}{T(M)} \qquad (4)$$

where $T(1)$ is the calculation time on a single node of the concurrent processor and $T(M)$ is the execution time for the calculation on an M node processor. The concurrent efficiency ϵ of the parallel algorithm is defined as

$$\epsilon = \frac{S}{M} \qquad (5)$$

The FIR filter parallelises well for a hypercube. However, for small filter orders, the partitioning of the inner-product calculation into M smaller inner-products that can be computed concurrently is dominated by the time to perform the global combine operation. Hence, a low efficiency is attained. The FIR filter calculation on a single processor is most efficient

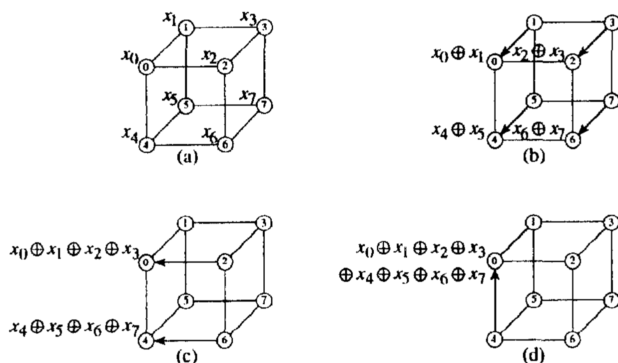

Figure 3: Summing the partial results from each FIR sub-filter section. The arrows indicate communication between nodes. The initial data distribution is shown in (a). Nodes communicate across dimension 0 (b), then dimension 1 (c) and finally dimension 3 (d).

Filter	2-cube			3-cube		
length	Time	S	ϵ (%)	Time	S	ϵ (%)
32	19.33	0.40	9.88	20.90	0.37	4.66
64	19.68	0.50	12.40	20.90	0.47	5.84
128	20.61	0.67	16.70	21.19	0.65	8.11
256	22.58	0.96	24.08	22.10	0.98	12.30
512	26.57	1.42	35.50	24.06	1.57	19.60
1024	34.51	2.02	50.50	28.01	2.49	31.13
2048	50.53	10.43	260.75	36.04	14.62	182.75
4096	82.54	12.71	317.75	52.05	20.16	252.00
8192	537.31	3.90	97.50	84.02	24.92	311.50

Table 1: Execution times (in μs), speedup and efficiency for the concurrent FIR filter running on DspCube. The application code tries to allocate the filter coefficient and filter memory to on-chip memory, if the problem size does not permit this, DRAM is used.

when the filter memory and coefficients are kept in on-chip memory. This accounts for the super-linear speedup that is shown in Table (1). On a hypercube ensemble of processors, the aggregate on-chip memory is obviously larger than for a single processor. Thus, large FIR filters that use only on-chip memory can be implemented on the parallel machine. The availability of a large amount of distributed on-chip memory provides a high level of performance for the inner-product calculations. This advantage on the computational side of the concurrent FIR filter compensates for the communication overhead introduced by the global combine and filter memory updates.

3.1 Hypercube Global Combine Operation

The global combine operation is the major source of inefficiency in the parallel FIR algorithm. The communication pattern required is the most difficult for hypercubes, in the sense that results from every node must be accumulated at a single processor. By comparison, updating the distributed filter memory is straightforward and can be performed efficiently. The overhead incurred by the global combine can be somewhat ameliorated by computing a vector of sub-filter outputs, and then using a vector global combine algorithm. This reduces the per-element communication cost. The time complexity t_{gc1} of the global combine algorithm 1 described above when length n vectors are combined is

$$t_{gc1} = d(\alpha + n\beta + nt_{\oplus}) \tag{6}$$

where α is the message startup time, β is the per vector element transfer time, $t_{\oplus}$ is the time to combine two vector elements, n is the vector length and d is the dimension of the hypercube. This algorithm for combining vectors of subfilter outputs from each node is not optimal in the following sense: even if $\alpha = 0 = \beta$, combining $M = 2^d$ vectors requires time $dnt_{\oplus}$ on 2^d nodes versus $(2^d - 1)nt_{\oplus}$ on a single node, yielding a speedup of only $(2^d - 1)/d$.

A less known approach alluded to in [3] does yield optimal use of the nodes if $\alpha = 0 = \beta$. This approach (global combine algorithm 2) is shown in Figure(4). The advantage of this algorithm is that the combining of vector elements is *shared* between all the processing nodes. The time complexity t_{gc2} of this algorithm is

$$t_{gc2} = 2d\alpha + n\frac{2^d - 1}{2^d}(2\beta + t_{\oplus}) \tag{7}$$

Depending on the vector length n, either of global combine algorithm 1 or 2 may be more efficient.

Parallel filters were implemented on DspCube using both global combine algorithms. Detailed measurements of the computation and communication costs for various length filters were obtained. Based on this data, a model of parallel filter performance was constructed. The simple partitioning of FIR filters for hypercubes means the filter scales well to high dimensional hypercubes. The model was then used calculate the filter sample rate achievable for 200 and 4000 tap filters on hypercubes from dimension $d = 0$ to $d = 10$. Vector lengths of $n = 1$ and $n = 100$ are considered. This is shown in Figures(7) and (8). The speedup for the 4000 tap filter is shown in Figure(9).

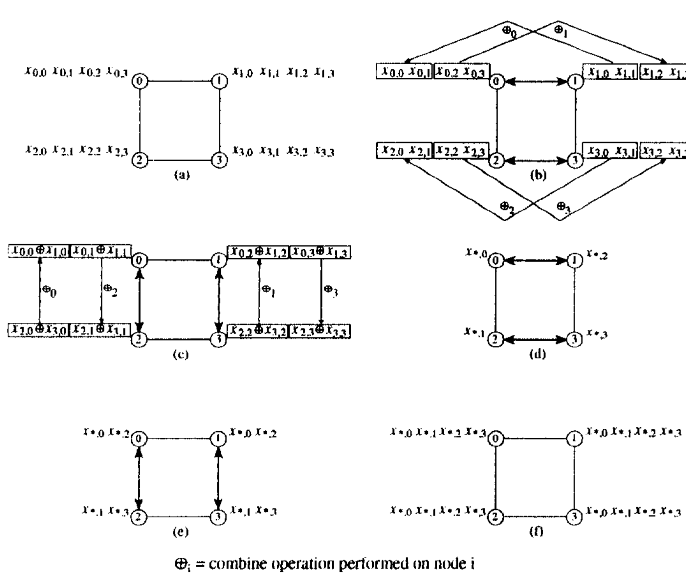

Figure 4: Global combine algorithm 2. This algorithm eliminates the inefficiencies of the more commonly used global combine algorithm shown in Figure(3).

3.2 Reducing the Computation

Higher sample rates for the parallel filter can be achieved by using a fast algorithm for computing the sub-filters. Most fast FIR algorithms use a transform domain approach. The computational advantages of these algorithms can be lost on VLSI DSP processors that are optimised for multiply-accumulate operations. However, in [1], Mou and Duhamel describe a fast FIR algorithm, that unlike transform domain approaches to computing filters, retains the basic multiply-accumulate structure of the FIR filter. The algorithm takes advantage of redundancies between computing consecutive filter outputs. This algorithm is briefly described below. A method of partitioning the filter for a hypercube computer is then presented.

Eq(1) can be written in scalar product form as

$$y_n = (x_n, x_{n-1}, \ldots, x_{n-N+1}) \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{N-1} \end{pmatrix} \quad (8)$$

If two outputs y_n and y_{n-1} are to be computed together, we can write in matrix form

$$\begin{pmatrix} y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} x_{n-1} & x_{n-1} & \cdots & x_{n-N} \\ x_n & x_{n-1} & \cdots & x_{n-N+1} \end{pmatrix} \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{N-1} \end{pmatrix} \quad (9)$$

Now define A, B, C, H_0 and H_1 as

$$A = (x_{n-1}, x_{n-3}, \ldots, x_{n-N+1}) \quad (10)$$

$$B = (x_{n-2}, x_{n-4}, \ldots, x_{n-N}) \quad (11)$$

$$C = (x_n, x_{n-2}, \ldots, x_{n-N+2}) \quad (12)$$

$$H_0 = (h_0, h_2, \ldots, h_{N-2})^T \quad (13)$$

$$H_1 = (h_1, h_3, \ldots, h_{N-1})^T \quad (14)$$

Eq. (9) can now be written as

$$\begin{pmatrix} y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} A & B \\ C & A \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \end{pmatrix} \quad (15)$$

Eq(15) can be re-expressed as

$$\begin{pmatrix} y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} A(H_0 + H_1) + (B - A)H_1 \\ A(H_0 + H_1) - (A - C)H_0 \end{pmatrix} \quad (16)$$

The overall computational load is now that of approximately three filters of length $\frac{1}{2}N$. Figure (5) is the signal-flow of a fast FIR structure based on this approach. Each of the $\frac{1}{2}N$-length filters run at half the sample rate of the original filter. This algorithm requires two input additions, two output additions, and three $\frac{1}{2}N$-tap filters to compute two outputs. The number of arithmetic operations is $2 + \frac{3}{2}\left(\frac{1}{2}N - 1\right)$ additions and $\frac{3}{4}N$ multiplications per output point, which means that both the number of additions and multiplications have been reduced by about 25% compared to a direct implementation of the filter. The effect on filter sample rate when using this algorithm at each node is shown in Figures(7) and (8).

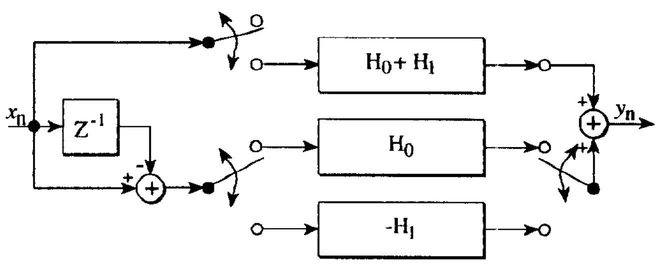

Figure 5: A FIR filter with reduced arithmetic complexity.

An alternative to using the fast FIR algorithm at each node is to perform a parallelisation of Figure(5). Further performance increase seems possible using this technique. This filter structure can be decomposed for hypercubes and load balance maintained. An example of partitioning a length 8 filter on a 2-cube is shown in Figure(6). This particular partitioning reduces link bandwidth requirements in part of the machine and is a topic of current investigation.

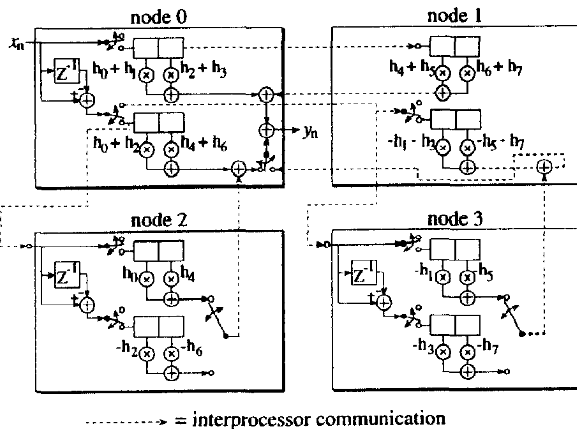

Figure 6: Hypercube partitioning of fast FIR algorithm.

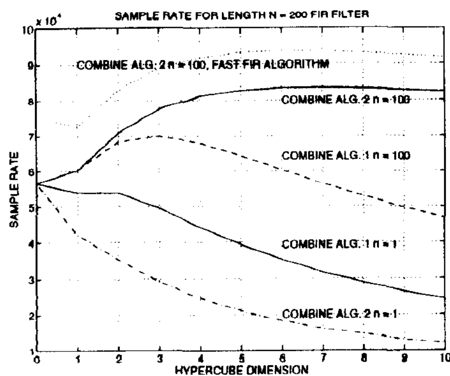

Figure 7: Sample rate versus hypercube dimension for a 200 tap filter.

4 Conclusion

It has been shown that FIR filters map well onto hypercube architectures. The motivation to explore this method for implementing real-time filters, lies in the desire to have a single piece of hardware for use on

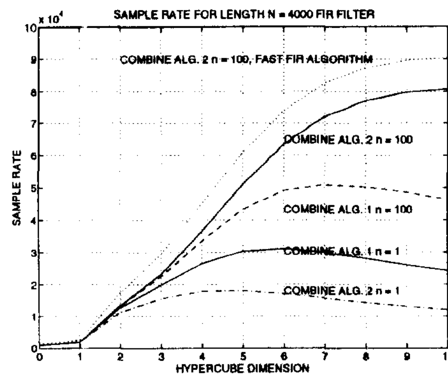

Figure 8: Sample rate versus hypercube dimension for a 4000 tap filter.

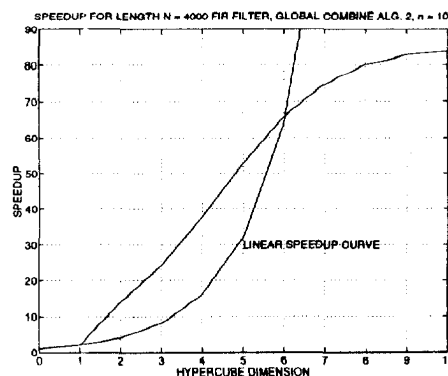

Figure 9: Speedup for the 4000 tap filter.

many signal processing applications. The super-linear speedup obtained for certain filter length/hypercube dimensions is a *real* effect that can be explained by the increase in on-chip memory that occurs when a parallel machine is used.

References

[1] Z. J. Mou, P. Duhamel "Fast FIR Filtering: Algorithms and Implementations," *Signal Processing*, Vol. 13, pp. 377-384, 1987.

[2] F. T. Leighton, *Introduction to parallel algorithms and architectures: arrays,trees,hypercubes*, Morgan Kaufmann, California, 1992.

[3] G. C. Fox M. A. Johnson, G. A. Lyzenga S. W. Otto, and J. K. Salmon, *Solving problems on concurrent processors volume I*, Prentice-Hall, Engelwood Cliffs, NJ, 1988.