A Blocking Algorithm for Parallel 1-D FFT on Clusters of PCs

Daisuke Takahashi, Taisuke Boku, and Mitsuhisa Sato

Center for Computational Physics, Institute of Information Sciences and Electronics, University of Tsukuba, 1-1-1 Tennodai, Tsukuba-shi, Ibaraki 305-8577, Japan,
{daisuke,taisuke,msato}@is.tsukuba.ac.jp

Abstract. In this paper, we propose a blocking algorithm for a parallel one-dimensional fast Fourier transform (FFT) on clusters of PCs. Our proposed parallel FFT algorithm is based on the six-step FFT algorithm. The six-step FFT algorithm can be altered into a block nine-step FFT algorithm to reduce the number of cache misses. The block nine-step FFT algorithm improves performance by utilizing the cache memory effectively. We use the block nine-step FFT algorithm to design the parallel one-dimensional FFT algorithm. In our proposed parallel FFT algorithm, since we use cyclic distribution, all-to-all communication is required only once. Moreover, the input data and output data are both can be given in natural order. We successfully achieved performance of over 1.3 GFLOPS on an 8-node dual Pentium III 1 GHz PC SMP cluster.

1 Introduction

The fast Fourier transform (FFT) [1] is an algorithm widely used today in science and engineering. Parallel FFT algorithms on distributed-memory parallel computers have been well studied [2,3,4,5,6].

Many FFT algorithms work well when data sets fit into a cache. When a problem size exceeds the cache size, however, the performance of these FFT algorithms decreases dramatically. The key issue of the design for large FFTs is to minimize the number of cache misses.

In this paper, we propose a blocking algorithm for a parallel one-dimensional FFT algorithm on clusters of PCs.

Our proposed parallel one-dimensional FFT algorithm is based on the six-step FFT algorithm [7,8]. The six-step FFT algorithm requires two multicolumn FFTs and three data transpositions. The three transpose steps typically are the chief bottlenecks in cache-based processors.

Some previously presented six-step FFT algorithms [8,9] separate the multicolumn FFTs from the transpositions.

Taking the opposite approach, we combine the multicolumn FFTs and transpositions to reduce the number of cache misses, and we modify the six-step FFT algorithm to reuse data in the cache memory [10]. We call this a block six-step FFT algorithm. The block six-step FFT algorithm can be altered into a block

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 691–700.

nine-step FFT algorithm to reduce the number of cache misses. We use the block nine-step FFT algorithm to design the parallel one-dimensional FFT algorithm.

We have implemented the block nine-step FFT-based parallel one-dimensional FFT algorithm on an 8-node dual Pentium III 1 GHz PC SMP cluster, and we report the performance in this paper.

The rest of the paper is organized as follows. Section 2 describes the six-step FFT algorithm. In section 3, we propose a nine-step FFT algorithm. Section 4, we propose a block nine-step FFT algorithm used for problems that exceed the cache size. Section 5, we propose a parallel FFT algorithm based on the block nine-step FFT. Section 6 describes the in-cache FFT algorithm used for problems that fit into a data cache. Section 7 gives performance results. In section 8, we present some concluding remarks.

2 The Six-Step FFT

The discrete Fourier transform (DFT) is given by

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \quad 0 \le k \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n has factors n_1 and n_2 ($n = n_1 \times n_2$), then the indices j and k can be expressed as:

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{2}$$

We can define x and y as two-dimensional arrays (in Fortran notation):

$$x_j = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{3}$$

$$y_k = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{4}$$

Substituting the indices j and k in equation (1) with those in equation (2), and using the relation of $n = n_1 \times n_2$, we can derive the following equation:

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{5}$$

This derivation leads to the following six-step FFT algorithm [7,8]:

Step 1: Transpose

$$x_1(j_2, j_1) = x(j_1, j_2).$$

Step 2: n_1 individual n_2-point multicolumn FFTs

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1) \omega_{n_2}^{j_2 k_2}.$$

Step 3: Twiddle-factor multiplication

$$x_3(k_2, j_1) = x_2(k_2, j_1)\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 4: Transpose

$$x_4(j_1, k_2) = x_3(k_2, j_1).$$

Step 5: n_2 individual n_1-point multicolumn FFTs

$$x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1 k_1}.$$

Step 6: Transpose

$$y(k_2, k_1) = x_5(k_1, k_2).$$

The distinctive features of the six-step FFT algorithm can be summarized as:

- Two multicolumn FFTs are performed, one in step 2 and the other in step 5. Each column FFT is small enough to fit into the data cache.
- The six-step FFT algorithm has three transpose steps, which typically are the chief bottlenecks in cache-based processors.

In order to reduce the number of cache misses, the block six-step FFT algorithm has been proposed [10].

3 A Nine-Step FFT Algorithm

We can extend the six-step FFT algorithm in another way into a three-dimensional formulation. If n has factors n_1, n_2 and n_3 ($n = n_1 n_2 n_3$), then the indices j and k can be expressed as:

$$\begin{aligned} j &= j_1 + j_2 n_1 + j_3 n_1 n_2, \\ k &= k_3 + k_2 n_3 + k_1 n_2 n_3. \end{aligned} \tag{6}$$

We can define x and y as three-dimensional arrays (in Fortran notation), e.g.,

$$\begin{aligned} x_j = x(j_1, j_2, j_3), \quad & 0 \le j_1 \le n_1 - 1, \\ 0 \le j_2 \le n_2 - 1, \quad & 0 \le j_3 \le n_3 - 1, \end{aligned} \tag{7}$$

$$\begin{aligned} y_k = y(k_3, k_2, k_1), \quad & 0 \le k_1 \le n_1 - 1, \\ 0 \le k_2 \le n_2 - 1, \quad & 0 \le k_3 \le n_3 - 1. \end{aligned} \tag{8}$$

Substituting the indices j and k in equation (1) by those in equation (6) and using the relation of $n = n_1 n_2 n_3$, we can derive the following equation:

$$\begin{aligned} y(k_3, k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} & x(j_1, j_2, j_3)\omega_{n_3}^{j_3 k_3} \\ & \omega_{n_2 n_3}^{j_2 k_3} \omega_{n_2}^{j_2 k_2} \omega_{n}^{j_1 k_3} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \end{aligned} \tag{9}$$

This derivation leads to the following nine-step FFT:

Step 1: Transpose

$$x_1(j_3, j_1, j_2) = x(j_1, j_2, j_3).$$

Step 2: $n_1 n_2$ individual n_3-point multicolumn FFTs

$$x_2(k_3, j_1, j_2) = \sum_{j_3=0}^{n_3-1} x_1(j_3, j_1, j_2)\omega_{n_3}^{j_3 k_3}.$$

Step 3: Twiddle-factor multiplication

$$x_3(k_3, j_1, j_2) = x_2(k_3, j_1, j_2)\omega_{n_2 n_3}^{j_2 k_3}.$$

Step 4: Transpose

$$x_4(j_2, j_1, k_3) = x_3(k_3, j_1, j_2).$$

Step 5: $n_1 n_3$ individual n_2-point multicolumn FFTs

$$x_5(k_2, j_1, k_3) = \sum_{j_2=0}^{n_2-1} x_4(j_2, j_1, k_3)\omega_{n_2}^{j_2 k_2}.$$

Step 6: Twiddle-factor multiplication

$$x_6(k_2, j_1, k_3) = x_5(k_2, j_1, k_3)\omega_n^{j_1 k_3}\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 7: Transpose

$$x_7(j_1, k_2, k_3) = x_6(k_2, j_1, k_3).$$

Step 8: $n_2 n_3$ individual n_1-point multicolumn FFTs

$$x_8(k_1, k_2, k_3) = \sum_{j_1=0}^{n_1-1} x_7(j_1, k_2, k_3)\omega_{n_1}^{j_1 k_1}.$$

Step 9: Transpose

$$y(k_3, k_2, k_1) = x_8(k_1, k_2, k_3).$$

The distinctive features of the nine-step FFT algorithm can be summarized as:

- Three multicolumn FFTs are performed in steps 2, 5 and 8. The locality of the memory reference in the multicolumn FFT is high. Therefore, the nine-step FFT is suitable for cache-based processors because of the high performance which can be obtained with high hit rates in the cache memory.
- The matrix transposition takes place four times.

For extremely large FFTs, we should switch to a four-dimensional formulation and higher approaches.

4 A Block Nine-Step FFT Algorithm

We combine the multicolumn FFTs and transpositions to reduce the number of cache misses, and we modify the nine-step FFT algorithm to reuse data in the

```
COMPLEX*16 X(N1,N2,N3),Y(N3,N2,N1)
COMPLEX*16 U2(N3,N2),U3(N1,N2,N3)
COMPLEX*16 YWORK(N2+NP,NB),ZWORK(N3+NP,NB)
DO J=1,N2
  DO II=1,N1,NB
    DO KK=1,N3,NB
      DO I=II,II+NB-1
        DO K=KK,KK+NB-1
          ZWORK(K,I-II+1)=X(I,J,K)
        END DO
      END DO
    END DO
    DO I=1,NB
      CALL IN_CACHE_FFT(ZWORK(1,I),N3)
    END DO
    DO K=1,N3
      DO I=II,II+NB-1
        X(I,J,K)=ZWORK(K,I-II+1)*U2(K,J)
      END DO
    END DO
  END DO
END DO
DO K=1,N3
  DO II=1,N1,NB
    DO JJ=1,N2,NB
      DO I=II,II+NB-1
        DO J=JJ,JJ+NB-1
          YWORK(J,I-II+1)=X(I,J,K)
        END DO
      END DO
    END DO
    DO I=1,NB
      CALL IN_CACHE_FFT(YWORK(1,I),N2)
    END DO
    DO J=1,N2
      DO I=II,II+NB-1
        X(I,J,K)=YWORK(J,I-II+1)*U3(I,J,K)
      END DO
    END DO
  END DO
  DO J=1,N2
    CALL IN_CACHE_FFT(X(1,J,K),N1)
  END DO
END DO
DO II=1,N1,NB
  DO JJ=1,N2,NB
    DO KK=1,N3,NB
      DO I=II,II+NB-1
        DO J=JJ,JJ+NB-1
          DO K=KK,KK+NB-1
            Y(K,J,I)=X(I,J,K)
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

Fig. 1. A Block Nine-Step FFT Algorithm

cache memory. As in the nine-step FFT above, it is assumed in the following that $n = n_1 n_2 n_3$ and that n_b is the block size. We assume that each processor has a multi-level cache memory. A block nine-step FFT algorithm can be stated as follows.

1. Consider the data in main memory as an $n_1 \times n_2 \times n_3$ complex array `X`. Fetch and transpose the data n_b rows at a time into an $n_3 \times n_b$ work array `ZWORK`. The $n_3 \times n_b$ work array `ZWORK` fits into the L2 cache.
2. For each n_b columns, perform n_b individual n_3-point multicolumn FFTs on the $n_3 \times n_b$ array `ZWORK` in the L2 cache. Each column FFT fits into the L1 data cache.
3. Multiply the resulting data in each of the $n_3 \times n_b$ complex matrices by the twiddle factors `U2`. Then transpose each of the resulting $n_3 \times n_b$ matrices, and return the resulting n_b rows to the same locations in the main memory from which they were fetched.
4. Fetch and transpose the data n_b rows at a time into an $n_2 \times n_b$ work array `YWORK`.
5. Perform n_b individual n_2-point multicolumn FFTs on the $n_2 \times n_b$ work array `YWORK` in the L2 cache. Each column FFT fits into the L1 data cache.
6. Multiply the resulting data in each of the $n_2 \times n_b$ complex matrices by the twiddle factors `U3`. Then transpose each of the resulting $n_2 \times n_b$ matrices, and return the resulting n_b rows to the same locations in the main memory from which they were fetched.

7. Perform $n_3 n_2$ individual n_1-point multicolumn FFTs on the $n_1 \times n_2 \times n_3$ array `X`. Each column FFT fits into the L1 data cache.
8. Transpose and store the resulting data on an $n_3 \times n_2 \times n_1$ complex matrix.

We note that this algorithm is a *three-pass* algorithm. Fig. 1 gives the pseudo-code for this block nine-step FFT algorithm. Here the twiddle factors $\omega_{n_2 n_3}^{j_2 k_3}$ and $\omega_n^{j_1 k_3} \omega_{n_1 n_2}^{j_1 k_2}$ are stored in arrays `U2` and `U3`, respectively. The arrays `YWORK` and `ZWORK` are the work arrays. The parameters `NB` and `NP` are the blocking parameter and the padding parameter, respectively. If an out-of-place algorithm (e.g. Stockham autosort algorithm [11]) is used for the individual FFTs, the additional scratch requirement for performing the individual FFTs in steps 2, 5 and 7 is $O(n^{1/3})$ at most.

If we do not require an ordered transform, the $n_3 \times n_2 \times n_1$ complex matrix (the array `Y` in Fig. 1) and the transposition in step 8 can be omitted.

5 Parallel FFT Algorithm Based on the Block Nine-Step FFT

We can adopt the idea of the block nine-step FFT as described in section 4.

Let N have factors N_1, N_2 and N_3 ($N = N_1 \times N_2 \times N_3$). The original one-dimensional array $x(N)$ can be defined as a three-dimensional array $x(N_1,N_2,N_3)$ (in Fortran notation). On a distributed-memory parallel computer which has P nodes, the array $x(N_1, N_2, N_3)$ is distributed along the first dimension N_1. If N_1 is divisible by P, each node has distributed data of size N/P. We introduce the notation $\hat{N}_r \equiv N_r/P$ and we denote the corresponding index as $\hat{J}_r$ which indicates that the data along J_r are distributed across all P nodes. Here, we use the subscript r to indicate that this index belongs to dimension r. The distributed array is represented as $\hat{x}(\hat{N}_1, N_2, N_3)$. At node m, the local index $\hat{J}_r(m)$ corresponds to the global index as the *cyclic* distribution:

$$J_r = \hat{J}_r(m) \times P + m, \quad 0 \le m \le P-1, \quad 1 \le r \le 3. \tag{10}$$

To illustrate the all-to-all communication it is convenient to decompose N_i into two dimensions $\tilde{N}_i$ and P_i, where $\tilde{N}_i \equiv N_i/P_i$. Although P_i is the same as P, we are using the subscript i to indicate that this index belongs to dimension i.

Starting with the initial data $\hat{x}(\hat{N}_1, N_2, N_3)$, the nine-step FFT-based parallel FFT can be performed according to the following steps:

Step 1: Transpose

$$\hat{x_1}(J_3, \hat{J}_1, J_2) = \hat{x}(\hat{J}_1, J_2, J_3).$$

Step 2: $(N_1/P) \cdot N_2$ individual N_3-point multicolumn FFTs

$$\hat{x_2}(K_3, \hat{J}_1, J_2) = \sum_{J_3=0}^{N_3-1} \hat{x_1}(J_3, \hat{J}_1, J_2) \omega_{N_3}^{J_3 K_3}.$$

Step 3: Twiddle-factor multiplication and rearrangement

$$\hat{x_3}(\hat{J_1},\, J_2,\, \tilde{K_3},\, P_3) = \hat{x_2}(P_3,\, \tilde{K_3},\, \hat{J_1},\, J_2)\omega_{N_2N_3}^{J_2K_3}$$
$$\equiv \hat{x_2}(K_3,\, \hat{J_1},\, J_2)\omega_{N_2N_3}^{J_2K_3}.$$

Step 4: All-to-all communication

$$\hat{x_4}(\tilde{J_1},\, J_2,\, \hat{K_3},\, P_1) = \hat{x_3}(\hat{J_1},\, J_2,\, \tilde{K_3},\, P_3).$$

Step 5: Rearrangement

$$\hat{x_5}(J_2,\, \tilde{J_1},\, \hat{K_3},\, P_1) = \hat{x_4}(\tilde{J_1},\, J_2,\, \hat{K_3},\, P_1).$$

Step 6: $N_1 \cdot (N_3/P)$ individual N_2-point multicolumn FFTs

$$\hat{x_6}(K_2,\, \tilde{J_1},\, \hat{K_3},\, P_1) = \sum_{J_2=0}^{N_2-1} \hat{x_5}(J_2,\, \tilde{J_1},\, \hat{K_3},\, P_1)\omega_{N_2}^{J_2K_2}.$$

Step 7: Twiddle-factor multiplication and rearrangement

$$\hat{x_7}(J_1,\, K_2,\, \hat{K_3}) \equiv \hat{x_7}(P_1,\, \tilde{J_1},\, K_2,\, \hat{K_3})$$
$$= \hat{x_6}(K_2,\, \tilde{J_1},\, \hat{K_3},\, P_1)\omega_N^{J_1(\hat{K_3}+K_2N_3)}.$$

Step 8: $N_2 \cdot (N_3/P)$ individual N_1-point multicolumn FFTs

$$\hat{x_8}(K_1,\, K_2,\, \hat{K_3}) = \sum_{J_1=0}^{N_1-1} \hat{x_7}(J_1,\, K_2,\, \hat{K_3})\omega_{N_1}^{J_1K_1}.$$

Step 9: Transpose

$$\hat{y}(\hat{K_3},\, K_2,\, K_1) = \hat{x_8}(K_1,\, K_2,\, \hat{K_3}).$$

The distinctive features of the nine-step FFT-based parallel FFT algorithm can be summarized as:

- $N^{2/3}/P$ individual $N^{1/3}$-point multicolumn FFTs are performed in steps 2, 6 and 8 for the case of $N_1 = N_2 = N_3 = N^{1/3}$.
- The all-to-all communication occurs just once. Moreover, the input data x and the output data y are both can be given *natural order*.

If both of N_1 and N_3 are divisible by P, the workload on each node is also uniform.

6 In-Cache FFT Algorithm

We use the radix-2, 4 and 8 Stockham autosort algorithm for in-cache FFTs.

Table 1 shows the number of operations required for radix-2, 4 and 8 FFT kernels on Pentium III processor. The higher radices are more efficient in terms of both memory and floating-point operations. A high ratio of floating-point instructions to memory operations is particularly important in a cache-based processor. In view of the high ratio of floating-point instructions to memory operations, the radix-8 FFT is more advantageous than the radix-4 FFT. A power-of-two FFT (except for 2-point FFT) can be performed by a combination of radix-8 and radix-4 steps containing at most two radix-4 steps. That is, the power-of-two FFTs can be performed as a length $n = 2^p = 4^q 8^r$ ($p \geq 2,\ 0 \leq q \leq 2,\ r \geq 0$).

Table 1. Real inner-loop operations for radix-2, 4 and 8 FFT kernels based on the Stockham autosort algorithm on Pentium III processor

	Radix-2	Radix-4	Radix-8
Loads and stores	8	16	32
Multiplications	4	12	32
Additions	6	22	66
Total floating-point operations ($n \log_2 n$)	5.000	4.250	4.083
Floating-point instructions	10	34	98
Floating-point/memory ratio	1.250	2.125	3.063

7 Performance Results

To evaluate the block nine-step FFT-based parallel FFT, we compared its performance against that of the block nine-step FFT-based parallel FFT and that of the FFT library of the FFTW (version 2.1.3) [12] which is known as one of the fastest FFT libraries for many processors. We averaged the elapsed times obtained from 10 executions of complex forward FFTs. These parallel FFTs were performed on double-precision complex data and the table for twiddle factors was prepared in advance.

An 8-node dual Pentium III 1 GHz PC SMP cluster (i840 chipset, 1 GB RDRAM main memory per node, Linux 2.2.16) was used. The nodes on the PC SMP cluster are interconnected through a 1000Base-SX Gigabit Ethernet switch.

MPICH-SCore [13] was used as a communication library. We used an intra-node MPI library for the PC SMP cluster.

For the block nine-step FFT-based parallel FFT, the compiler used was g77 version 2.95.2. For the FFTW, the compiler used was gcc version 2.95.2.

Table 2 compares the block nine-step FFT-based parallel FFT and the FFTW in terms of their run times and MFLOPS. The first column of a table indicates the number of processors. The second column gives the problem size. The next four columns contain the average elapsed time in seconds and the average execution performance in MFLOPS. The MFLOPS value is based on $5N \log_2 N$ for a transform of size $N = 2^m$.

Table 3 shows the results of the all-to-all communication timings on the dual Pentium III PC SMP cluster. The first column of a table indicates the number of processors. The second column gives the problem size. The next two columns contain the average elapsed time in seconds and the average bandwidth in MB/sec.

For $N = 2^{26}$ and $P = 8{\times}2$, the block nine-step FFT-based parallel FFT runs about 2.75 times faster than the FFTW, as shown in Table 2.

In Tables 2 and 3, we can clearly see that all-to-all communication overhead dominates the execution time. Although the FFTW requires three all-to-all communication steps, the block nine-step FFT-based parallel FFT requires only one all-to-all communication step. Moreover, the performance of the block nine-step

Table 2. Performance of parallel one-dimensional FFTs on dual Pentium III PC SMP cluster

P (Nodes × CPUs)	N	Block Nine-Step FFT		FFTW	
		Time	MFLOPS	Time	MFLOPS
1×1	2^{23}	5.40606	178.45	10.50152	91.86
1×2	2^{23}	3.33968	288.86	7.49437	128.72
2×1	2^{24}	7.46566	269.67	16.55127	121.64
2×2	2^{24}	4.99214	403.29	11.96556	168.26
4×1	2^{25}	8.22695	509.82	17.79209	235.74
4×2	2^{25}	6.01907	696.84	15.44108	271.63
8×1	2^{26}	8.68712	1004.26	19.33295	451.26
8×2	2^{26}	6.58020	1325.82	18.06414	482.95

Table 3. All-to-all communication performance on dual Pentium III PC SMP cluster

P (Nodes × CPUs)	N	Time	MB/sec
1×2	2^{23}	0.46537	72.10
2×1	2^{24}	2.18825	30.67
2×2	2^{24}	2.00209	25.14
4×1	2^{25}	2.48046	40.58
4×2	2^{25}	2.60625	22.53
8×1	2^{26}	3.01393	38.97
8×2	2^{26}	3.46417	18.16

FFT-based parallel FFT remains at a high level even for the larger problem size, owing to cache blocking. This is the reason why the block nine-step FFT-based parallel FFT is most advantageous with the dual Pentium III PC SMP cluster.

These results clearly indicate that the block nine-step FFT-based parallel FFT is superior to the FFTW.

We note that on an 8-node dual Pentium III 1 GHz PC SMP cluster, over 1.3 GFLOPS was realized with size $N = 2^{26}$ in the block nine-step FFT-based parallel FFT as in Table 2.

8 Conclusion

In this paper, we proposed a blocking algorithm for parallel one-dimensional FFT on clusters of PCs. We reduced the number of cache misses for the six-step FFT algorithm. In our proposed parallel FFT algorithm, since we use cyclic distribution, all-to-all communication is required only once. Moreover, the input data and output data are both can be given in natural order.

Our block nine-step FFT-based parallel one-dimensional FFT algorithm has resulted in high-performance one-dimensional parallel FFT transforms suitable for clusters of PCs. The block nine-step FFT-based parallel FFT is most advan-

tageous with processors that have a considerable gap between the speed of the cache memory and that of the main memory.

We successfully achieved performance of over 1.3 GFLOPS on an 8-node dual Pentium III 1 GHz PC SMP cluster. The performance results demonstrate that the block nine-step FFT-based parallel FFT has low communication cost, and utilize cache memory effectively.

Implementation of the extended split-radix FFT algorithm [**?**], which has a lower operation count than radix-8 FFT on clusters of PCs is one of the important problems for the future.

The proposed block nine-step FFT-based parallel one-dimensional FFT routines can be obtained in the "*FFTE*" package at `http://www.ffte.jp`.

References

1. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19** (1965) 297–301
2. Swarztrauber, P.N.: Multiprocessor FFTs. Parallel Computing **5** (1987) 197–210
3. Agarwal, R.C., Gustavson, F.G., Zubair, M.: A high performance parallel algorithm for 1-D FFT. In: Proc. Supercomputing '94. (1994) 34–40
4. Hegland, M.: A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. Numerische Mathematik **68** (1994) 507–547
5. Edelman, A., McCorquodale, P., Toledo, S.: The future fast Fourier transform? SIAM J. Sci. Comput. **20** (1999) 1094–1114
6. Mirković, D., Johnsson, S, L.: Automatic performance tuning in the UHFFT library. In: Proc. 2001 International Conference on Computational Science (ICCS 2001). Volume 2073 of Lecture Notes in Computer Science., Springer-Verlag (2001) 71–80
7. Bailey, D.H.: FFTs in external or hierarchical memory. The Journal of Supercomputing **4** (1990) 23–35
8. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia, PA (1992)
9. Wadleigh, K.R.: High performance FFT algorithms for cache-coherent multiprocessors. The International Journal of High Performance Computing Applications **13** (1999) 163–171
10. Takahashi, D.: A blocking algorithm for FFT on cache-based processors. In: Proc. 9th International Conference on High Performance Computing and Networking Europe (HPCN Europe 2001). Volume 2110 of Lecture Notes in Computer Science., Springer-Verlag (2001) 551–554
11. Swarztrauber, P.N.: FFT algorithms for vector computers. Parallel Computing **1** (1984) 45–63
12. Frigo, M., Johnson, S.G.: The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, MIT Lab for Computer Science (1997)
13. Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T., Ishikawa, Y.: High performance communication using a commodity network for cluster systems. In: Proc. Ninth International Symposium on High Performance Distributed Computing (HPDC-9). (2000) 139–146
14. Takahashi, D.: An extended split-radix FFT algorithm. IEEE Signal Processing Letters **8** (2001) 145–147