Chapter 1

VECTOR-PARALLEL ALGORITHMS FOR 1-DIMENSIONAL FAST FOURIER TRANSFORM

Yusaku Yamamoto*
Dept. of Computational Science and Engineering
Nagoya University
yamamoto@na.cse.nagoya-u.ac.jp

Hiroki Kawamura
Hitachi Software Engineering Corp.
kawamu_h@itg.hitachi.co.jp

Mitsuyoshi Igai
Hitachi ULSI Systems Corp.
igai@hitachi-ul.co.jp

Abstract We review 1-dimensional FFT algorithms for distributed-memory machines with vector processing nodes. To attain high performance on this type of machine, one has to achieve both high single-processor performance and high parallel efficiency at the same time. We explain a general framework for designing 1-D FFT based on a 3-dimensional representation of the data that can satisfy both of these requirements. Among many algorithms derived from this framework, two variants are shown to be optimal from the viewpoint of both parallel performance and usability. We also introduce several ideas that further improve performance and flexibility of user interface. Numerical experiments on the Hitachi SR2201, a distributed-memory parallel machine with pseudo-vector processing nodes, show that our program can attain 48% of the peak performance when computing the FFT of 2^{26} points using 64 nodes.

*This work was done while the author was at the Central Research Laboratory, Hitachi Ltd.

Keywords: fast Fourier transform, distributed-memory, vector processor, parallel algorithm, Stockham's algorithm, cyclic distribution, block cyclic distribution, SR2201

1. Introduction

The Fourier transform is one of the most fundamental tools in science and engineering and has applications in such diverse areas as signal processing, time series analysis and solution of partial differential equations. While a straightforward computation of the Fourier transform of N points requires $O(N^2)$ work, Cooley and Tukey proposed a novel algorithm called the *fast Fourier transform* (FFT) that requires only $O(N \log N)$ work in 1965 [6]. Since then, many variants of the FFT have been proposed so far, including autosort FFT [12][15], FFT for general N [1], FFT for real data [9][15] and so on.

The FFT has a large degree of parallelism in each stage of the computation, and accordingly, its implementations on parallel machines have been well studied. See, for example, [5] [13] for implementations on shared-memory parallel machines and [2] [7] [9] [10] [13] [14] [16] for implementations on distributed-memory parallel machines.

In this article, we review 1-dimensional FFT algorithms for distributed-memory machines with (pseudo-)vector processing nodes. This type of machines have become increasingly popular recently in high-end applications such as weather forecasting and electronic structure calculation. Representative machines that fall into this category include NEC SX-7, Fujitsu VPP5000 and Hitachi SR2201 and SR8000.

To attain high performance on this type of machine, one has to achieve both high single-processor performance and high parallel efficiency at the same time. The former is realized by maximizing the length of the innermost loops, while the latter is realized when the volume and frequency of inter-processor communication is minimized. We explain a general framework for 1-dimensional FFT based on a 3-dimensional representation of the data [2][14] that satisfies both of these requirements. In designing an FFT routine using this framework, one can consider several possible variants which differ in the way the data is distributed among the processing nodes at each stage of computation. We examine these variants and point out that two of them are optimal from the viewpoint of both parallel performance and usability. They need only one global transposition and input/output data using cyclic distribution. One of them called the *variant zzx* coincides with the algorithm proposed by Takahashi [14].

Next, we introduce several ideas to further improve the performance and flexibility of user interface. Specifically, we describe methods for enhancing single-processor performance by increasing the length of the innermost loops and enhancing parallel efficiency by overlapping interprocessor communication with computation. We also propose an extension that enables the routine to input/output data using general block cyclic distributions. The block sizes for input/output data can be specified independently by the user and this flexibility is realized without increase in the amount of interprocessor communication.

The rest of this paper is organized as follows: In section 2 we describe the general framework for 1-D FFT based on the 3-dimensional representation and find out the best variants among those derived from this framework. Ideas for further improving their performance and flexibility of user interface are introduced in section 3. Section 4 shows the performance of our program on the Hitachi SR2201. Conclusions are given in the final section.

2. A general framework for 1-D FFT on vector-parallel machines

In this section, we will explain a general framework for designing a 1-dimensional FFT routine on vector-parallel machines following [2] [3] [14]. It is intended to achieve both high single-processor performance and high parallel efficiency at the same time and is based on a 3-dimensional representation of the data. To derive the framework, we start with the case of 1-D FFT algorithms for vector machines.

2.1 A 1-D FFT algorithm for vector machines based on a 2-dimensional representation of data

The discrete Fourier transform of a 1-dimensional complex sequence $\{f_0, f_1, \ldots, f_{N-1}\}$ is defined as follows:

$$c_k = \sum_{j=0}^{N-1} f_j \omega_N^{jk} \qquad (k = 0, 1, \ldots, N-1), \tag{1.1}$$

where $\omega_N = \exp(-2\pi i/N)$ and $i = \sqrt{-1}$.

When N can be factored as $N = N_x N_y$, the indices j and k can be expressed in a two-dimensional form:

$$j = j_x N_y + j_y \quad (j_x = 0, \ldots, N_x - 1, \quad j_y = 0, \ldots, N_y - 1), \tag{1.2}$$

$$k = k_x + k_y N_x \quad (k_x = 0, \ldots, N_x - 1, \quad k_y = 0, \ldots, N_y - 1). \tag{1.3}$$

Accordingly, $\{f_j\}$ and $\{c_k\}$ can be regarded as two-dimensional arrays:

$$f_{j_x,j_y} = f_{j_x N_y + j_y}, \tag{1.4}$$
$$c_{k_x,k_y} = c_{k_x + k_y N_x}. \tag{1.5}$$

Using these notations, we can rewrite eq. (1.1) as follows:

$$\begin{aligned} c_{k_x,k_y} &= \sum_{j_y=0}^{N_y-1} \sum_{j_x=0}^{N_x-1} f_{j_x,j_y} \omega_N^{(j_x N_y + j_y)(k_x + k_y N_x)} \\ &= \sum_{j_y=0}^{N_y-1} \left(\left(\sum_{j_x=0}^{N_x-1} f_{j_x,j_y} \omega_{N_x}^{j_x k_x} \right) \omega_N^{j_y k_x} \right) \omega_{N_y}^{j_y k_y}. \end{aligned} \tag{1.6}$$

This shows that the Fourier transform of $\{f_j\}$ can be computed by the following algorithm proposed by Bailey [3]:

[Algorithm 1]

1 Compute $c'_{k_x,j_y} = \sum_{j_x=0}^{N_x-1} f_{j_x,j_y} \omega_{N_x}^{j_x k_x}$ by repeating N_x-point FFT N_y times.

2 Multiply c'_{k_x,j_y} by $\omega_N^{j_y k_x}$.

3 Compute $c_{k_x,k_y} = \sum_{j_y=0}^{N_y-1} c'_{k_x,j_y} \omega_{N_y}^{j_y k_y}$ by repeating N_y-point FFT N_x times.

The factor $\omega_N^{j_y k_x}$ appearing in step 2 is called *twiddle factor* and the step 2 is called *twiddle factor multiplication*. This algorithm requires about the same amount of computational effort as the FFT of N data points. It is especially suited to vector machines if N_y and N_x are chosen so that both of them are $O(\sqrt{N})$ and the loops over j_y and k_x are used as the innermost loops in steps 1 and 3, respectively. Then the innermost loops will have a fixed length of $O(\sqrt{N})$. Moreover, the factor ω is a constant within these loops and can be loaded outside the loops.

2.2 The five-step FFT based on a 3-dimensional representation of data

In the algorithm explained in the previous subsection, we decompose the 1-D FFT into multiple FFTs of smaller size and use the multiplicity for vectorization. In the case of distributed-memory vector-parallel machines, we need another dimension to use for parallelization. To this end, we factor N as $N = N_x N_y N_z$ and introduce a three-dimensional

representation for the indices j and k:

$$\begin{aligned} j &= j_x N_y N_z + j_y N_z + j_z \qquad (1.7)\\ &\quad (j_x = 0, \ldots, N_x - 1, \quad j_y = 0, \ldots, N_y - 1, \quad j_z = 0, \ldots, N_z - 1),\\ k &= k_x + k_y N_x + k_z N_x N_y \qquad (1.8)\\ &\quad (k_x = 0, \ldots, N_x - 1, \quad k_y = 0, \ldots, N_y - 1, \quad k_z = 0, \ldots, N_z - 1). \end{aligned}$$

By regarding the input and output sequences as three-dimensional arrays f_{j_x,j_y,j_z} and c_{k_x,k_y,k_z}, we can rewrite eq. (1.1) as follows:

$$\begin{aligned} & c_{k_x,k_y,k_z} \\ = & \sum_{j_z=0}^{N_z-1} \left(\left(\sum_{j_y=0}^{N_y-1} \left(\left(\sum_{j_x=0}^{N_x-1} f_{j_x,j_y,j_z} \omega_{N_x}^{j_x k_x} \right) \omega_{N_x N_y}^{j_y k_x} \right) \omega_{N_y}^{j_y k_y} \right) \omega_N^{j_z (k_x + k_y N_x)} \right) \omega_{N_z}^{j_z k_z}. \end{aligned} \qquad (1.9)$$

This suggests the following five-step FFT [14]:

[Algorithm 2: Five-step FFT]

1 Compute $c'_{k_x,j_y,j_z} = \sum_{j_x=0}^{N_x-1} f_{j_x,j_y,j_z} \omega_{N_x}^{j_x k_x}$ by repeating N_x-point FFT $N_y N_z$ times.

2 Twiddle factor multiplication (I): multiply c'_{k_x,j_y,j_z} by $\omega_{N_x N_y}^{j_y k_x}$.

3 Compute $c''_{k_x,k_y,j_z} = \sum_{j_y=0}^{N_y-1} c'_{k_x,j_y,j_z} \omega_{N_y}^{j_y k_y}$ by repeating N_y-point FFT $N_x N_z$ times.

4 Twiddle factor multiplication (II): multiply c''_{k_x,k_y,j_z} by $\omega_N^{j_z (k_x + k_y N_x)}$.

5 Compute $c_{k_x,k_y,k_z} = \sum_{j_z=0}^{N_z-1} c''_{k_x,k_y,j_z} \omega_{N_z}^{j_z k_z}$ by repeating N_z-point FFT $N_x N_y$ times.

Because the operation in step 1 consists of $N_y N_z$ independent FFTs, we can, for example, use the index j_y for vectorization and the index j_z for parallelization. Steps 3 and 5 can be executed in a similar way.

2.3 A general framework for vector-parallel FFT based on the five-step algorithm

There are many possible ways to exploit the parallelism in Algorithm 2 for vectorization and parallelization. For example, in step 1, we can use the y-direction for vectorization and the z direction for parallelization, or vice versa. Similarly, we have two possible choices in each of steps 3

and 5. In total, there are $2^3 = 8$ possible variants. In this subsection, we clarify which variant is optimal from the viewpoint of both parallel performance and usability.

Let zxy denote the variant which uses z, x and y-directions for parallelization in steps 1, 3 and 5, respectively. In this variant, the 3-dimensional arrays are scattered along the z-direction among the nodes in step 1, while they are scattered in the x and y-direction in step 3 and 5, respectively. Accordingly, redistribution of the array is necessary after step 1 and step 3. This operation is called *global transposition*. Among the eight possible variants, yxy, yzx, yzy and zxy need two global transpositions. In contrast, variants yxx, zxx, zzx and zzy need only one global transposition and their communication overhead is half of the former ones. We can therefore expect that the latter group will achieve higher parallel performance and consider only them from now on. We illustrate the vectorization and parallelization in the zxx variant in Fig. 1.1.

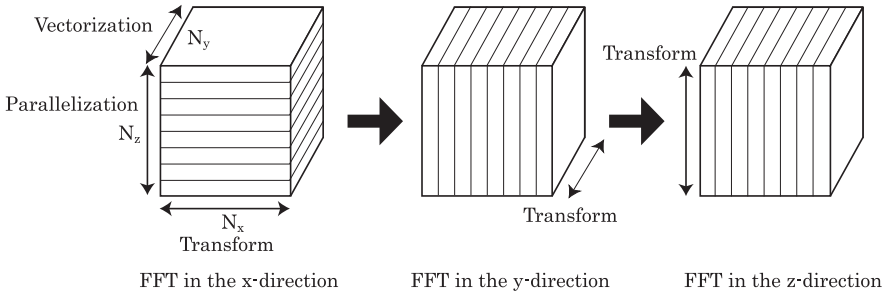

Figure 1.1. Vectorization and parallelization in the zxx variant.

Now assume that the number of points (N_x, N_y and N_z) in the direction along which the array is scattered is divisible by P, the number of processing nodes, and that we adopt cyclic distribution for scattering the data in each direction. From eqs. (1.7) and (1.8), we know that the indices j and k change contiguously when indices j_z and k_x change contiguously, respectively. As a result, the input data is distributed in a cyclic manner in j when the array is scattered in the z-direction, while it is distributed in a block cyclic manner with block size N_z when the array is scattered in the y-direction. Similarly, the output data is distributed in a cyclic manner in k when the array is scattered in the x-direction, while it is distributed in a block cyclic manner with block size N_x when the array is scattered in the y-direction. These observations are summarized in Table 1.1 for the four variants. Here, C and BC denote cyclic and block cyclic distribution, respectively.

From the table, we can see that the variants yxx and zzy use different data distributions for the input and output data, while zxx and zzx use the same (cyclic) distribution. From user's point of view, it seems more natural that the FFT routine uses the same data distribution for input and output data. Thus we can conclude that the variants zxx and zzx are the best ones judging both from parallel performance and usability among the eight variants that can be considered within our general framework. Of these two, the variant zzx has been proposed by Takahashi [14] as an algorithm suited to vector-parallel machines.

2.4 The detailed algorithm of the variant *zxx*

In this subsection, we describe a detailed algorithm of the 1-D parallel FFT based on the variant zxx. To this end, we first introduce some notations. Let $X_p^{(i)}$ denote the partial array allocated to node p at step i. We also define the indices and their ranges as follows:

$$j_x = 0, \ldots, N_x - 1, \quad j_y = 0, \ldots, N_y - 1, \quad j_z = 0, \ldots, N_z - 1, \tag{1.10}$$

$$k_x = 0, \ldots, N_x - 1, \quad k_y = 0, \ldots, N_y - 1, \quad k_z = 0, \ldots, N_z - 1, \tag{1.11}$$

$$p = 0, \ldots, P - 1, \quad q = 0, \ldots, P - 1, \tag{1.12}$$

$$j'_z = 0, \ldots, N_z/P - 1, \tag{1.13}$$

$$k'_x = 0, \ldots, N_x/P - 1. \tag{1.14}$$

Here, j'_z and k'_x are local indices corresponding to j_z and k_x, respectively, and are related to the latter in the following way:

$$j_z = j'_z P + p, \tag{1.15}$$

$$k_x = k'_x P + p, \tag{1.16}$$

where p is the node number.

Using these notations, the algorithm can be described as follows:

[Algorithm 3: Detailed algorithm of the variant zxx]

Table 1.1. Comparison of the four variants.

Step	Direction of transform	Direction of parallelization/vectorization			
		variant yxx	variant zxx	variant zzx	variant zzy
Input	–	BC	C	C	C
1	x	y/z	z/y	z/y	z/y
3	y	x/z	x/z	z/x	z/x
5	z	x/y	x/y	x/y	y/x
Output	–	C	C	C	BC

1 Data input: $X_p^{(1)}(j_y, j'_z, j_x) = f_{j_x N_y N_z + j_y N_z + j'_z P + p}$.

2 FFT in the x-direction:
$X_p^{(2)}(j_y, j'_z, k_x) = \sum_{j_x=0}^{N_x-1} X_p^{(1)}(j_y, j'_z, j_x)\omega_{N_x}^{j_x k_x}$.

3 Twiddle factor multiplication (I):
$X_p^{(3)}(j_y, j'_z, k_x) = X_p^{(2)}(j_y, j'_z, k_x)\omega_{N_x N_y}^{j_y k_x}$.

4 Data packing for global transposition:
$X_p^{(4)}(j_y, j'_z, k'_x, q) = X_p^{(3)}(j_y, j'_z, k'_x P + q)$.

5 Global transposition: $X_p^{(5)}(j_y, j'_z, k'_x, q) = X_q^{(4)}(j_y, j'_z, k'_x, p)$.

6 Data unpacking:
$X_p^{(6)}(j'_z P + q, k'_x, j_y) = X_p^{(5)}(j_y, j'_z, k'_x, q)$.

7 FFT in the y-direction:
$X_p^{(7)}(j_z, k'_x, k_y) = \sum_{j_y=0}^{N_y-1} X_p^{(6)}(j_z, k'_x, j_y)\omega_{N_y}^{j_y k_y}$.

8 Twiddle factor multiplication (II):
$X_p^{(8)}(k'_x, k_y, j_z) = X_p^{(7)}(j_z, k'_x, k_y)\omega_N^{j_z(k'_x P + p + k_y N_x)}$.

9 FFT in the z-direction:
$X_p^{(9)}(k'_x, k_y, k_z) = \sum_{j_z=0}^{N_z-1} X_p^{(8)}(k'_x, k_y, j_z)\omega_{N_z}^{j_z k_z}$.

10 Data output: $c_{k'_x P + p + k_y N_x + k_z N_x N_y} = X_p^{(9)}(k'_x, k_y, k_z)$.

In this algorithm, the most computationally intensive parts are the FFTs in steps 2, 7 and 9. The indexing scheme for array $X_p^{(i)}$ is designed so that the index with respect to which the Fourier transform is performed comes last and the loop merging techniques to be described in subsection 3.2 can be applied easily.

The computational steps of this algorithm are illustrated in Fig. 1.2 for the case of $N = 512$ and $P = 4$. Here we used the global 3-dimensional array rather than the partial 3-dimensional arrays for illustration to facilitate understanding. The numbers in the first and third 3-dimensional arrays correspond to the indices of input sequence f_j and output sequence c_k, respectively. The shaded area represents elements which are allocated to node 0, and the area enclosed by a thick line represents a set of elements used to perform a single FFT in the x, y or z-direction. It is apparent from the figure that (i) the FFTs in each direction can be computed within each node, (ii) there is only one global transposition, and (iii) the input and output data are scattered with a cyclic distribution, as required.

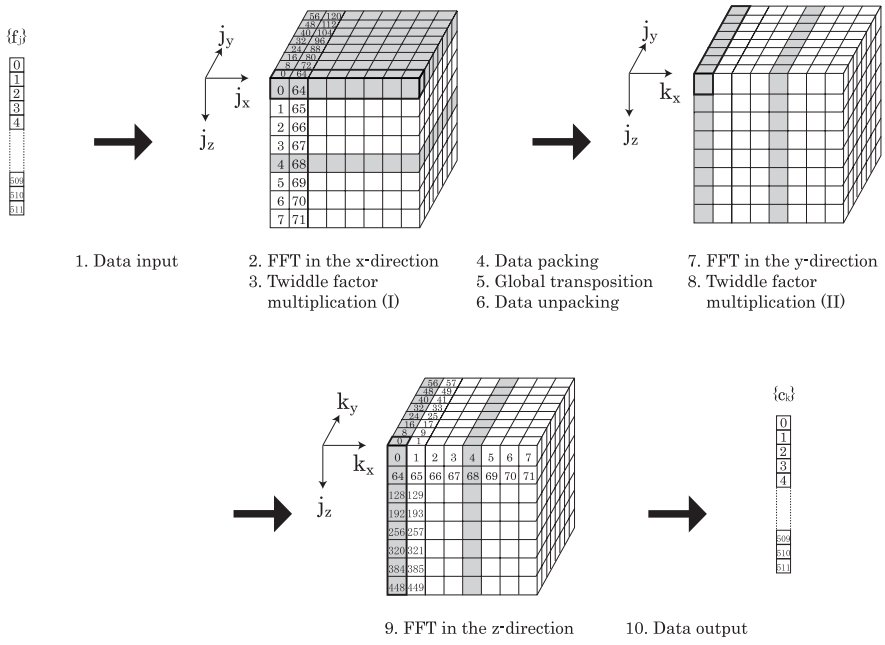

Figure 1.2. Computational steps of our FFT routine.

3. Further improvements

In this section, we introduce several ideas that can further improve the performance and usability of the 1-dimensional vector-parallel FFT described in subsection 2.4. However, all the ideas apply to algorithms based on other variants as well.

3.1 Optimization of N_x, N_y and N_z

In the derivation given in subsection 2.3, N_x, N_y and N_z for the variant zxx were assumed to be arbitrary as long as N_x and N_z are divisible by P. We can use this freedom to increase the vector length. From Table 1.1 it can be seen that y, z and y-directions are used for vectorization in the FFTs in the x, y and z-directions, respectively. So we can maximize the single-processor performance by maximizing N_y and N_z subject to the above constraints.

3.2 Increasing the loop length by loop merging

To further extend the vector length, we can use loop merging techniques [14]. First, if $N_z/P > 1$, each processing node computes FFT for

multiple values of j'_z in step 2 of Algorithm 3. So the loop over j'_z can be merged with the loop over j_y, extending the loop length to $N_y N_z/P$.

Second, we can use Stockham's algorithm [12][15] suited for vector processors in performing the FFT in each step. Let $n = 2^p$ and assume that we want to compute the FFT of an n-point sequence $Y_0(0,0), Y_0(1,0), \ldots, Y_0(n-1,0)$. This can be done with the following algorithm.

[Algorithm 4: Stockham FFT]

do $L = 0, p-1$
  $\alpha_L = 2^L$
  $\beta_L = 2^{p-L-1}$
  do $m = 0, \alpha_L - 1$
    do $l = 0, \beta_L - 1$
      $Y_{L+1}(l,m) = Y_L(l,m) + Y_L(l+\beta_L, m)\,\omega_n^{m\beta_L}$
      $Y_{L+1}(l, m+\alpha_L) = Y_L(l,m) - Y_L(l+\beta_L, m)\,\omega_n^{m\beta_L}$
    end do
  end do
end do

The result is stored in $Y_p(0,0), Y_p(0,1), \ldots, Y_p(0,n-1)$.

Notice that the ω in the innermost loop does not depend on l. This means that if we use this algorithm to compute the N_x-point FFT in step 2 of Algorithm 3, we can merge the loop over l with the loop over j_y. Combined with the loop merging mentioned above, the innermost loop length is finally extended to $N_y N_z \beta_L / P$.

Because the loop of length β_L appears α_L times in Stockham's algorithm, the average length of the innermost loops in step 2 is

$$\begin{aligned} \frac{N_y N_z}{P} \times \frac{\sum_{L=0}^{\log_2 N_x - 1} \alpha_L \beta_L}{\sum_{L=0}^{\log_2 N_x - 1} \alpha_L} &= \frac{N_y N_z}{P} \times \frac{\frac{N_x}{2} \log_2 N_x}{N_x - 1} \\ &\sim N_y N_z \log_2 N_x / 2P. \end{aligned} \tag{1.17}$$

Hence the loop length can be increased by a factor of $N_z \log_2 N_x / 2P$. Similarly, the innermost loop length in steps 7 and 9 can be extended to $N_x N_z \log_2 N_y / 2P$ and $N_x N_y \log_2 N_z / 2P$, respectively.

3.3 Overlapping the communication with computation

As we have shown in subsection 2.3, the variants yxx, zxx, zzx and zzy can attain higher parallel efficiency than other variants because they need only one global transposition. However, even one global transposition incurs considerable overhead because the amount of data each processing node has to transfer is $O(N/P)$ and is comparable to the computational work per node of $O(N \log N/P)$. This is expected to

cause a severe problem for future-generation vector-parallel computers, for the speed of interprocessor data transfer evolves much more slowly than the processor speed.

To mitigate the problem, we can construct a modified algorithm in which the data transfer is overlapped with computation. In this algorithm, the data is divided into two parts depending on whether its j_y index is even or odd and one of them is transferred while the other is computed. The outline of the algorithm can be stated as follows:

[Algorithm 5: Overlapping the communication with computation]

1 Compute the FFT in the x-direction using only those elements with even j_y. Multiply the results with twiddle factors.

2 Compute the FFT in the x-direction using only those elements with odd j_y. Multiply the results with twiddle factors. At the same time, perform global transposition operation for those elements with even j_y.

3 Compute the first $\log_2 N_y - 1$ stages of the FFT in the y-direction using only those elements with even j_y. At the same time, perform global transposition operation for those elements with odd j_y.

4 Compute the first $\log_2 N_y - 1$ stages of the FFT in the y-direction using only those elements with odd j_y.

5 Compute the last stage of the FFT in the y-direction using all the data. Multiply the results with twiddle factors.

6 Compute the FFT in the $z$direction using all the data.

This algorithm exploits the fact that in the first $\log_2 N_y - 1$ steps of the y-FFT, computations involving elements with even j_y and those with odd j_y can be done separately [15]. As a result, the overhead due to global transposition can be hidden if the computing times in steps 2 and 3 are longer than the communication time in these steps.

3.4 Use of user-specified input/output block sizes

In the variant zxx and zzx, both the input and output data are scattered among the processing nodes in a cyclic manner. However, some users may need more flexibility of data distribution. For example, block cyclic distribution is frequently used when solving linear simultaneous equations or eigenvalue problems on distributed-memory machines [4]. So if the user wants to connect the FFT routine with these routines, it

is more convenient that the FFT routine can input/output data using block cyclic data distribution with user-specified block sizes. Note that the block sizes suitable for input and output data may not be the same, so it is more desirable if they can be specified independently.

To construct an FFT routine that meets these requirements, we can use the five-step FFT as a basis. Let the block sizes for input and output data be L_1 and L_2, respectively, and assume that N_z and N_x are divisible by $L_1 * P$ and $L_2 * P$, respectively. Now we scatter the three-dimensional array along the z-direction in steps 1 and 2 of Algorithm 2 using block cyclic distribution of block size L_1, and along the x-direction in steps 3-5 using block cyclic distribution of block size L_2. Then, from eq. (1.7), we know that the whole input sequence of length N is scattered with a block cyclic distribution of block size L_1. Likewise, the whole output sequence is scattered with a block cyclic distribution of block size L_2. This method requires only one global transposition like the variant zxx and leaves the room for vectorization using indices j_y, j_z and j_y in steps 1, 3 and 5, respectively.

One shortcoming of this approach is that N_y, which is the length of the innermost loops in steps 1 and 5, tends to become small because N_x and N_z need to be large enough to be multiples of $L_1 * P$ and $L_2 * P$, respectively. We can mitigate this problem by using loop merging techniques described in subsection 3.2. The readers are referred to [16] for more detailed description and performance evaluation of this approach.

4. Experimental results

We implemented Algorithm 3 on the Hitachi SR2201 [8] and evaluated its performance. The SR2201 is a distributed-memory parallel machine with pseudo-vector processing nodes. Each node consists of a RISC processor with a pseudo-vector mechanism [11], which preloads the data from pipelined main memory to on-chip special register bank at a rate of 1 word per cycle. One node has peak performance of 300MFLOPS and 256MB of main memory. The nodes are connected via a multi-dimensional crossbar network, which enables all-to-all communication among P nodes to be done in $P - 1$ steps without contention [17].

Our FFT routine is written in FORTRAN and inter-processor communication is done using remote DMA, which enables data stored in the main memory of one node to be transferred directly to the main memory of another node without buffering. The FFT in the x, y and z direction in steps 2, 7 and 9 is performed using Stockham's radix 4 FFT [15], a variant of Algorithm 4 which saves both computational work and memory access by computing Y_{L+2} directly from Y_L.

To measure the performance of our FFT routine, we varied the problem size per node, N/P, from 2^{14} to 2^{20}. As for the number of nodes P, we measured the performance in two cases, namely, $P = 1$ and $P = 64$. We adopted optimization of N_x, N_y and N_z introduced in subsection 3.1, but did not incorporate the loop merging technique and overlapping of communication and computation. We didn't adopt the modifications to make the input/output block sizes user-specifiable, either. Readers interested in the last point are referred to the performance results given in [16]. The ω's used in the FFT and twiddle factor multiplication are pre-computed, so the time for computing them is not included in the execution time to be reported below.

Table 1.2 and Fig. 1.3shows the execution time and the performance of our routine. From these results, we can see that (i) the maximum performance on a single node is 176MFLOPS, which is more than 58% of the peak performance and (ii) parallel performance on 64 nodes is 9.18GFLOPS, which is about 48% of the peak performance.

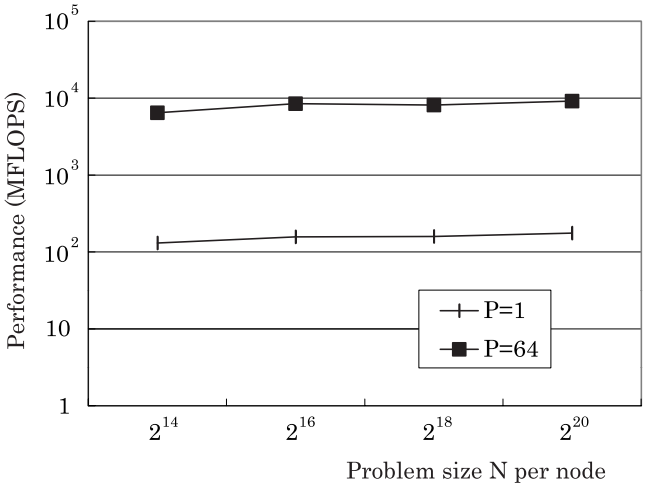

Figure 1.3. Performance results for $P = 1$ and $P = 64$.

Table 1.2. Performance results for $P = 1$ and $P = 64$.

P	$N/P = 2^{14}$	$N/P = 2^{16}$	$N/P = 2^{18}$	$N/P = 2^{20}$
1	130.86MF	157.07MF	158.82MF	176.11MF
	43.33%	52.35%	52.94%	58.7%
64	6477.44MF	8477.41MF	8146.88MF	9175.07MF
	33.73%	44.15%	42.43%	47.78%

From these results, we can conclude that the FFT algorithm described in this article can attain high performance on a (pseudo-)vector-parallel machine.

5. Conclusion

In this article, we reviewed 1-dimensional FFT algorithms for distributed-memory machines with vector processing nodes. We explained a general framework for designing 1-D FFT based on a 3-dimensional representation of the data that can achieve both high single-processor performance and high parallel efficiency at the same time. Among the many algorithms derived from this framework, we showed that two variants are optimal from the viewpoint of both parallel performance and usability. We also introduced several ideas that further improve performance and flexibility of user interface.

We implemented the algorithm on the Hitachi SR2201, a distributed-memory parallel machine with pseudo-vector processing nodes, and obtained the performance of 9175 MFLOPS, or 48% of the peak performance, when transforming 2^{26} point data on 64 nodes. It should be easy to adapt our method to other similar vector-parallel machines.

Acknowledgments

We would like to thank Dr. Mamoru Sugie at the Central Research Laboratory, Hitachi Ltd. for many valuable comments on the first version of this paper. We are also grateful to Mr. Nobuhiro Ioki and Mr. Shinichi Tanaka at the Software Development Division of Hitachi Ltd. for providing the environments for our computer experiments.

References

[1] R. C. Agarwal and J. W. Cooley: Vectorized Mixed Radix Discrete Fourier Transform Algorithms, *Proc. of IEEE*, Vol. 75, No. 9, pp. 1283-1292 (1987).

[2] R. C. Agarwal, F. G. Gustavson and M. Zubair: A High Prformance Parallel Algorithm for 1-D FFT, *Proc. of Supercomputing '94*, pp. 34-40 (1994).

[3] D. H. Bailey: FFTs in External or Hierarchical Memory, *The Journal of Supercomputing*, Vol. 4, pp. 23-35 (1990).

[4] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. Whaley: ScaLAPACK User's Guide, SIAM, Philadelphia, PA, 1997.

[5] D. A. Carlson: Ultrahigh-Performance FFTs for the Cray-2 and Cray Y-MP Supercomputers, *Journal of Supercomputing*, Vol. 6, pp. 107-116 (1992).

[6] J. W. Cooley and J. W. Tukey: An Algorithm for the Machine Calculation of Complex Fourier Series, *Mathematics of Computation*, Vol. 19, pp. 297-301 (1965).

[7] A. Dubey, M. Zubair and C. E. Grosch: A General Purpose Subroutine for Fast Fourier Transform on a Distributed Memory Parallel Machine, *Parallel Computing*, Vol. 20, pp. 1697-1710 (1994).

[8] H. Fujii, Y. Yasuda, H. Akashi, Y. Inagami, M. Koga, O. Ishihara, M. Kashiyama, H. Wada and T. Sumimoto: Architecture and Performance of the Hitachi SR2201 Massively Parallel Processor System, *Proc. of IPPS '97*, pp. 233-241, 1997.

[9] M. Hegland: Real and Complex Fast Fourier Transforms on the Fujitsu VPP500, *Parallel Computing*, Vol. 22, pp. 539-553 (1996).

[10] S. L. Johnson and R. L. Krawitz: Cooley-Tukey FFT on the Connection Machine, *Parallel Computing*, Vol. 18, pp. 1201-1221 (1992).

[11] K. Nakazawa, H. Nakamura, H. Imori and S. Kawabe: Pseudo Vector Processor Based on Register-Windowed Superscalar Pipeline, *Proc. of Supercomputing '92*, pp. 642-651 (1992).

[12] P. N. Swarztrauber: FFT Algorithms for Vector Computers, *Parallel Computing*, Vol. 1, pp. 45-63 (1984).

[13] P. N. Swarztrauber: Multiprocessor FFTs, *Parallel Computing*, Vol. 5, pp. 197-210 (1987).

[14] D. Takahashi: Parallel FFT Algorithms for the Distributed-Memory Parallel Computer Hitachi SR8000, *Proc. of JSPP2000*, pp. 91-98, 2000 (in Japanese).

[15] C. Van Loan: Computational Frameworks for the Fast Fourier Transform, SIAM Press, Philadelphia, PA (1992).

[16] Y. Yamamoto, M. Igai and K. Naono: A Vector-Parallel FFT with a User-Specifiable Data Distribution Scheme, in M. Guo and L. T. Yang, eds., *Parallel and Distributed Processing and Applications*, Lecture Notes in Computer Science 2745, Springer-Verlag, pp. 362-374, 2003.

[17] Y. Yasuda, H. Fujii, H. Akashi, Y. Inagami, T. Tanaka, J. Nakagoshi, H. Wada and T. Sumimoto: Deadlock-Free Fault-Tolerant Routing in the Multi-Dimensional Crossbar Network and its Implementation for the Hitachi SR2201, *Proc. of IPPS '97*, pp. 346-352, 1997.