COMPARISON OF 2-D FFT IMPLEMENTATIONS ON THE INTEL PARAGON MASSIVELY PARALLEL SUPERCOMPUTER

M. An[1] *N. Anupindi*[1], *M. Bletsas*[3], *G. Kechriotis*[1], *C Lu*[2], *E. S. Manolakos*[3] *and R. Tolimieri*[1]

[1] Aware Inc.
One Memorial Drive
Cambridge, MA 02142

[2] Dept. of Computer Science
Towson State University,
Baltimore, MD 21204

[3] Communications and Digital Signal Processing (CDSP) Center
Electrical and Computer Engineering Dept.
Northeastern University, Boston, MA 02115

ABSTRACT

In this paper [1] we discuss the parallel implementation of multidimensional FFTs on distributed memory multi-processor machines. We introduce a compact notation to describe four equivalent parallel algorithms and discuss their advantages and disadvantages. Two algorithms, suitable for the case when initial and final data are distributed either row- or column-wise, the traditional Row-Column (RC) and a variation of the Vector Radix (VR) that we call partial Vector Radix (PVR) are presented and their efficiency on the Paragon is compared. It is shown that the PVR, although it requires larger amount of interprocessor communication, results in more efficient implementations due to the regularity of local and distributed memory accesses. For the case in which data are partitioned along both dimensions, two suitable parallel algorithms, the Collect-Distribute (CD) and the general full Vector-Radix (FVR), are presented. Again, it is shown that regularity in memory accesses for the case of the FVR, results in more efficient implementations.

1. INTRODUCTION

Since the invention of the Fast Fourier Transform (FFT) [1], Discrete Fourier Transforms (DFT) have been used extensively as a computational tool in many different areas in signal processing, communications and numerical solution of differential equations. With the increase in the use and development of parallel multi-processor machines as a low-cost alternative to traditional supercomputers, the need for efficient parallel algorithms for single and multi-dimensional FFTs has become apparent [2, 3, 4, 5]. Due to its global dependencies (every output point depends on every input point), the parallelization of the FFT is a non-trivial task. The separability property of the multidimensional FFTs allows for the decomposition of the problem into a collection of smaller dimensional problems that can be solved independently in each dimension. We will assume that the parallel system consists of p processors that communicate via interconnection links. Although the data are always physically stored in the memory as an 1D vector, they can be interpreted as multidimensional arrays with suitable dimensions. In such an interpretation it is assumed that the Fortran storage convention is followed, i.e. when the data are being accessed sequentially, the first index of the multidimensional array changes faster, then the second, third etc.

The computation of any multidimensional FFT can be decomposed into smaller building block computations and data reorderings. The tensor product notation [6, 7] along with its identities and transformations provides a convenient mathematical framework for describing multidimensional FFT algorithms as well as for deriving equivalent formulations. In this paper we will describe four equivalent algorithms for the computation of the 2D FFT in a distributed memory multiprocessor environment in terms of elementary operators that act upon the data. These elementary operators are:

- $T2$: transposes a two dimensional array.

[1]This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under contract number F49620-91-C-0098.

- $T3(i,j)$: exchanges the i^{th} and j^{th} indices of a three dimensional array.
- $T4(i,j)$: exchanges the i^{th} and j^{th} indices of a four dimensional array.
- F: performs one or more 1D FFTs. It performs a single FFT if applied to a vector, m FFTs if applied to a 2D array with m columns, m x n FFTs if applied to a 3D array whose last two dimensions are m and n, etc.
- W: pointwise multiplies each of the columns of a matrix (in general consecutive data of length equal to the first dimension of a multidimensional array) by a vector of equal length.
- $G(p)$: All-to-all communication operator. When applied to an (m,p,p) array distributed among p processors along the last dimension, it exchanges the second and third indices. To achieve this, each node keeps one data block of length m, and exchanges $p-1$ messages of length m with $p-1$ other nodes in the network.

Note that only the $G(p)$ operator involves inter-processor communication whereas all the others operate on data that are stored in the processor's local memory.

2. ALGORITHMIC DESCRIPTION

2.1. THE RC METHOD

The Row-Column (RC) method is the most widely used for the implementation of parallel multi-dimensional FFT algorithms. In the 2D case each node holds a number of columns of the 2D array, and first it performs 1D FFTs of the columns. Then, all nodes cooperate in order to *globally transpose* the array, such that each node holds a number of rows of the original array. To perform this global matrix transposition, both local and global data permutations are required that can be expressed in terms of the operators $G(p)$, $T2$ and $T3(i,j)$ introduced in the previous section. After the global data transposition a second set of 1D FFT computations is being performed and finally, one more global transposition step is required if it is desired to obtain the results in the same nodes where the corresponding data were stored.

The RC method can be summarized in terms of the operators T, G and F as follows:

$$\begin{array}{llll}
(m,n/p) & \xrightarrow{F} & (m,n/p) & \longrightarrow \\
(m/p,p,n/p) & \xrightarrow{T3(2,3)} & (m/p,p,n/p) & \longrightarrow \\
(m\cdot n/p^2,p) & \xrightarrow{G(p)} & (m/p,n/p,p) & \longrightarrow \\
(m/p,n) & \xrightarrow{T2} & (n,m/p) & \xrightarrow{F} \\
(n,m/p) & \longrightarrow & (n/p,p,m/p) & \xrightarrow{T3(2,3)} \\
(n/p,m/p,p) & \xrightarrow{G(p)} & (n/p,m/p,p) & \longrightarrow \\
(n/p,m) & \xrightarrow{T2} & (m,n/p) &
\end{array}$$

In the previous algorithmic description an arrow without superscript involves no actual computation or data transfer but simply indicates that a multidimensional array (actually a vector in the memory) is being considered as having a different set of dimensions.

2.2. THE PARTIAL VECTOR RADIX METHOD

In this method, the data is assumed to be distributed in exactly the same fashion as in the RC method. There are p nodes available, and radix p FFTs are computed across the nodes.

$$\begin{array}{llll}
(m,n/p) & \xrightarrow{F} & (m,n/p) & \longrightarrow \\
(m\cdot n/p,p) & \xrightarrow{G(p)} & (m\cdot n/p,p) & \xrightarrow{T2} \\
(p,m\cdot n/p) & \xrightarrow{F} & (p,m\cdot n/p) & \xrightarrow{W} \\
(p,m\cdot n/p) & \xrightarrow{T2} & (m\cdot n/p,p) & \xrightarrow{G(p)} \\
(m\cdot n/p,p) & \longrightarrow & (m,n/p) & \xrightarrow{T2} \\
(n/p,m) & \xrightarrow{F} & (n/p,m) & \xrightarrow{T2} \\
(m,n/p) & \longrightarrow & (m\cdot n/p,p) & \xrightarrow{G(p)} \\
(m\cdot n/p,p) & & &
\end{array}$$

2.3. THE COLLECT-DISTRIBUTE METHOD

For this method, each node is assumed to hold data partitioned along both columns and rows. Each node stores in its local memory an $(m/p,n/q)$ submatrix of the (m,n) array. This method is essentially an extension of the RC method for this distribution of the data. Each node collects first a number of columns, performs 1D FFTs, redistributes them, and then performs the same for the rows. In terms of the operators we defined above, the method can be described as:

$$\begin{array}{llll}
(m/p,n/q) & \longrightarrow & (m/p,n/q\cdot p,p) & \xrightarrow{G(p)} \\
(m/p,n/(q\cdot p),p) & \xrightarrow{T3(2,3)} & (m/p,p,n/(q\cdot p)) & \longrightarrow \\
(m,n/(q\cdot p)) & \xrightarrow{F} & (m,n/(q\cdot p)) & \longrightarrow \\
(m/p,p,n/(q\cdot p)) & \xrightarrow{T3(2,3)} & (m/p,n/(q\cdot p),p) & \xrightarrow{G(p)} \\
(m/p,n/(q\cdot p),p) & \longrightarrow & (m/p,n/q) & \longrightarrow \\
(m/(p\cdot q),q,n/q) & \xrightarrow{T3(2,3)} & (m/(p\cdot q),n/q,q) & \xrightarrow{G(q)} \\
(m/(p\cdot q),n/q,q) & \longrightarrow & (m/(p\cdot q),n) & \xrightarrow{T2} \\
(n,m/(p\cdot q)) & \xrightarrow{F} & (n,m/(p\cdot q)) & \xrightarrow{T2} \\
(m/(p\cdot q),n) & \longrightarrow & (m/(p\cdot q),n/q,q) & \xrightarrow{G(q)} \\
(m/(p\cdot q),n/q,q) & \xrightarrow{T3(2,3)} & (m/(p\cdot q),q,n/q) & \longrightarrow \\
(m/p,n/(q\cdot p),p) & & &
\end{array}$$

2.4. THE GENERAL VECTOR RADIX $P \times Q$ PARALLEL ALGORITHM

The algorithm assumes that $p \times q$ nodes are available, each storing in its local memory an $(m/p,n/q)$ submatrix of the global (m,n) matrix.

$$\begin{array}{l}
(m/p,n/q) \longrightarrow \\
(m/(p\cdot p),p,n/(q\cdot q),q) \xrightarrow{T4(2,3)} \\
(m/(p\cdot p),n/(q\cdot q),p,q) \longrightarrow \\
(m\cdot n/(p\cdot p\cdot q\cdot q),p\cdot q) \xrightarrow{G(p\cdot q)} \\
(m\cdot n/(p\cdot p\cdot q\cdot q),p\cdot q) \xrightarrow{T2} \\
(p\cdot q,m\cdot n/(p\cdot p\cdot q\cdot q)) \longrightarrow \\
(p,q,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{F} \\
(p,q,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{W} \\
(p,q,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{T3(1,2)} \\
(q,p,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{F} \\
(q,p,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{W} \\
(q,p,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{T3(1,2)} \\
(p,q,m\cdot n/(p\cdot p\cdot q\cdot q)) \longrightarrow \\
(p\cdot q,m\cdot n/(p\cdot p\cdot q\cdot q)) \xrightarrow{T2} \\
(m\cdot n/(p\cdot p\cdot q\cdot q),p\cdot q) \xrightarrow{G(p\cdot q)} \\
(m\cdot n/(p\cdot p\cdot q\cdot q),p\cdot q) \longrightarrow \\
(m/(p\cdot p),n/(q\cdot q),p,q) \xrightarrow{F} \\
(m/(p\cdot p),n/(q\cdot q),p,q) \xrightarrow{T4(1,2)} \\
(n/(q\cdot q),m/(p\cdot p),p,q) \xrightarrow{F} \\
(n/(q\cdot q),m/(p\cdot p),p,q) \xrightarrow{T4(1,2)} \\
(m/(p\cdot p),n/(q\cdot q),p,q) \longrightarrow \\
(m\cdot n/(p\cdot p\cdot q\cdot q),p\cdot q) \xrightarrow{G(p\cdot q)} \\
(m\cdot n/(p\cdot p\cdot q\cdot q),p\cdot q) \longrightarrow (m/p,n/q)
\end{array}$$

3. IMPLEMENTATION RESULTS

We have implemented the parallel algorithms described in the previous section on the Intel Paragon multiprocessor system, that is based on the i860XR microprocessor and employs a mesh interconnection network. The programs are written in Fortran calling upon optimized assembly coded routines for the node computations. Assembly-coded routines for the nodes include 1D FFTs, routines from BLAS and matrix transposition routines. The RC method can be made very efficient since optimized 1D FFT routines can be used. For the case of the partial and full VR algorithms, the computation of the p-point FFTs (p is the number of nodes) is being performed either via optimized hand coded assembly routines that perform strided small-sized FFTs with twiddle factor multiplication, or by performing radix-2 butterflies explicitly using vectorized complex multiply-accumulate routines from the BLAS library. When the number of nodes p is relatively small, the computation of the $p-$point FFTs along the second dimension of a $n \times p$ matrix (Fortran storage convention is assumed) via our vectorized FFT codes is substantially faster than transposing the matrix, performing m $p-$point FFTs, and then transposing again. In the vectorized radix-2 butterflies, operations are performed on data vectors of suitable length not to exceed the size of the processor's data cache and therefore the number of cache misses is significantly reduced.

In Table 1, we compare the RC and PVR implementations for a variety of test and machine sizes. Although the PVR method has not been fully optimized it performs generally better than the RC with the advantage being more evident for relatively small sized machine partitions. For more than 16 nodes, the PVR algorithm performs only slightly better than the RC, however substantial optimization can be performed.

In Table 3, we compare the Collect-Distribute (CD) implementation with the Full VR. In both implementations the 2D data are being distributed along both dimensions and the results are obtained in-place. Again, as in the case of the RC, the CD method has the advantage of using highly optimized 1D FFT routines, at the expense of increased data movements. Clearly, as we can see from Table 3, the FVR implementation is more efficient that the CD method, and additional optimization in the computation of the radix $p \times q$ FFTs is possible.

4. CONCLUSIONS - CURRENT DIRECTIONS

Algorithms that do not use the traditional Row-Column approach for the computation of the 2D FFT have been known to result in more efficient implementations on single processor systems. We have shown that the same

m	n	nodes	PVR	RC
256	256	2	83	90
256	512	2	162	190
512	512	2	390	400
512	1024	2	690	918
1024	1024	2	1581	2065
256	512	4	96	109
512	512	4	187	229
512	1024	4	371	495
1024	1024	4	829	1093
1024	2048	4	1729	2282
2048	2048	4	3584	4742
512	512	8	113	123
512	1024	8	210	267
1024	1024	8	449	582
1024	2048	8	900	1186
2048	2048	8	1853	2443
512	512	16	84	66
512	1024	16	140	127
1024	1024	16	254	260
1024	2048	16	484	522
2048	2048	16	973	1110
2048	4096	16	1945	2061
512	512	32	93	71
512	1024	32	119	104
1024	1024	32	189	185
1024	2048	32	318	334
2048	2048	32	542	608
2048	4096	32	1021	1115
4096	4096	32	2036	2087

Table 1: Comparison of the partial Vector-Radix approach and the Row Column optimized implementation (execution times are in milliseconds).

result is true for the case of distributed memory multiprocessor systems with fast interprocessor communication links, despite the larger amount of data communication that is required for the Vector-Radix and Vector-Radix like algorithms. In the case of RISC processors such as the Intel i860 that uses efficient pipelining to perform a floating point multiply accumulate operation every other clock cycle, the advantage is mainly not in the number of processor clock cycles but due to the more regular accessing of the data stored in the local memory of the processors that allows for the optimal use of cache memory. We are currently working on the optimization of the VR codes. Issues of interest are the efficient computation of small-sized 2D FFTs along the second and third dimension of a three dimensional array, and the optimization of the node code for the computation of larger 2D FFTs.

m	n	nodes	FVR	CD
256	512	4	119	155
512	512	4	230	301
512	1024	4	464	606
1024	1024	4	951	1237
1024	2048	4	2111	2676
512	512	8	132	-
512	1024	8	249	-
1024	1024	8	487	-
1024	2048	8	1062	-
2048	2048	8	2180	-
512	512	16	81	99
512	1024	16	151	188
1024	1024	16	275	377
1024	2048	16	546	750
2048	2048	16	1104	1559
2048	4096	16	2402	3106

Table 2: Timings for the Full VR implementation (execution times are in milliseconds).

5. REFERENCES

[1] J.W. Cooley and J.W. Tukey. An Algorithm for the Machine Computation of Complex Fourier Series. *Math. Comp.*, 19:297–301, 1965.

[2] G. Angelopoulos and I. Pitas. Two-dimensional FFT Algorithms on Hypercube and Mesh Machines. *Signal Processing*, 30:355–371, 1993.

[3] G. Kechriotis, M. An, M. Bletsas, R. Tolimieri, and E. S. Manolakos. A New Approach for Computing Multi-dimensional DFTs on Parallel Machines and its Implementation on the iPSC/860 Hypercube. *IEEE Trans. on Singal Processing*, To appear: January 1995.

[4] I. Gertner and M. Rofheart. A Parallel Algorithm for 2-D DFT Computation with No Interprocessor Communication. *IEEE Trans. on Parallel and Distributed Systems*, 1:377–382, July 1990.

[5] I. Gertner and R. Tolimieri. Fast Algorithms to Compute Multidimensional Discrete Fourier Transform. In *Proc. SPIE Real-Time Signal Processing*, pages 132–146, San Diego, CA, 1989.

[6] R. Tolimieri, M. An, and C. Lu. *Mathematics of Multidimensional Fourier Transform Algoritms.* Springer-Verlag, New York, 1993.

[7] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transform and Convolution.* Springer-Verlag, New York, 1989.