計算機アーキテクチャ 123－2
（1997. 3. 7）

並列計算機における二次記憶を用いた一次元FFTの実現と評価

高橋大介[†] 金田康正[††]

本論文では，並列計算機における二次記憶を用いた一次元FFTの実現とその評価について述べる．FFTアルゴリズムは行列の分解に帰着する．この行列分解の考え方を二次記憶を用いた一次元FFTに適用し，さらに並列FFTアルゴリズムに拡張出来ることを示す．また二次記憶を用いたFFTアルゴリズムを具体的に示す．

この並列FFTアルゴリズムを分散メモリ型並列計算機HITACHI SR2201およびIBM SP2上に実現し，性能評価を行った結果について述べる．

Implementation and Evaluation of 1-D FFT with External Memory on Parallel Computers

DAISUKE TAKAHASHI[†] and YASUMASA KANADA[††]

This paper describes how the 1-D external memory fast Fourier transform (FFT) was implemented and the evaluation on the parallel computers. Our 1-D external memory parallel FFT algorithm is derived by means of matrix factorization. External memory FFT algorithms are shown with the experimental results on the distributed memory parallel computers of HITACHI SR2201 and IBM SP2.

1. はじめに

高速フーリエ変換 (fast Fourier transfom, FFT)[1] は，科学技術計算において今日広く用いられているアルゴリズムである．FFTにおいて大量のデータを高速に処理するために，並列計算機におけるFFTアルゴリズムがこれまでに多く提案されている．[2),3)]

近年，並列計算機の主記憶容量は増大する傾向にある．しかし，高速アクセスの可能な主記憶は，コスト的あるいは技術的な要因からその実装可能容量に限界があるため，並列計算機において主記憶に入り切らないデータを処理する場合には，二次記憶を用いた並列アルゴリズムが必要になると考えられる．

これまでに，逐次計算機やベクトル計算機に対する，データが主記憶に収まらないような場合の二次記憶を用いたFFTアルゴリズムが提案されている[4),5)]が，並列計算機においては二次記憶を用いたFFTアルゴリズムの実現や評価は現在十分に行われていないのが現状である．

そこで本論文では，並列計算機において二次記憶を用いた一次元FFTアルゴリズムを実現し，評価した結果について述べる．

FFTアルゴリズムは行列の分解に帰着出来ることが知られている．この行列分解の考え方を二次記憶を用いたFFTに適用し，さらに二次記憶を用いた並列FFTアルゴリズムに拡張出来ることを示す．この並列FFTアルゴリズムを分散メモリ型並列計算機HITACHI SR2201およびIBM SP2上に実現し，性能評価を行う．

以下，第2章で高速フーリエ変換について，第3章で二次記憶を用いたFFTアルゴリズムについて，第4章で二次記憶を用いた並列FFTアルゴリズムを示す．第5章で今回実現した並列アルゴリズムの性能評価結果を示す．最後の第6章はまとめである．

2. 高速フーリエ変換

FFTは，離散フーリエ変換 (discrete Fourier transform, DFT) を高速に計算するアルゴリズムとして知られている．DFTおよび逆DFTは次式で定義される．

$$F_j(x) = \sum_{k=0}^{n-1} x_k e^{-2\pi ijk/n}, \quad 0 \le j < n \quad (1)$$

$$F_j^{-1}(x) = \frac{1}{n} \sum_{k=0}^{n-1} x_k e^{2\pi ijk/n}, \quad 0 \le j < n \quad (2)$$

n が $n = lm$ と分解出来るものとすると，式(1),(2)

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo
†† 東京大学大型計算機センター
Computer Centre, University of Tokyo

における j および k は，$j = pm + q$, $k = rl + s$ と書くことができる．ここで，p, s は $0 \sim l-1$ の値をとり，q, r は $0 \sim m-1$ の値をとるものとする．そうすると，式 (1) は次のように書くことが出来る．

$$
\begin{aligned}
&F_{pm+q}(x) \\
&= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} x_{rl+s} e^{-2\pi i(pm+q)(rl+s)/n} \\
&= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} x_{rl+s} e^{-2\pi ipms/n} e^{-2\pi iqrl/n} e^{-2\pi iqs/n} \\
&= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} x_{rl+s} e^{-2\pi ips/l} e^{-2\pi iqr/m} e^{-2\pi iqs/n} \\
&= \sum_{s=0}^{l-1}\left[\sum_{r=0}^{m-1}(x_{rl+s} e^{-2\pi iqs/n}) e^{-2\pi iqr/m}\right] e^{-2\pi ips/l}, \\
&\qquad\qquad (0 \le p < l,\ 0 \le q < m)
\end{aligned}
$$

上式は，n 点 DFT が l 点 DFT と m 点 DFT に分解されることを示している．

これから，例えば $n = n_1 n_2$ と分解されるとき，次に示すような "four step" FFT アルゴリズムとして知られている方法が導かれる[5]．

1. n_1 組の n_2 点 FFT を $n_1 \times n_2$ の複素数行列に対して行う．
2. $n_1 \times n_2$ 行列 A_{jk} を考え，それに $e^{-2\pi ijk/n}$ を掛ける．
3. $n_1 \times n_2$ 行列を $n_2 \times n_1$ 行列に転置する．
4. n_2 組の n_1 点 FFT を $n_2 \times n_1$ の複素数行列に対して行う．

また "four step" FFT の変形として，"six step" FFT アルゴリズムとして知られている以下に示す方法も導かれている[5]．

1. $n_1 \times n_2$ の複素数行列を $n_2 \times n_1$ 行列に転置する．
2. n_1 組の n_2 点 FFT を $n_2 \times n_1$ 行列に対して行う．
3. $n_2 \times n_1$ 行列 A_{ij} に $e^{-2\pi ijk/n}$ を掛ける．
4. $n_2 \times n_1$ 行列を $n_1 \times n_2$ 行列に転置する．
5. n_2 組の n_1 点 FFT を $n_1 \times n_2$ 行列に対して行う．
6. $n_1 \times n_2$ 行列を $n_2 \times n_1$ 行列に転置する．

3. 二次記憶を用いた FFT アルゴリズム

二次記憶を用いた FFT アルゴリズムとしては，Singleton[4] による逐次計算機におけるアルゴリズムや，Bailey[5] によるベクトル計算機におけるアルゴリズムが知られている．

Singleton による二次記憶を用いた FFT アルゴリズムは，Cooley-Tukey[1] のアルゴリズムをそのまま二次記憶を用いた場合に適用したものといえる．一方，Bailey[5] による二次記憶を用いた場合のアルゴリズムは，第 1 章で述べた "six step" FFT に基づいたアルゴリズムとなっており，ベクトル計算機にも適しているのが特徴である．

"six step" FFT に基づく二次記憶を用いた FFT を FORTRAN で記述したプログラム例を以下に示す．

```
 1  SUBROUTINE DISKFFT(IA,IB,A,B,N1,N2,NBLK)
 2  IMPLICIT REAL*8 (A-H,O-Z)
 3  COMPLEX*16 A(N1*N2/NBLK),B(N1*N2/NBLK)
 4  NBUF=N1*N2/(NBLK*NBLK)
 5  CALL DISKTRANS(IA,IB,A,B,N1,N2,NBLK)
 6  DO K=0,NBLK-1
 7    DO J=1,NBLK
 8      READ(IB,REC=J+NBLK*K)
 9 &        (B((J-1)*NBUF+I),I=1,NBUF)
10    END DO
11    DO J=1,N1/NBLK
12      CALL FFT(B((J-1)*N2+1),N2)
13    END DO
14    CALL TWIDDLE(B,N2,N1,NBLK,K)
15    DO J=1,NBLK
16      WRITE(IB,REC=J+NBLK*K)
17 &        (B((J-1)*NBUF+I),I=1,NBUF)
18    END DO
19  END DO
20  CALL DISKTRANS(IB,IA,B,A,N1,N2,NBLK)
21  DO K=0,NBLK-1
22    DO J=1,NBLK
23      READ(IA,REC=J+NBLK*K)
24 &        (A((J-1)*NBUF+I),I=1,NBUF)
25    END DO
26    DO J=1,N2/NBLK
27      CALL FFT(A((J-1)*N1+1),N1)
28    END DO
29    DO J=1,NBLK
30      WRITE(IA,REC=J+NBLK*K)
31 &        (A((J-1)*NBUF+I),I=1,NBUF)
32    END DO
33  END DO
34  CALL DISKTRANS(IA,IB,A,B,N1,N2,NBLK)
35  RETURN
36  END
```

上のプログラムにおいて，5 行目の `CALL DISKTRANS()` で二次記憶に格納されている $N_1 \times N_2$ 行列を $N_2 \times N_1$ 行列に転置する．次に，7 行目から 18 行目で二次記憶からメモリにデータを読み出し，その都度 N_2 点 FFT および，ひねり係数との乗算を行う．そして，20 行目の `CALL DISKTRANS()` で二次記憶に格納されている $N_2 \times N_1$ 行列を $N_1 \times N_2$ 行列に転置する．さらに，22 行目から 32 行目で，二次記憶からメモリにデータを読み出してその都度 N_1 点 FFT を行う．最後に，34 行目の `CALL DISKTRANS()` で二次記憶に格納されている $N_1 \times N_2$ 行列を $N_2 \times N_1$ 行列に転置して終わりとなる．サブルーチン `DISKTRANS()` においては二次

記憶に格納されている行列の転置が必要になる．二次記憶に格納されている行列の転置アルゴリズムとしてはFraserのアルゴリズム[7),8)]が知られているので，今回の実現に際してはこれを用いた．

この二次記憶を用いたFFTアルゴリズムを実際にベクトル計算機上に実現し，主記憶だけを用いたFFTと性能を比較した．

アルゴリズムとしては，主記憶のみを用いた場合には“four step” FFTを，拡張記憶を用いた場合には，“six step” FFTを用いた．なお，実現にあたっては基数2と基数4を組み合わせることにより高い性能が得られるように工夫している．

今回評価に用いたのはベクトル計算機HITAC S-3800/480 (4CPU，主記憶2GB，拡張記憶32GB，ピーク性能32GFLOPS)である．評価にあたっては，$N = 2^m$のmを変化させてS-3800/480の4つのCPUのうち，1CPU(ピーク性能8GFLOPS)を用いて複素順FFTと逆FFTを10回実行し，その平均のCPU時間を測定することにより行った．なおFFTの計算は倍精度複素数で行い，三角関数のテーブルはあらかじめ作り置きとしている．

プログラムは，全てFORTRANで記述し，コンパイラは日立のFORT77/HAP 26-07を用い，最適化オプションとして
PARM.FORT=('HAP',NODOCHK,ALC,IS,
'INLINE(NOERCHK)',SOPT,DCOM,COMARY)を指定した．

結果は表1のようになった．表1において，*となっているのは，主記憶容量または拡張記憶容量の不足のために実行できなかったことを示している．

表1から分かるように，拡張記憶を用いた場合のFFTは，主記憶だけを用いた場合に比べて，3倍以上性能が悪くなっていることが分かる．これは，拡張記憶へのデータの転送に時間を取られていることと，三角関数のテーブルを全て主記憶内に作成することが出来ないため，その分だけ演算量が増えているのが主な原因である．

例えば2^{24}点FFTでは，主記憶だけを用いた場合にはCPU時間が1.0803秒，VPU時間が1.0798秒と，ベクトルプロセッサの利用率が高くなっているが，拡張記憶を用いた場合には，CPU時間が4.7062秒(うち拡張記憶の転送時間1.1856秒)，VPU時間が3.4106秒となり，拡張記憶の転送時間が全実行時間の約1/4を占めていることが分かる．

また，主記憶だけを用いた場合には，2章で示したように，行列の転置が1回で済む“four step” FFTを適用出来るのに対して，拡張記憶を用いた場合では，行列の転置が3回必要になる“six step” FFTを適用することになるので，行列の転置に要する時間が増大し，メモリコピーの時間もそれだけ増えていることも主記憶だけを用いた場合に比べて不利になる．

表1. 拡張記憶を用いた場合の基数4+2のFFTの性能 (HITAC S-3800/480, 1CPU)

N	主記憶のみを使用		拡張記憶を使用	
	time(s)	MFLOPS	time(s)	MFLOPS
2^{16}	0.0033	3139.7	0.0607	172.8
2^{17}	0.0071	3157.2	0.1042	213.8
2^{18}	0.0136	3469.5	0.1785	264.3
2^{19}	0.0268	3720.8	0.3007	331.3
2^{20}	0.0562	3729.0	0.4914	426.8
2^{21}	0.1216	3621.6	0.8802	500.3
2^{22}	0.2548	3621.7	1.5338	601.6
2^{23}	0.4922	3920.2	2.6985	715.0
2^{24}	1.0803	3727.3	4.7062	855.6
2^{25}	*	*	8.9804	934.1
2^{26}	*	*	17.7517	982.9
2^{27}	*	*	35.7867	1012.6
2^{28}	*	*	72.4143	1037.9

4. 二次記憶を用いた並列FFTアルゴリズム

4.1 アルゴリズムの概要

二次記憶を用いた並列FFTアルゴリズムを考えるにあたって“six step” FFTの考え方を適用する．

一次元FFTはデータ数Nを合成数で表すと，多次元のFFTとして表現できる．[3)]

データの長さを$N = N_1 \cdot N_2$，Pをプロセッサ数とした場合，次の手順でFFTを計算することが出来る[3)]．

1. $N_1 \cdot N_2$の入力データ (行列と考える) に対して，各プロセッサでは，$N_1 \cdot (N_2/P)$のデータを各プロセッサの二次記憶に分散して持つ．
2. $N_1 \times N_2$の行列を$N_2 \times N_1$行列に転置する．
3. 各プロセッサの二次記憶に格納されている$N_2 \times (N_1/P)$のデータを主記憶に読み出しながら，N_1/P組のN_2点FFTを行い，結果を二次記憶の同じ領域に書き戻す．
4. $N_2 \times N_1$行列A_{jk}に，$\exp(-2\pi ijk/N)$を掛ける．
5. $N_2 \times N_1$行列を$N_1 \times N_2$行列に転置する．
6. 各プロセッサは二次記憶に格納されている$N_1 \times N_2$行列から主記憶に読み出しながら，N_2/P組のN_1点FFTを行い，結果を二次記憶の同じ領域に書き戻す．
7. $N_1 \times N_2$行列を$N_2 \times N_1$行列に転置する．

このアルゴリズムは，上記ステップ3,6において$N_1 = N_2 = \sqrt{N}$としたとき，独立な$\sqrt{N}$点FFTが$\sqrt{N}/P$回繰り返される形になるのが特徴である．その

ため，主記憶容量が$\sqrt{N}$よりも大きい場合には主記憶内で$\sqrt{N}$点FFTを繰り返し行うことが出来るという利点がある．

上記ステップ2,5,7の各部分で二次記憶に格納された行列の転置が行われるが，これには後述するような二次記憶を用いた全対全通信が必要にする．このアルゴリズムでは，N_1およびN_2がプロセッサ数Pで割り切れる場合には各プロセッサの演算量は均一となることに注意しておく．

4.2 二次記憶を用いた全対全通信アルゴリズム

4.1節で述べたように，二次記憶を用いた並列FFTアルゴリズムにおいては，二次記憶を用いて全対全通信を行う必要がある．全対全通信アルゴリズムとしては，さまざまなアルゴリズムが提案されている[6]が，今回はデータ数をN，プロセッサ数をPとしたときに，N/P^2個のデータを自分以外のプロセッサに$P-1$回送るアルゴリズムを適用した．通信ライブラリにMPI[13]を使ったFORTRANによる二次記憶を用いた全対全通信のプログラムは次のようになる．

```
 1   SUBROUTINE ALLTOALL(IA,IB,A,B,N,ME,NPU)
 2   INCLUDE 'mpif.h'
 3   REAL*8 A(N/NPU/NPU),B(N/NPU/NPU)
 4   INTEGER*4 ISTATUS(MPI_STATUS_SIZE)
 5   N2=N/(NPU*NPU)
 6   READ(IA,REC=ME+1) A
 7   WRITE(IB,REC=ME+1) A
 8   DO I=1,NPU-1
 9     ISEND=MOD(ME+I,NPU)
10     IRECV=MOD(ME+NPU-I,NPU)
11     READ(IA,REC=ISEND+1) A
12     CALL MPI_ISEND(A,N2,MPI_REAL8,
13    &    ISEND,I,MPI_COMM_WORLD,IREQ1,IERR)
14     CALL MPI_IRECV(B,N2,MPI_REAL8,
15    &    IRECV,I,MPI_COMM_WORLD,IREQ2,IERR)
16     CALL MPI_WAIT(IREQ1,ISTATUS,IERR)
17     CALL MPI_WAIT(IREQ2,ISTATUS,IERR)
18     WRITE(IB,REC=IRECV+1) B
19   END DO
20   RETURN
21   END
```

上のサブルーチンALLTOALLは二次記憶に入っている倍精度実数のデータを他のすべてのプロセッサに全対全通信する．変数IA,IBはI/Oの装置番号，配列A,BはI/Oのバッファである．変数Nはデータの個数，MEはプロセッサ番号であり，NPUはプロセッサ台数である．ここで，ファイルはFORTRANの直接探査ファイルを用いており，レコード長は8*N/(NPU*NPU)となっている．まず，6行目と7行目でプロセッサ間通信の必要のない自分自身のプロセッサ内で，データのコピーを行う．そして，11行目で送信すべきデータを二次記憶IAから配列Aに読み出し，プロセッサ間通信を行う．また，18行目では受信されたデータを二次記憶IBに書き込む．このようにすれば，二次記憶を用いて全対全通信が可能になる．

4.3 プロセッサ内の逐次FFTアルゴリズム

4章で述べたように，今回実現した二次記憶を用いた並列FFTアルゴリズムでは，データ数をNとすると，各プロセッサは$\sqrt{N}$点FFTを$\sqrt{N}/P$回繰り返す形になる．したがって，主記憶の大きさが$\sqrt{N}$以上であれば，$\sqrt{N}$点FFTが主記憶内で実行できるため，I/O回数を削減することが可能になる．

並列計算機の各プロセッサの主記憶内では逐次FFTを行うことになるが，基数2のFFTアルゴリズムとしてはStockhamのアルゴリズム[9]を用いた．このStockhamのアルゴリズムは，Cooley-Tukeyのアルゴリズム[1]に比べて入力と出力が重ね書きできないために，メモリは2倍必要となる．しかし，最内側のループの配列アクセスが連続的になるので，キャッシュのヒット率が良くなるという特徴がある．またベクトルプロセッサにも適しているアルゴリズムとしても知られている[9]．

また基数2のFFTにおいては，演算回数の少ない基数4,8のFFTを部分的に適用することにより効率を高くすることが出来る[10]が，基数を大きくするにしたがってアルゴリズムが複雑となるので今回はStockhamのアルゴリズムを基数2と基数4で実現し，評価を行った．

5. 二次記憶を用いた並列FFTアルゴリズムの性能評価

二次記憶を用いた並列FFTアルゴリズムの評価においては，$N=2^m$のmおよびプロセッサ数Pを変化させて複素順FFTを10回実行し，その平均の実行時間を測定することにより行った．なおFFTの計算は倍精度複素数で行い，三角関数のテーブルはあらかじめ作り置きとしている．

並列計算機として，分散メモリ型並列計算機HITACHI SR2201(1024PE, 総主記憶256GB，理論ピーク性能307.2GFLOPS)のうちの16PEおよびIBM SP2 thin-node(16PE，総主記憶4GB，理論ピーク性能4.3GFLOPS)を用いた．

5.1 HITACHI SR2201

HITACHI SR2201の通信ライブラリとしては，メモリコピーの発生しないリモートDMA転送[12]が高速であるが，今回の実現にあたっては他の機種との互換性があり，プログラミングが容易なMPI[13]を用いた．

プログラムは全てFORTRANで記述した．コンパイラは日立の最適化FORTRAN77 V02-04を用い，最適化オプションとして-W0,'pvec(pvfunc(1)),opt(o(s),fold(2),prefetch(1),rapidcall(1),ischedule(3),reroll(1),scope(1),split(1),uinline(2))'を指定した．

HITACHI SR2201における二次記憶を用いた並列FFTの実行時間を測定した結果を表2に示す．表2において，*となっているのは，主記憶容量または二次記憶容量の不足のために実行できなかったことを示している．

表2. 二次記憶を用いた並列FFTの実行時間
HITACHI SR2201 (単位 sec)

N	P=1	P=2	P=4	P=8	P=16
2^{14}	1.722	1.282	1.620	2.105	3.659
2^{15}	3.273	2.006	2.187	2.857	4.661
2^{16}	6.285	3.519	3.003	3.285	4.964
2^{17}	12.333	6.692	4.104	5.053	6.272
2^{18}	24.482	12.738	7.331	6.320	8.047
2^{19}	48.774	25.095	13.567	9.388	10.526
2^{20}	97.252	49.598	25.787	16.000	16.371
2^{21}	195.339	98.467	50.103	30.611	31.658
2^{22}	*	198.346	99.029	68.518	73.869
2^{23}	*	397.038	196.523	136.081	139.489
2^{24}	*	*	393.733	258.508	271.369

表3. 主記憶だけを用いた並列FFTの実行時間
HITACHI SR2201 (単位 sec)

N	P=1	P=2	P=4	P=8	P=16
2^{14}	0.030	0.018	0.012	0.009	0.009
2^{15}	0.060	0.036	0.021	0.014	0.011
2^{16}	0.122	0.070	0.040	0.025	0.017
2^{17}	0.252	0.142	0.078	0.046	0.030
2^{18}	0.524	0.293	0.157	0.087	0.054
2^{19}	1.080	0.606	0.322	0.171	0.099
2^{20}	2.247	1.242	0.659	0.346	0.185
2^{21}	*	2.569	1.353	0.705	0.368
2^{22}	*	*	2.795	1.424	0.742
2^{23}	*	*	*	2.929	1.503
2^{24}	*	*	*	*	3.075

表2から分かるように，$N = 2^{14}$点FFTにおいては2台まではプロセッサ数の増加に伴い実行時間が短縮されているが，4台以上になると逆に実行時間が増えてしまっている．これは，今回評価に用いたHITACHI SR2201では計算を行うプロセッサに対してI/Oを行うプロセッサが16:1の割合で装備されているため，プロセッサ台数が増加するにしたがって，I/Oを行うプロセッサにI/Oが集中し，I/O時間が増大しているのが主な原因である．

また，プロセッサ数が増加するに従って一度に送るデータ量がプロセッサ数の自乗に反比例して小さくなるため，I/Oおよびプロセッサ間通信の立ち上がり時間が増大することも原因と考えられる．

参考までに，主記憶だけを用いた並列FFTの実行時間を測定した結果を表3に示す．アルゴリズムとしては，文献[11]で述べているような並列FFTアルゴリズムを用いている．他の測定条件は二次記憶を用いた場合と同様である．

表2と表3から分かるように，二次記憶を用いた場合のFFTは，主記憶だけを用いた場合に比べて，2桁以上性能が悪くなっていることが分かる．

これは，二次記憶へのデータの転送時間にほとんど時間を取られていることと，三角関数のテーブルを全て主記憶内に作成することが出来ないため，その分だけ演算量が増えているのが主な原因である．

5.2 IBM SP2

IBM SP2の通信ライブラリとしてはMPIを用いた．プログラムは全てFORTRANで記述した．コンパイラはIBMのXL Fortran version 3.2を用い，最適化オプションとして-O3 -qarch=pwr2 -qhot -qtune=pwr2を指定した．

IBM SP2における二次記憶を用いた並列FFTの実行時間を測定した結果を表4に示す．表4において，*となっているのは，主記憶容量または二次記憶容量の不足のために実行できなかったことを示している．

表4から分かるように，，IBM SP2の場合は$N = 2^{14}$点FFTのように比較的小さなデータを計算している場合でも，プロセッサ数の増加に伴い実行時間が短縮されていることが分かる．これは，IBM SP2では各プロセッサに二次記憶として磁気ディスクが装備されており，その結果並列にI/Oを行うことが出来るため，その結果I/O時間が削減されているためである．

しかし，プロセッサ数が増加するに従って一度に送るデータ量がプロセッサ数の自乗に反比例して小さくなるため，I/Oおよびプロセッサ間通信の立ち上がり時間は無視できなくなってくると考えられる．特に，二次記憶を用いた全対全通信においては，プロセッサ間通信の立上り時間よりもI/Oの立上り時間が大きいため，なるべく一度に多くのデータを二次記憶から読み書きすることが重要になる．

参考までにIBM SP2においても，主記憶だけを用いた並列FFTの実行時間を測定した結果を表5に示す．アルゴリズムとしては，HITACHI SR2201で用いたものと同じ並列FFTアルゴリズムを用いている．他の測定条件は二次記憶を用いた場合と同様である．

表4と表5から分かるように，$N = 2^{17}$以上の領域においては，二次記憶を用いた場合の並列FFTは，主記憶だけを用いた場合に比べると，最悪でも3倍以下の時間で計算を行うことが出来ることが分かる．これは，並列にI/Oを行うことで二次記憶へのデータの転送時間が少なくなっているので，その結果I/O時間が計算時間に比べてそれほど大きくならずに済んでいるのが原因であると考えられる．

このことより，二次記憶を用いた並列FFTにおいては，各プロセッサが並列にI/Oをすることが出来れば，主記憶だけを用いた場合と比べてもそれほど実行効率が低下しないことが分かる．

表 4. 二次記憶を用いた並列 FFT の実行時間
IBM SP2 (単位 sec)

N	P=1	P=2	P=4	P=8	P=16
2^{14}	0.435	0.259	0.159	0.094	0.087
2^{15}	0.843	0.503	0.280	0.164	0.131
2^{16}	1.616	0.907	0.504	0.300	0.230
2^{17}	3.423	1.735	0.990	0.504	0.340
2^{18}	8.259	3.540	1.853	0.999	0.521
2^{19}	17.970	7.905	3.776	2.046	1.129
2^{20}	40.682	18.657	7.908	4.047	2.231
2^{21}	80.576	40.918	18.401	9.856	5.874
2^{22}	*	90.983	41.309	17.586	10.458
2^{23}	*	*	95.824	44.255	18.561
2^{24}	*	*	*	100.035	48.233

表 5. 主記憶だけを用いた並列 FFT の実行時間
IBM SP2 (単位 sec)

N	P=1	P=2	P=4	P=8	P=16
2^{14}	0.100	0.053	0.021	0.014	0.010
2^{15}	0.272	0.122	0.070	0.029	0.015
2^{16}	0.770	0.285	0.150	0.079	0.032
2^{17}	1.462	0.674	0.333	0.170	0.081
2^{18}	3.614	1.651	0.771	0.374	0.165
2^{19}	9.016	3.999	1.821	0.848	0.398
2^{20}	20.555	9.781	4.330	1.990	0.872
2^{21}	*	21.972	10.870	4.614	2.023
2^{22}	*	*	24.154	11.253	4.841
2^{23}	*	*	*	25.624	12.122
2^{24}	*	*	*	*	26.234

6. ま と め

本論文では，二次記憶を有する並列計算機における一次元 FFT の実現とその評価について述べた．FFT アルゴリズムにおける行列分解の考え方を二次記憶を用いた一次元 FFT に適用して並列 FFT アルゴリズムを構成した．さらに，分散メモリ型並列計算機 HITACHI SR2201 および IBM SP2 上に実現し，評価を行った．

その結果，二次記憶に磁気ディスクを用いた並列 FFT においては，各プロセッサが並列に I/O をすることが出来れば，主記憶だけを用いた場合と比べてもそれほど実行効率が低下しないことが分かった．

しかし，並列計算機の要素プロセッサの演算性能は近年性能が向上しているのに対し，二次記憶 (特に磁気ディスク) のアクセス速度はそれほど速くはなっていないことを考えると，今後は，ベクトル計算機に装備されている拡張記憶と同様の各プロセッサの入出力速度を向上させる装置が必須と考える．

参 考 文 献

1) Cooley, J. W. and Tukey, J. W.: An algorithm for the machine calculation of complex Fourier series, *Math. Comp.*, Vol. 19, pp. 297–301 (1965).
2) Swarztrauber, P. N.: Multiprocessor FFTs, *Parallel Computing*, Vol. 5, pp. 197–210 (1987).
3) Agarwal, R. C.: A High Performance Parallel Algorithm for 1-D FFT, *Proceedings of Supercomputing '92*, pp. 34–40 (1992).
4) Singleton, R. C.: A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage, *IEEE Trans. Audio Electroacoust.*, Vol. 15, pp. 91–98 (1967).
5) Bailey, D. H.: FFTs in External or Hierarchical Memory, *The J. Supercomputing*, Vol. 4, pp. 23–35 (1990).
6) Hegland, M: An implementaion of multiple and multivariate fourier transforms on vector processors, *SIAM J. Sci. Comput.*, Vol. 16, No. 2, pp. 271–288 (1995)
7) Fraser, D.: Array Permutation by Index-Digit Permutation, *J. ACM*, Vol. 23, pp. 298–309 (1976).
8) Swarztrauber, P. N.: Transposing Large Arrays in Extended Memory, in *Multiprocessing in Meterological Models*, G. R. Hoffmann and D.F.Snelling eds., Springer-Verlag, pp. 283–287 (1988).
9) Swarztrauber, P. N.: FFT Algorithms for Vector Computers, *Parallel Computing*, Vol. 1, pp. 45–63 (1984).
10) Temperton, C.: Self-sorting mixed-radix fast Fourier transforms, *J. Comput. Phys.*, Vol. 52, No. 1, pp. 1–23 (1983).
11) 高橋大介，金田康正：分散メモリ型並列計算機による 2, 3, 5 基底の FFT の実現と評価，情報処理学会研究報告 96-HPC-62, pp. 117–122 (1996).
12) 日立製作所：HI-UX/MPP リモート DMA 転送使用の手引 6A20-3-021 (1996).
13) Message Passing Interface Forum: MPI: A Message—Passing Interface Standard, Version 1.1, (1995).