Journal of Supercomputing, 15, 207–228 (2000)

High-Performance Radix-2, 3 and 5 Parallel 1-D Complex FFT Algorithms for Distributed-Memory Parallel Computers

DAISUKE TAKAHASHI daisuke@pi.cc.u-tokyo.ac.jp
YASUMASA KANADA kanada@pi.cc.u-tokyo.ac.jp
Computer Centre, University of Tokyo, 2-11-16 Yayoi, Bunkyo-ku, Tokyo, 113-8658, Japan

(Received July 1998; final version accepted July 1999.)

Abstract. In this paper, we propose high-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. We use the four-step or six-step FFT algorithms to implement the radix-2, 3 and 5 parallel 1-D complex FFT algorithms. In our parallel FFT algorithms, since we use cyclic distribution, all-to-all communication takes place only once. Moreover, the input data and output data are both in natural order.

We also show that the suitability of a parallel FFT algorithm is machine-dependent because of the differences in the architecture of the processor elements in distributed-memory parallel computers. Experimental results of $2^p 3^q 5^r$ point FFTs on distributed-memory parallel computers, HITACHI SR2201 and IBM SP2 are reported. We succeeded to get performances of about 130 GFLOPS on a 1024PE HITACHI SR2201 and about 1.25 GFLOPS on a 32PE IBM SP2.

Keywords: fast Fourier transform, radix-2, 3 and 5, distributed-memory parallel computer, cyclic distribution, all-to-all communication

1. Introduction

The fast Fourier transform (FFT) [7] is an algorithm widely used today in science and engineering. Parallel FFT algorithms have been well studied [15, 3, 9, 2, 8].

For almost all scalar and vector computers, FFT algorithms with radix-2, 3 and 5 are proposed [13, 17, 1]. Many vendors support parallel 1-D complex and real FFT algorithms with radix-2, but few vendors support radix-2, 3 and 5 parallel 1-D complex FFT on distributed-memory parallel computers.

The parallel FFT algorithm can be derived from the four-step or six-step FFT algorithms [19]. These ideas can be adopted not only for the radix-2 parallel FFT but also for the radix-2, 3 and 5 parallel FFT. We succeeded to implement a radix-2, 3 and 5 parallel 1-D complex FFT algorithm on the HITACHI SR2201 and the IBM SP2, and we report their performance in this paper.

According to theoretical analysis, we show that the suitability of the parallel FFT algorithm differs between machines because of the difference of the CPU architecture for the processor elements of distributed-memory parallel computers.

2. The four-step and six-step FFT algorithms

2.1. The four-step FFT

The discrete Fourier transform (DFT) is given by

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

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:

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

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

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

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

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 four-step FFT algorithm [19, 4]:

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

Step 2: $x_2(j_1, k_2) = x_1(j_1, k_2) \omega_{n_1 n_2}^{j_1 k_2}$.

Step 3: $x_3(k_2, j_1) = x_2(j_1, k_2)$.

Step 4: $y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} x_3(k_2, j_1) \omega_{n_1}^{j_1 k_1}$.

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

- If n_1 is equal to n_2 ($n_1 = n_2 \equiv \sqrt{n}$), the innermost loop length can be fixed to $\sqrt{n}$. This feature makes the algorithm suitable for vector processors.
- A matrix transposition takes place just once (step 3).
- Two multirow FFTs are performed in steps 1 and 4. In this case the locality of the memory references is low, resulting in many cache misses. The four-step FFT is therefore not suitable for the RISC processors which depend on high cache hit rates to obtain high performance.

2.2. *The six-step FFT*

There is an algorithm known as the six-step FFT algorithm which is an extension of the four-step FFT algorithm [19, 4] in the following sense:

Step 1: $x_1(j_2, j_1) = x(j_1, j_2)$.

Step 2: $x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1)\omega_{n_2}^{j_2k_2}$.

Step 3: $x_3(k_2, j_1) = x_2(k_2, j_1)\omega_{n_1n_2}^{j_1k_2}$.

Step 4: $x_4(j_1, k_2) = x_3(k_2, j_1)$.

Step 5: $x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1k_1}$.

Step 6: $y(k_2, k_1) = x_5(k_1, k_2)$.

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

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

2.3. *An extended three-dimensional four-step FFT*

We can extend the four-step FFT algorithm in another way into a three-dimensional formulation. If n has factors n_1, n_2 and n_3 ($n = n_1n_2n_3$), then the indices j and k can be expressed as:

$$\begin{aligned} j &= j_1 + j_2n_1 + j_3n_1n_2, \\ k &= k_3 + k_2n_3 + k_1n_2n_3. \end{aligned} \qquad [6]$$

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

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

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

Substituting the indices j and k in equation (1) by those in equation (6) and using the relation of $n = n_1n_2n_3$, we can derive the following equation:

$$y(k_3, k_2, k_1) = \sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3)\omega_{n_3}^{j_3k_3}\omega_{n_2n_3}^{j_2k_3}\omega_{n_2}^{j_2k_2}\omega_{n}^{j_1k_3}\omega_{n_1n_2}^{j_1k_2}\omega_{n_1}^{j_1k_1}. \qquad [9]$$

This derivation leads to the following extended three-dimensional four-step FFT:

Step 1: $x_1(j_1, j_2, k_3) = \sum_{j_3=0}^{n_3-1} x(j_1, j_2, j_3)\omega_{n_3}^{j_3k_3}$.

Step 2: $x_2(j_1, j_2, k_3) = x_1(j_1, j_2, k_3)\omega_{n_2n_3}^{j_2k_3}$.

Step 3: $x_3(k_3, j_1, j_2) = x_2(j_1, j_2, k_3)$.

Step 4: $x_4(k_3, j_1, k_2) = \sum_{j_2=0}^{n_2-1} x_3(k_3, j_1, j_2)\omega_{n_2}^{j_2k_2}$.

Step 5: $x_5(k_3, j_1, k_2) = x_4(k_3, j_1, k_2)\omega_{n}^{j_1k_3}\omega_{n_1n_2}^{j_1k_2}$.

Step 6: $x_6(k_3, k_2, j_1) = x_5(k_3, j_1, k_2)$.

Step 7: $y(k_3, k_2, k_1) = \sum_{j_1=0}^{n_1-1} x_6(k_3, k_2, j_1)\omega_{n_1}^{j_1k_1}$.

The distinctive features of the extended three-dimensional four-step FFT can be summarized as:

- If n_1, n_2 and n_3 are equal ($n_1 = n_2 = n_3 \equiv n^{1/3}$), the innermost loop length can be fixed to $n^{2/3}$. So, the three-dimensional four-step FFT algorithm is more suitable for vector processors than the "original" four-step FFT algorithm.
- The matrix transposition takes place twice.
- Three multirow-like FFTs are performed in each step, the locality of the memory references by multirow-like FFT is again low. So, the three-dimensional four-step FFT algorithm is not suitable for RISC processors as they depend on a high cache utilization to obtain high performance.

3. Parallel FFT algorithm

3.1. Algorithm (1)

The first parallel FFT algorithm we implemented is based on the six-step FFT algorithm. We will call it algorithm (1) hereafter.

Let N have two factors N_1 and N_2 ($N = N_1 \times N_2$). The original one-dimensional array $x(N)$ can be defined as a two-dimensional array $x(N_1, N_2)$ (in FORTRAN notation). On a distributed-memory parallel computer which has P processors, the array $x(N_1, N_2)$ is distributed along the first dimension N_1. If N_1 is divisible by P, each processor 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 is indicating that the data along J_r are distributed across all P processors. Here, we use the subscript r to indicate that this index belongs to dimension r. The distributed array is represented as $\hat{x}(\hat{N}_1, N_2)$. At processor m, the local index $\hat{J}_r(m)$ corresponds to the global index as the *cyclic* distribution:

$$J_r = \hat{J}_r(m) \times P + m, \quad 0 \leq m \leq P - 1, \quad 1 \leq r \leq 2. \qquad [10]$$

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

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

Step 1: Transpose

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

Step 2: Multicolumn FFTs

$$\hat{x}_2(K_2, \hat{J}_1) = \sum_{J_2=0}^{N_2-1} \hat{x}_1(J_2, \hat{J}_1)\omega_{N_2}^{J_2K_2}.$$

Step 3: Twiddle factor multiplication and transpose

$$\hat{x}_3(\hat{J}_1, P_2, \tilde{K}_2) \equiv \hat{x}_3(\hat{J}_1, K_2) = \hat{x}_2(K_2, \hat{J}_1)\omega_{N_1N_2}^{\hat{J}_1K_2}.$$

Step 4: Rearrangement

$$\hat{x}_4(\hat{J}_1, \tilde{K}_2, P_2) = \hat{x}_3(\hat{J}_1, P_2, \tilde{K}_2).$$

Step 5: All-to-all communication

$$\hat{x}_5(\tilde{J}_1, \hat{K}_2, P_1) = \hat{x}_4(\hat{J}_1, \tilde{K}_2, P_2).$$

Step 6: Transpose

$$\hat{x}_6(J_1, \hat{K}_2) \equiv \hat{x}_6(P_1, \tilde{J}_1, \hat{K}_2) = \hat{x}_5(\tilde{J}_1, \hat{K}_2, P_1).$$

Step 7: Multicolumn FFTs

$$\hat{x}_7(K_1, \hat{K}_2) = \sum_{J_1=0}^{N_1-1} \hat{x}_6(J_1, \hat{K}_2)\omega_{N_1}^{J_1K_1}.$$

Step 8: Transpose

$$\hat{y}(\hat{K}_2, K_1) = \hat{x}_7(K_1, \hat{K}_2).$$

In steps 2 and 7, multicolumn FFTs are performed along the local dimensions. Computation in step 3 is accompanied with a transposition and twiddle factor multiplication. Step 4 is a local transposition for data rearrangement.

We note that we combined some of the operations with data movements as in step 3 to gain efficiency in utilizing the memory bandwidth.

The distinctive features of the first parallel algorithm can be summarized as:

- Independent $\sqrt{N}$ point FFT is repeated $\sqrt{N}/P$ times in steps 2 and 7 for the case of $N_1 = N_2 = \sqrt{N}$.
- The all-to-all communication occurs just once. Moreover, the input data x and the output data y are both *natural order*.

If both of N_1 and N_2 are divisible by P, the workload on each processor is uniform.

For $N = 2^{20}$ point FFT, the working set size is in the order of $\sqrt{N}(=2^{10})$ and working set fits entirely into the cache. Thus, the multicolumn FFTs can be performed at high speed on cache-based RISC processors like the POWER2 processor as employed in the IBM SP2.

We now discuss the case of a (pseudo) vector processor processing element, e.g., HITACHI SR2201.

When an n point FFT is performed on a vector processor, the innermost loop length is $1, 2, \ldots, n/2$ or $n/2, n/4, \ldots, 1$. By interchanging the loop index, the average loop length can be in the order of $\sqrt{n}$.

Even if the innermost loop is interchanged for speed, the average loop length in the parallel algorithm is in the order of $N^{1/4}$ for an N point FFT because each processor performs an $N_1(= N_2 = \sqrt{N})$ point FFT repeatedly in this algorithm. So, even for a large N of 2^{32} the average loop length is 256 ($= 2^{32/4} = 2^8$) which is too short and inefficient for vector processing.

Even though pipeline startup time is very short for the processing element of the HITACHI SR2201 as shown in Figure 1 because of the pseudo-vector processing [11] feature compared to other vector processors, the minimum loop length to obtain peak performance is more than 200. So, the algorithm (1) is not suitable for the vector parallel architecture processors.

3.2. Algorithm (2)

Let us consider how to perform long-vector FFTs on the processing elements of vector-parallel processors.

We can adopt the idea of an extended three-dimensional four-step FFT as described in Section 2.

Let N have factors N_1, N_2 and N_3 ($N = N_1 \times N_2 \times N_3$). Starting with the initial data $\hat{x}(\hat{N}_1, N_2, N_3)$, the FFT can be performed according to the following steps:

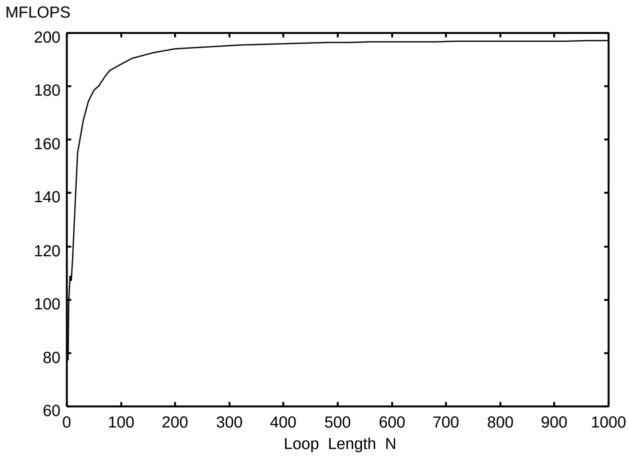

Figure 1. Performance of FFT kernel (radix-4) (HITACHI SR2201 1PE).

Step 1: Multirow-like FFTs

$$\hat{x}_1(\hat{J}_1, J_2, K_3) = \sum_{J_3=0}^{N_3-1} \hat{x}(\hat{J}_1, J_2, J_3)\omega_{N_3}^{J_3K_3}.$$

Step 2: Twiddle factor multiplication and transpose

$$\hat{x}_2(K_3, \hat{J}_1, J_2) = \hat{x}_1(\hat{J}_1, J_2, K_3)\omega_{N_2N_3}^{J_2K_3}.$$

Step 3: Multirow-like FFTs

$$\hat{x}_3(K_3, \hat{J}_1, K_2) = \sum_{J_2=0}^{N_2-1} \hat{x}_2(K_3, \hat{J}_1, J_2)\omega_{N_2}^{J_2K_2}.$$

Step 4: Twiddle factor multiplication and rearrangement

$$\begin{aligned}\hat{x}_4(P_3, \tilde{K}_3, K_2, \hat{J}_1) &\equiv \hat{x}_4(K_3, K_2, \hat{J}_1)\\ &= \hat{x}_3(K_3, \hat{J}_1, K_2)\omega_N^{\hat{J}_1(K_3+K_2N_3)}.\end{aligned}$$

Step 5: Transpose

$$\hat{x}_5(\tilde{K}_3, K_2, \hat{J}_1, P_3) = \hat{x}_4(P_3, \tilde{K}_3, K_2, \hat{J}_1).$$

Step 6: All-to-all communication

$$\hat{x}_6(\hat{K}_3, K_2, \tilde{J}_1, P_1) = \hat{x}_5(\tilde{K}_3, K_2, \hat{J}_1, P_3).$$

Step 7: Rearrangement

$$\begin{aligned}\hat{x}_7(\hat{K}_3, K_2, J_1) &\equiv \hat{x}_7(\hat{K}_3, K_2, P_1, \tilde{J}_1)\\ &= \hat{x}_6(\hat{K}_3, K_2, \tilde{J}_1, P_1).\end{aligned}$$

Step 8: Multirow-like FFTs

$$\hat{y}(\hat{K}_3, K_2, K_1) = \sum_{J_1=0}^{N_1-1} \hat{x}_7(\hat{K}_3, K_2, J_1)\omega_{N_1}^{J_1K_1}.$$

The distinctive features of this second algorithm, which we call algorithm (2) from now on, can be summarized as:

- $N^{2/3}/P$ simultaneous $N^{1/3}$ point multirow-like FFTs are performed in steps 1, 3 and 8 for the case of $N_1 = N_2 = N_3 = N^{1/3}$.
- Only one all-to-all communication is required. Moreover, the input data x and the output data y are both in *natural order*.

3.3. *Adaptability of parallel FFT algorithms to processor architecture*

In this section we want to analyze the adaptability of algorithm (1) and algorithm (2) to the type of processing element in parallel computers, e.g., processing elements of the vector processor type or of the cache-based scalar RISC processor type. In this respect the average inner loop length is particularly important. For ease of analysis, we assume N_1, N_2 and N_3 are equal ($N_1 = N_2 = N_3 = N^{1/3}$) in algorithm (2). The average loop lengths in the FFTs are $N^{2/3}/P$ in the algorithm (2), and $N^{1/4}$ in the algorithm (1). P is about 2^{10} at most and N is in the order of 2^{24} or more. The expression $N^{5/12} > P$ follows from the inequality $N^{2/3}/P > N^{1/4}$. This relation means that algorithm (2) is favorable for vector-parallel architectures with the values given above for P and N.

Next, we focus on the working set size of each processing element of the cache-based RISC processor type. We note that the working set is defined as the region of memory required in each inner loop of the FFTs. The working set size for the floating point operations in algorithms (1) and (2) is to be analyzed.

The working set size is $\sqrt{N}$ in algorithm (1), because $\sqrt{N}/P$ individual $\sqrt{N}$ point FFTs are performed independently in algorithm (1) (see Figure 2). In algorithm (2), the working set size is N/P because $N^{2/3}/P$ simultaneous $N^{1/3}$ point multirow-like FFTs are performed in algorithm (2) (see Figure 3).

Therefore algorithm (1) is favorable for parallel computers with cache-based RISC processor processing elements if $\sqrt{N} > P$, which is derived from the comparison of the working set size.

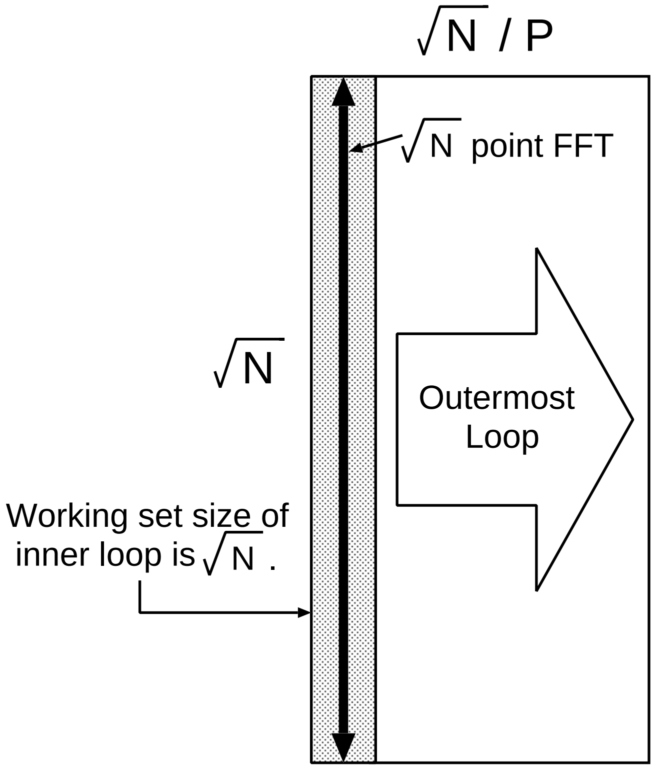

Figure 2. The shaded region is the working set of algorithm (1).

4. Radix-2, 3 and 5 FFT algorithm on a single processor

As for a single processor algorithm we used radix-2, 3 and 5 FFT algorithm based on the mixed-radix FFT algorithms of Temperton [17]. The Stockham FFT algorithm [14] was used for radix-2 FFT transforms. We modified the Stockham algorithm by including Rader's "small-*n*" transform [12] for radix-3 and radix-5.

The "small-*n*" transform, based on the WFTA (Winograd Fourier transform algorithm) [20] by Winograd, has two more additions as compared to Rader's radix-5 algorithm. By contrast, Rader's transform uses two more multiplications (see Table 1).

Therefore, Rader's "small-*n*" transform is more efficient when the CPU time for multiplication operation is equal to that of addition operation and the multiplication operation and addition operation can be performed simultaneously on the processing element as is the case on the HITACHI SR2201 and IBM SP2.

When performing a 2^p point FFT, a radix-4, or radix-8 FFT is faster than a radix-2 FFT [5] because of less memory access and a reduced number of floating point operations. In the same way, a radix-6 ($= 2 \times 3$) FFT and a radix-9 ($= 3 \times 3$) FFT, can be applied to 2^p3^q point FFTs. These higher radix FFTs reduce the number of multiplies and the total floating point operation count in the algorithm. However,

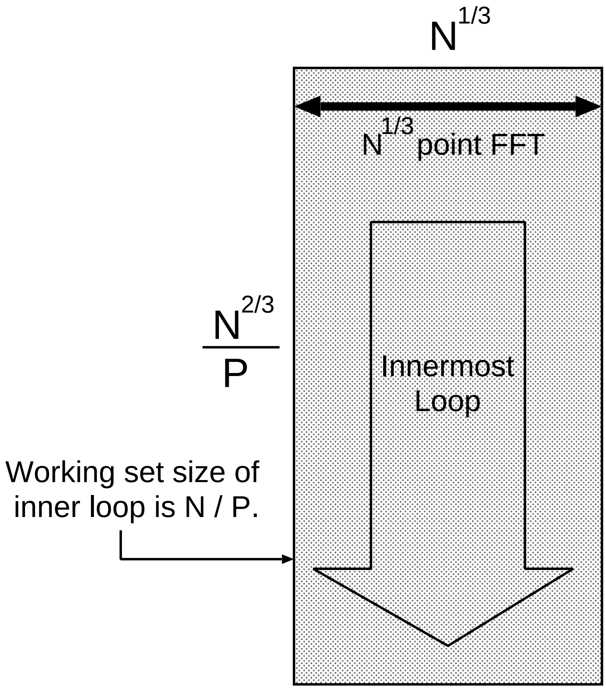

Figure 3. The shaded region is the working set of algorithm (2).

higher radix FFTs require more registers to hold intermediate results. Present day most CPUs have insufficient registers for high radix operation. For this reason, we only implemented the radix-2, 3, 4 and 5 FFT algorithms for the evaluation.

4.1. The radix-2 FFT

Let $n = 2^p$, $X_0(j) = x_j$, $0 \le j < n$ and $\omega_q = e^{-2\pi i/q}$. The radix-2 FFT algorithm can be expressed as follows:

$l = n/2$; $m = 1$
do $t = 1, p$
  do $j = 0, l - 1$

Table 1. Number of real operations for small-n transforms [16]

	Rader		Winograd	
n	Adds	Mults	Adds	Mults
2	4	0	4	0
3	12	4	12	4
4	16	0	16	0
5	32	12	34	10

$$
\begin{array}{l}
\qquad\qquad \textbf{do } k = 0,\ m-1 \\
\qquad\qquad\quad c_0 = X_{t-1}(k + jm) \\
\qquad\qquad\quad c_1 = X_{t-1}(k + jm + lm) \\
\qquad\qquad\quad X_t(k + 2jm) = c_0 + c_1 \\
\qquad\qquad\quad X_t(k + 2jm + m) = \omega_{2l}^{j}(c_0 - c_1) \\
\qquad\quad \textbf{end do} \\
\quad \textbf{end do} \\
\quad l = l/2;\ m = m * 2 \\
\textbf{end do}
\end{array}
$$

Here the variables c_0 and c_1 are temporary variables.

4.2. *The radix-3 FFT*

Let $n = 3^p$, $X_0(j) = x_j$, $0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-3 FFT algorithm can be expressed as follows:

$$
\begin{array}{l}
l = n/3;\ m = 1 \\
\textbf{do } t = 1,\ p \\
\quad \textbf{do } j = 0,\ l-1 \\
\qquad \textbf{do } k = 0,\ m-1 \\
\qquad\quad c_0 = X_{t-1}(k + jm) \\
\qquad\quad c_1 = X_{t-1}(k + jm + lm) \\
\qquad\quad c_2 = X_{t-1}(k + jm + 2lm) \\
\qquad\quad d_0 = c_1 + c_2 \\
\qquad\quad d_1 = c_0 - \frac{1}{2} d_0 \\
\qquad\quad d_2 = -i\left(\sin\dfrac{\pi}{3}\right)(c_1 - c_2) \\
\qquad\quad X_t(k + 3jm) = c_0 + d_0 \\
\qquad\quad X_t(k + 3jm + m) = \omega_{3l}^{j}(d_1 + d_2) \\
\qquad\quad X_t(k + 3jm + 2m) = \omega_{3l}^{2j}(d_1 - d_2) \\
\qquad \textbf{end do} \\
\quad \textbf{end do} \\
\quad l = l/3;\ m = m * 3 \\
\textbf{end do}
\end{array}
$$

Here the variables c_0–c_2 and d_0–d_2 are temporary variables.

4.3. *The radix-4 FFT*

Let $n = 4^p$, $X_0(j) = x_j$, $0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-4 FFT algorithm can be expressed as follows:

$$
\begin{array}{l}
l = n/4;\ m = 1 \\
\textbf{do } t = 1, p \\
\quad \textbf{do } j = 0, l-1
\end{array}
$$

```
        do k = 0, m − 1
           c_0 = X_{t−1}(k + jm)
           c_1 = X_{t−1}(k + jm + lm)
           c_2 = X_{t−1}(k + jm + 2lm)
           c_3 = X_{t−1}(k + jm + 3lm)
           d_0 = c_0 + c_2
           d_1 = c_0 − c_2
           d_2 = c_1 + c_3
           d_3 = −i(c_1 − c_3)
           X_t(k + 4jm) = d_0 + d_2
           X_t(k + 4jm + m) = ω_{4l}^{j}(d_1 + d_3)
           X_t(k + 4jm + 2m) = ω_{4l}^{2j}(d_0 − d_2)
           X_t(k + 4jm + 3m) = ω_{4l}^{3j}(d_1 − d_3)
        end do
     end do
     l = l/4; m = m * 4
  end do
```

Here the variables c_0–c_3 and d_0–d_3 are temporary variables.

4.4. *The radix-5 FFT*

Let $n = 5^p$, $X_0(j) = x_j$, $0 \le j < n$, and $\omega_q = e^{-2\pi i/q}$. The radix-5 FFT algorithm can be expressed as follows:

```
  l = n/5; m = 1
  do t = 1, p
     do j = 0, l − 1
        do k = 0, m − 1
           c_0 = X_{t−1}(k + jm)
           c_1 = X_{t−1}(k + jm + lm)
           c_2 = X_{t−1}(k + jm + 2lm)
           c_3 = X_{t−1}(k + jm + 3lm)
           c_4 = X_{t−1}(k + jm + 4lm)
           d_0 = c_1 + c_4
           d_1 = c_2 + c_3
           d_2 = (sin(2π/5))(c_1 − c_4)
           d_3 = (sin(2π/5))(c_2 − c_3)
           d_4 = d_0 + d_1
           d_5 = (√5/4)(d_0 − d_1)
           d_6 = c_0 − (1/4) d_4
           d_7 = d_6 + d_5
           d_8 = d_6 − d_5
           d_9 = −i(d_2 + (sin(π/5))/(sin(2π/5)) d_3)
           d_10 = −i((sin(π/5))/(sin(2π/5)) d_2 − d_3)
           X_t(k + 5jm) = c_0 + d_4
```

$X_t(k + 5jm + m) = \omega_{5l}^{j}(d_7 + d_9)$
$X_t(k + 5jm + 2m) = \omega_{5l}^{2j}(d_8 + d_{10})$
$X_t(k + 5jm + 3m) = \omega_{5l}^{3j}(d_8 - d_{10})$
$X_t(k + 5jm + 4m) = \omega_{5l}^{4j}(d_7 - d_9)$
end do
end do
$l = l/5;\ m = m * 5$
end do

Here the variables c_0–c_4 and d_0–d_{10} are temporary variables.

4.5. Arithmetic operation counts

Analysis of the operation count for the mixed-radix Cooley-Tukey FFT algorithm is explained in reference [17]. Here we adapt the formula given there to the case of $N = 2^p 3^q 5^r$.

The numbers of real additions $A(N)$ and multiplications $M(N)$ are given by:

$$A(N) = 2N\left(\frac{3}{2}p + \frac{8}{3}q + 4r - 1\right) + 2,$$
$$M(N) = 2N\left(p + 2q + \frac{14}{5}r - 2\right) + 4.$$

So, the total operation count is:

$$A(N) + M(N) = 2N\left(\frac{5}{2}p + \frac{14}{3}q + \frac{34}{5}r - 3\right) + 6. \qquad [11]$$

5. Experimental results of the parallel FFT

To evaluate our radix-2, 3 and 5 parallel 1-D complex FFT, p, q, r of $N = 2^p 3^q 5^r$ and the number of processors P were varied. We averaged the elapsed times obtained from 10 executions of complex forward FFTs. The parallel FFTs were performed on double precision complex data and the table for twiddle factors was prepared in advance.

A HITACHI SR2201 (1024 PEs, 256 MB per PE, 300 MFLOPS per PE, 256 GB total main memory size, communication bandwidth 300 MB/sec both way per link, and 307.2 GFLOPS peak performance) and an IBM SP2 thin-node system (32 PEs, 256 MB per PE, 266 MFLOPS per PE, 8 GB total main memory size, communication bandwidth 40 MB/sec per link, and 8.5 GFLOPS peak performance) were used as distributed-memory parallel computers in the experiment.

5.1. Experimental results on the HITACHI SR2201

Remote Direct Memory Access (RDMA) message transfer protocol [6] without memory copy was used as a communication library on the HITACHI SR2201. All routines were written in FORTRAN. The compiler used was optimized FORTRAN77 V02–05–/B of Hitachi Ltd. The optimization option, `-W0, 'opt(o(ss), split(2))'` was specified.

Tables 2 and 3 show the results of the average execution times of algorithm (1) and algorithm (2). The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average execution performance in GFLOPS. The GFLOPS value is based on equation (11) for a transform of size $N = 2^p 3^q 5^r$.

Algorithm (2) is better than algorithm (1) on the HITACHI SR2201 as is clear from Tables 2 and 3. The innermost loop length of the algorithm (2) is larger than that of the algorithm (1). The (pseudo) vector processor architecture of the HITACHI SR2201 processing element is able to take advantage of this fact.

We note that on the HITACHI SR2201 with 1024 PEs, about 130 GFLOPS was realized with size $N = 2^{30}$ in algorithm (2) as in Table 3.

Table 4 shows the results of the all-to-all communication timings on the HITACHI SR2201. The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average bandwidth in MB/sec.

5.2. Experimental results on the IBM SP2

MPI [10] was used as a communication library on IBM SP2. All routines were written in FORTRAN as on the HITACHI SR2201. The compiler used was IBM XL Fortran version 3.2. As an optimization option, `-O3 -qarch=pwr2 -qhot -qtune=pwr2` was specified. Tables 5 and 6 show the result of the average execution times of algorithm (1) and algorithm (2).

The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average execution performance in MFLOPS.

We can see that algorithm (1) is better than algorithm (2) on the IBM SP2. This is because the working set size of algorithm (1) is smaller than that of algorithm (2). Thus, the algorithm (1) is suitable for the parallel computers with cache-based scalar RISC processors as processing elements.

We note that on the IBM SP2 with 32 PEs, about 1.25 GFLOPS was realized with size $N = 2^{18} \cdot 3^2$ in algorithm (1) as shown in Table 5.

Table 2. Performance of parallel FFT algorithm (1) on the HITACHI SR2201

P	$N = 2^{20} \cdot 3 \cdot 5$		$N = 2^{21} \cdot 3^2$		$N = 2^{25} \cdot 3$		$N = 2^{22} \cdot 5^2$		$N = 2^{30}$	
	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS
8	3.6857	0.50	*	*	*	*	*	*	*	*
16	1.6233	1.13	2.1084	1.05	*	*	*	*	*	*
32	0.8178	2.25	1.0615	2.09	*	*	*	*	*	*
64	0.4165	4.42	0.5401	4.11	3.0333	4.26	3.3616	4.09	*	*
128	0.2218	8.29	0.3012	7.37	1.5341	8.42	1.6971	8.11	*	*
256	0.1295	14.20	0.2347	9.46	0.8433	15.32	0.8744	15.73	*	*
512	0.1013	18.16	0.1232	18.03	0.6775	19.07	0.5038	27.31	4.6271	33.42
1024	0.0525	35.06	0.0630	35.28	0.3406	37.93	0.3741	36.77	2.3158	66.77

*Means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient.

Table 3. Performance of parallel FFT algorithm (2) on the HITACHI SR2201

	$N = 2^{20} \cdot 3 \cdot 5$		$N = 2^{21} \cdot 3^2$		$N = 2^{25} \cdot 3$		$N = 2^{22} \cdot 5^2$		$N = 2^{30}$	
P	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS	Time	GFLOPS
8	3.1892	0.58	*	*	*	*	*	*	*	*
16	0.9153	2.01	1.0794	2.06	*	*	*	*	*	*
32	0.4788	3.84	0.5420	4.10	*	*	*	*	*	*
64	0.2466	7.46	0.2792	7.96	1.5621	8.27	1.5624	8.81	*	*
128	0.1298	14.17	0.1638	13.56	0.7975	16.20	0.8952	15.37	*	*
256	0.0720	25.55	0.0860	25.81	0.4274	30.22	0.4160	33.07	*	*
512	0.0406	45.27	0.0517	42.98	0.2541	50.85	0.2779	49.50	2.3436	65.98
1024	0.0379	48.53	0.0465	47.76	0.1359	95.04	0.1358	101.34	1.1912	129.80

*Means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient.

Table 4. All-to-all communication performance on the HITACHI SR2201

	$N = 2^{20} \cdot 3 \cdot 5$		$N = 2^{21} \cdot 3^2$		$N = 2^{25} \cdot 3$		$N = 2^{22} \cdot 5^2$		$N = 2^{30}$	
P	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec
8	0.1321	238.08	*	*	*	*	*	*	*	*
16	0.0647	243.00	0.0773	244.32	*	*	*	*	*	*
32	0.0353	222.64	0.0415	227.42	*	*	*	*	*	*
64	0.0235	167.51	0.0266	177.40	0.1078	233.37	0.1120	234.05	*	*
128	0.0242	81.24	0.0258	91.40	0.0677	185.82	0.0699	187.61	*	*
256	0.0182	53.89	0.0217	54.37	0.0584	107.67	0.0596	109.98	*	*
512	0.0122	40.39	0.0144	40.97	0.0723	43.52	0.0753	43.54	0.2278	147.27
1024	0.0081	30.35	0.0094	31.31	0.0437	35.97	0.0455	36.02	0.2439	68.80

*Means that we were not able to execute because the maximum available memory size of 224 MB per PE was insufficient.

Table 5. Performance of parallel FFT algorithm (1) on the IBM SP2

	$N = 2^{17} \cdot 3 \cdot 5$		$N = 2^{18} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{19} \cdot 5^2$		$N = 2^{25}$	
P	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
4	2.1675	92.46	2.8684	84.44	*	*	*	*	*	*
8	0.7222	277.50	0.9169	264.17	9.0235	158.04	9.2553	164.56	*	*
16	0.3133	639.67	0.3794	638.43	3.9170	364.07	3.5936	423.82	12.9452	308.45
32	0.1631	1228.75	0.1942	1247.28	1.7596	810.45	1.4650	1039.63	6.5209	612.34

*Means that we were not able to execute because the maximum available memory size of 256 MB per PE was insufficient.

Table 6. Performance of parallel FFT algorithm (2) on the IBM SP2

	$N = 2^{17} \cdot 3 \cdot 5$		$N = 2^{18} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{19} \cdot 5^2$		$N = 2^{25}$	
P	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
4	3.6755	54.53	4.7684	50.80	*	*	*	*	*	*
8	1.7492	114.57	2.2489	107.71	19.0635	74.81	15.8133	96.31	*	*
16	0.7383	271.45	1.0034	241.40	8.4410	168.94	7.4366	204.81	30.4703	131.05
32	0.2431	824.39	0.3972	609.82	3.7436	380.93	3.3936	448.80	15.0385	265.52

*Means that we were not able to execute because the maximum available memory size of 256 MB per PE was insufficient.

Table 7. All-to-all communication performance on the IBM SP2

	$N = 2^{17} \cdot 3 \cdot 5$		$N = 2^{18} \cdot 3^2$		$N = 2^{22} \cdot 3$		$N = 2^{19} \cdot 5^2$		$N = 2^{25}$	
P	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec	Time	MB/sec
4	0.2655	29.62	0.3184	29.64	*	*	*	*	*	*
8	0.1532	25.66	0.1819	25.94	0.9645	26.09	0.9963	26.31	*	*
16	0.0923	21.29	0.1094	21.57	0.5680	22.15	0.5876	22.31	1.4952	22.44
32	0.0611	16.08	0.0742	15.90	0.3616	17.40	0.3770	17.38	0.9589	17.50

*Means that we were not able to execute because the maximum available memory size of 256 MB per PE was insufficient.

Table 7 shows the results of the all-to-all communication timings on the IBM SP2. The column headed by P shows the number of processors. The next ten columns contain the average elapsed time in seconds and the average bandwidth in MB/sec.

6. Conclusion

In this paper, we explained how the radix-2, 3 and 5 parallel 1-D complex FFT was implemented on the distributed-memory parallel computers HITACHI SR2201 and IBM SP2. In our parallel FFT algorithms, since we use cyclic distribution, all-to-all communication takes place only once. Moreover, the input data and output data are in natural order.

We were able to show that the suitability of the parallel FFT algorithm depends on the CPU architecture of the processing elements of parallel computers. Our algorithms have resulted in high-performance 1-D parallel complex FFT transforms suitable for distributed-memory parallel computers.

We succeeded to get performances of about 130 GFLOPS on a 1024 PE HITACHI SR2201 and about 1.25 GFLOPS on a 32PE IBM SP2.

Implementation of the GPFA (generalized prime factor FFT algorithm) [18] which has a lower operation count than conventional FFT algorithms for any $N = 2^p 3^q 5^r$ on distributed-memory parallel computers is one of the important problems for the future.

ACKNOWLEDGMENTS

We would like to thank Dr. Aad van der Steen at the Utrecht University for valuable comments and suggestions.

References

1. R. C. Agarwal and J. W. Cooley, Vectorized mixed radix discrete Fourier transform algorithms. *Proc. IEEE*, 75:1283–1292, 1987.
2. R. C. Agarwal and F. G. Gustavson, and M. Zubair, M., A high performance parallel algorithm for 1-D FFT. *Proceedings of Supercomputing ’94*, pp. 34–40, 1994.
3. A. Averbuch, E. Gabber, and B. Gordissky, and Y. Medan, A parallel FFT on a MIMD machine. *Parallel Computing*, 15:61–74, 1990.
4. D. H. Bailey, FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.
5. G. D. Bergland, A fast Fourier transform algorithm using base 8 iterations. *Math. Comp.*, 22:275–279, 1968.
6. T. Boku, K. Itakura, H. Nakamura, and K. Nakazawa, CP-PACS: A massively parallel processor for large scale scientific calculations, *Proceedings of the 1997 International Conference on Supercomputing*, pp. 108–115, 1997.
7. J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.
8. M. Hegland, Real and complex fast Fourier transforms on the Fujitsu VPP 500. *Parallel Computing*, 22:539–553, 1996.

9. S. L. Johnsson and R. L. Krawitz, Cooley-Tukey FFT on the connection machine. *Parallel Computing*, 18, 1201–1221, 1992.
10. Message Passing Interface Forum, *MPI: A Message–Passing Interface Standard, Version 1.1*, 1995.
11. K. Nakazawa, H. Nakamura, H. Imori, and S. Kawabe, Pseudo vector processor based on register-windowed superscalar pipeline, *Proceedings of Supercomputing '92*, pp. 642–651, 1992.
12. C. M. Rader, Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, 56:1107–1108, 1968.
13. R. C. Singleton, An algorithm for computing the mixed radix fast Fourier transform. *IEEE Trans. Audio Electroacoust.*, 17:93–103, 1969.
14. P. N. Swarztrauber, FFT algorithms for vector computers. *Parallel Computing*, 1:45–63, 1984.
15. P. N. Swarztrauber, Multiprocessor FFTs, *Parallel Computing*, 5:197–210, 1987.
16. C. Temperton, A note on prime factor FFT algorithms, *J. Comput. Phys.*, 52:198–204, 1983.
17. C. Temperton, Self-sorting mixed-radix fast Fourier transforms, *J. Comput. Phys.*, 52:1–23, 1983.
18. C. Temperton, A generalized prime factor FFT algorithm for any $N = 2^p 3^q 5^r$, *SIAM J. Sci. Stat. Comput.*, 13:676–686, 1992.
19. C. Van Loan, *Computational frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, 1992.
20. S. Winograd, On computing the discrete Fourier transform, *Math. Comp.*, 32, 175–199, 1978.