IPSJ Transactions on Advanced Computing Systems Vol. 4 No. 3 9–21 (May 2011)

Regular Paper

On Auto-tuned Pre/postprocessing for the Singular Value Decomposition of Dense Square Matrices

Hiroki Toyokawa,[†1] Kinji Kimura,[†2]
Yusaku Yamamoto,[†3] Masami Takata,[†4]
Akira Ajisaka[†2] and Yoshimasa Nakamura[†2]

An auto-tuning technique is devised for fast pre/postprocessing for the singular value decomposition of dense square matrices with the Dongarra or the Bischof-Murata algorithms. The computation speed of these two algorithms varies depending on a parameter and specification of computers. By dividing these algorithms into several parts and by modeling each of them, we can estimate their computation times accurately. This enables us to choose an optimal parameter and the faster algorithm prior to execution. Consequently the pre/postprocessing is done faster and the singular value decomposition is applied faster to dense square matrices. Numerical experiments show the effectiveness of the proposed auto-tuning function. The I-SVD library, which incorporates this auto-tuning function, has been published.

1. Introduction

The I-SVD algorithm [1)–4)] is a fast and accurate algorithm for singular value decomposition. The original I-SVD algorithm can be applied only to bidiagonal matrices. Before we apply the I-SVD algorithm to a dense square matrix, a *preprocessing* stage that transforms a dense square matrix to a bidiagonal matrix should be applied to the matrix. In addition, a *postprocessing* stage is necessary, which transforms the singular vectors of the bidiagonal matrix to those of the original matrix.

The Dongarra [5)] algorithm and the combination of the Bischof algorithm [6)] and the Murata [7)] algorithm are fast pre/postprocessing. In this paper, we call the combination of the Bischof algorithm and the Murata algorithm as *the Bischof-Murata algorithm*. The computation time of the Bischof-Murata algorithm varies according to the parameter called *band width* L. Therefore we have to choose a proper value of the parameter L for fast computation. Furthermore the computational cost of the Bischof-Murata algorithm is larger than that of the Dongarra algorithm. However the Bischof-Murata algorithm consists of faster level-3 BLAS operations, while the Dongarra algorithm includes slower level-2 BLAS operations. Since level-3 BLAS operations run much faster than level-2 BLAS operations on modern processors [8),9)], the Bischof-Murata algorithm is faster than the Dongarra algorithm in some conditions.

A technique to select better algorithms and parameters is called *auto-tuning*. For example Dackland, et al. [10)] and Cuenca, et al. [11)] propose a way for modeling the performance of a matrix computation algorithm. For performance modeling, polynomial functions, spline functions [12)] and piecewise linear functions [13)] are used. However, it is difficult to estimate the whole computation time with functions of one kind if the algorithm is complicated and composed of several parts. In that case, the algorithms should be divided into several parts and modeling of each of them should be done separately.

In this article, we divide the Bischof-Murata algorithm into three parts to estimate its computation time accurately. We determine different function and make estimation curves for the three phases individually. Furthermore, for the Bischof-Murata algorithm, estimation curves depending on the parameter L are also created. Based on these curves, a faster algorithm and a better value of the parameter L can be determined.

2. Preprocessing and Postprocessing

In this section, we explain the Bischof-Murata and the Dongarra algorithms briefly, focusing on the kind and size of BLAS routines used in the algorithms.

2.1 The Bischof-Murata Algorithm

In the preprocessing stage of the Bischof-Murata algorithm, a dense square matrix A is transformed to a bidiagonal matrix B in two steps. In the following, we denote the size of A by N. In the first step, A is transformed to a lower triangular band matrix C with band width L using the Bischof algorithm [6)]. We

†1 NS Solutions Corporation
†2 Graduate School of Informatics, Kyoto University
†3 Graduate School of System Informatics, Kobe University
†4 Graduate School of Humanities and Sciences, Nara Women's University

can choose the bandwidth L ($1 \leq L \leq N-1$) freely. We call this step *Bischof preprocessing*. In the second step, the matrix C is transformed to the bidiagonal matrix B with the Murata algorithm[7]. This step is called *Murata preprocessing*. The singular values of B and C are same as those of A.

The postprocessing stage of the Bischof-Murata algorithm also consists of two steps. In the first step, called *Murata postprocessing*, the singular vectors of B are transformed to the singular vectors of C. In the second step, called *Bischof postprocessing*, the singular vectors of C are transformed to those of A.

In the following, we explain each of these two preprocessing and two postprocessing steps. For simplicity, we assume that N is divisible by L and define $N' = N/L$. We regard A as an $(N/L) \times (N/L)$ block matrix whose block size is $L \times L$, and denote the submatrix of A consisting of the I_1th through the I_2th block rows and the J_1th through the J_2th block columns by $A_{I_1:I_2,J_1:J_2}$. $A_{I_1:_2,*}$ means the submatrix of A consisting of the I_1th through the I_2th block rows and all the block columns. The identity matrix of order k is denoted by I_k and the transpose of matrix A is denoted by A^T.

2.1.1 Bischof Preprocessing

The algorithm of Bischof preprocessing is shown as **Algorithm 1**. In this algorithm, we transform the input dense matrix A to a lower triangular band matrix by eliminating its block rows and block columns step by step. Here, for an $M \times L$ matrix X, **BlockHouse**(X) denotes a subroutine that computes a block Householder transformation[14] $I_M - YTY^T$ ($Y \in \mathbf{R}^{M\times L}$, $T \in \mathbf{R}^{L\times L}$) that transforms X to a matrix whose first block is upper triangular and all the subsequent blocks are zero. After we have executed Algorithm 1, the matrix A contains a lower triangular band matrix, which we denote by C.

The computationally heaviest parts of Algorithm 1 are applications of block Householder transformations, in particular, Steps 3, 5, 7 and 9. As can be seen from the algorithm, these computations can be done with matrix-matrix multiplication, or BLAS routine DGEMM, which belongs to level-3 BLAS. For example, Step 3 is a multiplication of a $(N'-K+1)L \times (N'-K+1)L$ matrix and a $(N'-K+1)L \times L$ matrix, while Step 5 is a multiplication of a $(N'-K+1)L \times L$ matrix and a $L \times (N'-K+1)L$ matrix. Both of these matrix multiplications require $O(N^2L)$ work. By summing up these computational work, we know that the Bischof preprocessing requires $8N^3/3$ floating-point operations in the form of DGEMM. Other computations, such as the construction of block Householder transformations and computation of Steps 4 and 8, require only $O(N^2L)$ work in total, which is much smaller than the work for DGEMM when $L \ll N$.

```
[Algorithm 1: Bischof preprocessing]
1:  for $K = 1, N'$ do
2:      $(Y_R^{(K)}, T_R^{(K)}) = \mathbf{BlockHouse}(A^T_{K,K:N'})$
3:      $F = A_{K:N',K:N'} Y_R^{(K)}$
4:      $G = F T_R^{(K)}$
5:      $A_{K:N',K:N'} := A_{K:N',K:N'} - G(Y_R^{(K)})^T$
6:      $(Y_L^{(K)}, T_L^{(K)}) = \mathbf{BlockHouse}(A_{K+1:N',K})$
7:      $F^T = (Y_L^{(K)})^T A_{K+1:N',K:N'}$
8:      $G^T = (T_L^{(K)})^T F^T$
9:      $A_{K+1:N',K:N'} := A_{K+1:N',K:N'}$
                            $-Y_L^{(K)} G^T$
10: end for
```

2.1.2 Murata Preprocessing

In Murata preprocessing, we transform the lower triangular band matrix C to a bidiagonal matrix B step by step. This is carried out by applying small Householder transformations of length L to C repeatedly. The number of Householder transformations is N^2/L and the computation is done with the level-2 BLAS. The total computational work is $8LN^2$.

2.1.3 Murata Postprocessing

In Murata postprocessing, we apply the Householder transformations used in Murata preprocessing to singular vectors of B in reversed order. To speed up the computation, we combine L Householder transformations using a technique called compact WY representation[14] and apply them at once using matrix-matrix multiplication. The combining process is called *Creating compact WY representation*, while the applying phase is called *Applying compact WY representation*.

In applying compact WY representation, we perform multiplication of a $2L \times 2L$ matrix and a $2L \times N$ matrix $\left(\frac{N}{L}\right)^2 + \frac{N}{L}$ times using DGEMM. The total computational cost for this is $4N^3$. In addition, we need to do some data copy operations, copying an $L \times N$ matrix $\left(\frac{N}{L}\right)^2 + \frac{N}{L}$ times. This is done with the BLAS routine DCOPY. Note that DCOPY belongs to neither level-1, 2 nor 3 BLAS, because it does not perform any computation. Creating compact WY representation requires $O(N^2L)$ work, which is much smaller than the work for applying compact WY representation.

2.1.4 Bischof Postprocessing

The algorithm of Bischof postprocessing is shown as **Algorithm 2**. In this algorithm, we apply the block Householder transformations generated in Bischof preprocessing to the singular vectors of C in reversed order. In the algorithm, V and U denote matrices whose column vectors are the right and left singular vectors of C, respectively. The matrices $Y_R^{(K)}, T_R^{(K)}, Y_L^{(K)}$ and $T_L^{(K)}$ represent block Householder transformations generated in Algorithm 1.

```
[Algorithm 2: Bischof postprocessing]
1: for $K = N', 1, -1$ do
2:   $W^T = (Y_R^{(K)})^T V_{K:N',*}$
3:   $Z^T = T_R^{(K)} W^T$
4:   $V_{K:N',*} := V_{K:N',*} - Y_R^{(K)} Z^T$
5:   $W^T = (Y_L^{(K)})^T U_{K+1:N',*}$
6:   $Z^T = T_L^{(K)} W^T$
7:   $U_{K+1:N',*} := U_{K+1:N',*} - Y_L^{(K)} Z^T$
8: end for
```

As can be seen from Algorithm 2, the algorithm consists entirely of the level-3 BLAS, or DGEMM. The computationally heaviest parts are Steps 2, 4, 5 and 7. Step 2 is a DGEMM of a $L \times (N' - K + 1)L$ matrix and a $(N' - K + 1)L \times N$ matrix, while Step 4 is a DGEMM of a $(N' - K + 1)L \times L$ matrix and a $L \times N$ matrix. Both of these require $O(N^2L)$ work. Steps 5 and 7 are similar to Steps 2 and 4, respectively. The total computational work of Algorithm 2 is $4N^3$, all of which are executed as DGEMM.

2.2 The Dongarra Algorithm

The Dongarra algorithm[5] transforms a dense square matrix A into a bidiagonal matrix B directly. This algorithm is implemented in the matrix library LAPACK[15] and is widely used. In this subsection, we will explain the preprocessing stage and the postprocessing stage of the Dongarra algorithm briefly. We adopt the MATLAB notation and denote the submatrix of A consisting of the i_1th through the i_2th rows and the j_1th through the j_2th columns by $A_{i_1:i_2,j_1:j_2}$.

2.2.1 Dongarra Preprocessing

Like the Householder algorithm for bidiagonalization[8], the Dongarra algorithm transforms the input matrix to a bidiagonal matrix step by step, eliminating one row and one column at each step using Householder transformations.

At the step k of Dongarra preprocessing ($1 \le k \le N-1$), we construct a Householder transformation $I_{N-k+1} - t_R^{(k)} \mathbf{y}_R^{(k)} (\mathbf{y}_R^{(k)})^T$ that transforms $A_{k,k:N}^T$ to a vector whose first element is nonzero and all the other elements are zero. By applying this Householder transformation to $A_{k:N,k:N}$ from the left, as shown below, we can eliminate the upper triangular elements of the kth row of A.

$$\mathbf{f} = A_{k:N,k:N} \mathbf{y}_R^{(k)}, \tag{1}$$

$$\mathbf{g}_R^{(k)} = t_R^{(k)} \mathbf{f}, \tag{2}$$

$$A_{k:N,k:N} = A_{k:N,k:N} - \mathbf{g}_R^{(k)} (\mathbf{y}_R^{(k)})^T. \tag{3}$$

However, in the Dongarra algorithm, we compute only Eqs. (1) and (2) at step k. We wait performing the update operation (3) until k is a multiple of M_1, where M_1 is a positive integer called *block size*, and then update A using M_1 vectors $\mathbf{y}_R^{(k)}$ and $\mathbf{g}_R^{(k)}$ ($k = (k'-1)M_1, \ldots, k'M_1$ for some k'). The update operation can be done with DGEMM. On the other hand, Eq. (1) requires matrix-vector multiplication, or DGEMV, which is a level-2 BLAS operation. The elimination of columns are done in the same way. In summary, Dongarra preprocessing consists of both DGEMM and DGEMV, each of which requires $4N^3/3$ computational work.

2.2.2 Dongarra Postprocessing

In Dongarra postprocessing, we apply the Householder transformations generated in the preprocessing stage to singular vectors of B in reversed order. In applying the transformations, we combine M_2 consecutive Householder transformations, where M_2 is some positive integer, and construct a compact WY representation. Then, application of the M_2 Householder transformations can be

Table 1 Computational cost.

	BLAS	Dongarra	Bischof	Murata
Pre	DGEMM	$4N^3/3$	$8N^3/3$	
	DGEMV	$4N^3/3$		$8LN^2$
Post	DGEMM	$4N^3$	$4N^3$	$4N^3$

(in the case of $L \ll N$)

done at once using DGEMM. The sizes of the DGEMM's are almost the same as those in Bischof postprocessing, and the total computational work is $4N^3$.

2.2.3 The Optimal Block Sizes

In principle, we should discuss optimizing the block sizes M_1 and M_2 in the Dongarra algorithm. However, the optimal values of M_1 and M_2 have been discovered for various computers and matrix sizes, and these values are used in the LAPACK bidiagonalization routine automatically. We therefore adopt these values and do not discuss optimizing M_1 and M_2 in this article.

2.3 Summary of the Bischof-Murata and Dongarra Algorithms

We summarize the BLAS routines used in Dongarra/Bischof-Murata pre/postprocessing and their computational cost in **Table 1**.

3. Auto-Tuning

According to Table 1, it seems that the Dongarra algorithm is always faster than the Bischof-Murata algorithm for all (N, L), because its computational work is smaller. However this is not the case.

The Dongarra algorithm contains level-2 BLAS operations, DGEMV, which are not fast. On the contrary, the Bischof-Murata algorithm consists of level-3 BLAS operations, DGEMM, which is fast using cache memory. Depending on the band width L, the Bischof-Murata algorithm might be faster than the Dongarra algorithm. The faster algorithm depends on the value of L, the matrix size N, and the computational environment.

In this article, a function to determine the faster algorithm and a good parameter L automatically is proposed. This funtion is called *Auto-tuning*. The detail of the proposed auto-tuning function is described in this section.

3.1 Estimating the Computation Time of the Dongarra Algorithm

According to Table 1, the total computation work of the Dongarra algorithm is $O(N^3)$. Thus it is natural to model the computation time $E_{\mathrm{D}}(N)$ of the Dongarra algorithm by a cubic function in N. We did a preliminary experiment to see whether this gives a reasonably accurate model.

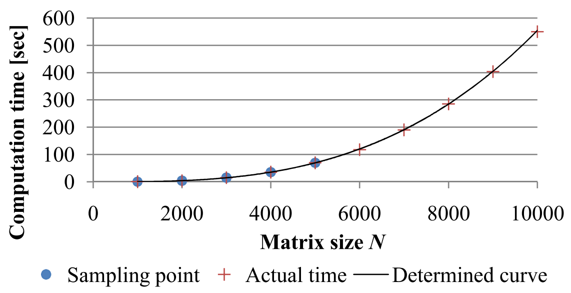


Fig. 1 Computation time of the Dongarra algorithm.

Table 2 Specification of the Test Computer.

Computer	G	I
CPU	Intel Core i7 X980 3.33 GHz (6 cores × 1 processor)	Intel Core 2 Duo E6700 2.66 GHz (2 cores×1 processor)
Cache	12 MBytes L3	4 MBytes L2
Memory	24 GBytes	8 GBytes
Memory bandwidth	25.6 GBytes/s	8.5 Gbytes/s
OS	Fedora Linux 13	Fedora Linux 12
(Kernel)	(2.6.34.6-47.fc13.x86_64)	(2.6.32.16-150.fc12.x86_64)
Compiler	icpc & ifort 11.1 (-xsse4.2 -ipo -O3 option)	icpc & ifort 11.1 (-xsse3 -ipo -O3 option)
BLAS	GotoBLAS2-1.13	

In the experiment, computation time for the Dongarra algorithm of dense square matrices ($N = 1{,}000, 2{,}000, \ldots, 10{,}000$) are measured (actual time in **Fig. 1**). Then we regard 5 points ($N = 1{,}000, 2{,}000, \ldots, 5{,}000$) as sampling points, and a cubic function is determined to fit the sampling points by the least squares method. The curve is shown in Fig. 1 as "Determined curve". The experiment is done on the computer G described in **Table 2**.

The result of the experiment shows that the actual computation time of the Dongarra algorithm can be modeled accurately with the curve. Therefore we estimate the computation time of the Dongarra algorithm with a cubic function of N determined by the least squares method.

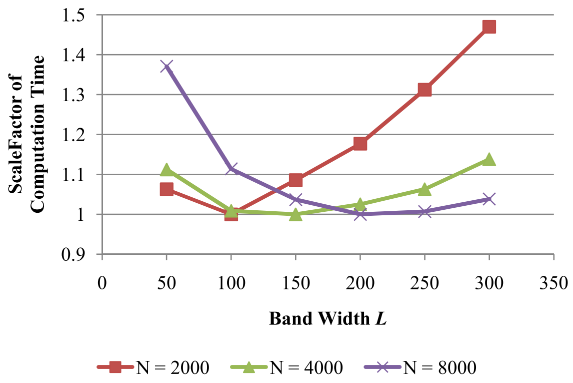


Fig. 2 Actual computation time of the Bischof-Murata algorithm (vertical axis: scale factor = time / $\min_L$(time), $N = 4{,}000$, computer: G).

3.2 Estimating the Computation Time of the Bischof-Murata Algorithm

According to Table 1, it is tempting to model the total computation time of the Bischof-Murata algorithm $E_{\mathrm{BM}}(N, L)$ with the expression as follows:

$$E_{\mathrm{BM}}(N, L) \propto \frac{8}{3}N^3 + 8LN^2 + 8N^3. \qquad (4)$$

This function is cubic in N and linear in L.

3.2.1 Practical Computation Time of the Bischof-Murata Algorithm

The actual computation time of the Bischof-Murata algorithm is shown in **Fig. 2**. Dense square matrices of size $2{,}000 \times 2{,}000$, $4{,}000 \times 4{,}000$ and $8{,}000 \times 8{,}000$ are prepared for the test. The computer G described in Table 2 is used.

Though Table 1 suggests that $E_{\mathrm{BM}}(N, L)$ is a linear function of L if N is fixed, the actual result shown in the figure does not match any linear function of L.

This difference occurs because of the cache memory. If the matrix size N is neither too large nor too small, the data transfer time is reduced by the cache memory. But if L is smaller, the cache memory does not work efficiently, consequently the computation is done slowly. This utilization of cache memory makes the functional form of $E_{\mathrm{BM}}(N, L)$ complicated. It is difficult to analyze the effect of cache memory analytically.

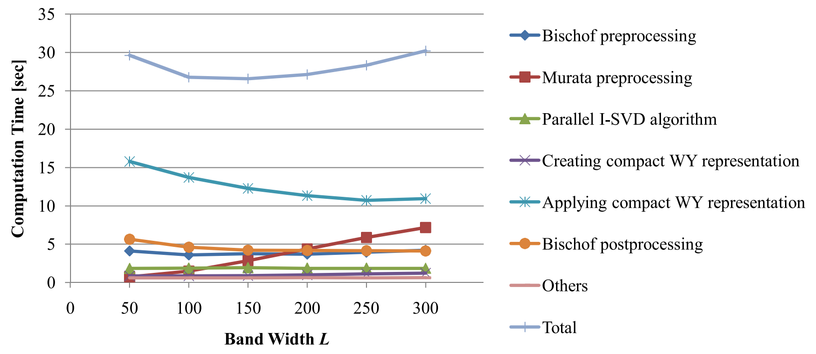


Fig. 3 Classification of the computation time of the Bischof-Murata algorithm ($N = 4{,}000$, computer: G).

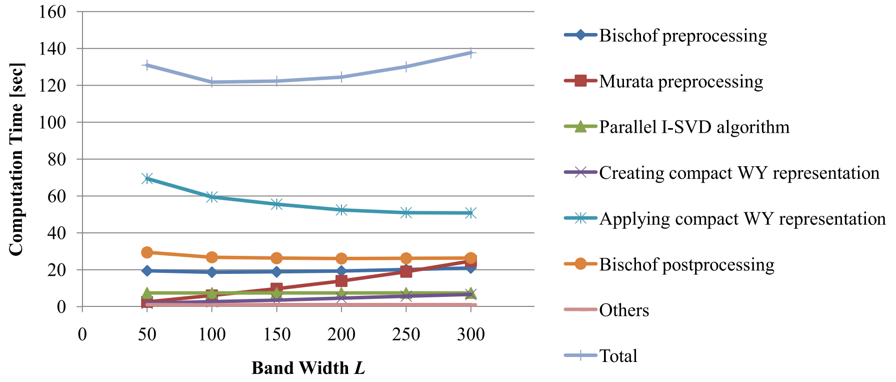


Fig. 4 Classification of the computation time of the Bischof-Murata algorithm ($N = 4{,}000$, computer: I).

3.2.2 Breakdown of the Computation Time of the Bischof-Murata Algorithm

Figures 3 and **4** shows the breakdown of computation time of the Bischof-Murata algorithm for a $4{,}000 \times 4{,}000$ matrix on the computer G and I.

According to these figures, the relation between L and the computation time of each part can be classified into three patterns as follows:

(1) The part whose computation time increases in proportion to L.

(2) The part whose computation time decreases gradually as L increases.

(3) The part whose computation time decreases rapidly as L increases and then stays constant.

The Murata preprocessing is classified as the first. Applying the compact WY representation is classified as the second. The other processes are classified as the third.

We divide the Bischof-Murata algorithm into the three parts as described above. First we estimate the computation time of these three parts individually. After estimating the time of those 3 parts accurately, we estimate the total computation time by adding the three estimated times.

$$E_{\mathrm{BM}}(N,L) = E_{\mathrm{BM}}^{(1)}(N,L) + E_{\mathrm{BM}}^{(2)}(N,L) + E_{\mathrm{BM}}^{(3)}(N,L), \qquad (5)$$

where $E_{\mathrm{BM}}^{(1)}(N,L)$ means the computation time for the Murata preprocessing, and $E_{\mathrm{BM}}^{(2)}(N,L)$ means the computation time for applying the compact WY representation. $E_{\mathrm{BM}}^{(3)}(N,L)$ means the computation time for the other processes.

3.2.2.1 The Murata Preprocessing

The computational cost of the Murata preprocessing is $O(LN^2)$. This preprocessing mainly consists of the level-2 BLAS operations, which cannot utilize cache memory well. For sufficiently large N, the matrix data is always out of cache, so the computational speed measured in GFLOPS is almost independent of L. Thus the computation time is proportional to the computational cost. Therefore we adopt the following function to estimate the computation time of the Murata preprocessing:

$$E_{\mathrm{BM}}^{(1)}(N,L) = cLN^2.$$

We determine the coefficient c with the least squares method.

3.2.2.2 Applying the Compact WY Representation

The process of applying the compact WY representation mainly consists of matrix-matrix product and data copy operations with DGEMM and DCOPY routines in the BLAS. If L is small, these routines do not work efficiently. But the performance is expected to improve as L increases.

As explained in Section 2.1.3, DGEMM and DCOPY routines are called $\left(\frac{N}{L}\right)^2 + \frac{N}{L}$ times for applying the compact WY representation. Therefore the theoretical computation time is described as follows:

$$E_{\mathrm{BM}}^{(2)}(N,L) = \left\{ \left(\frac{N}{L}\right)^2 + \frac{N}{L} \right\} \times \left(E_{\mathrm{DGEMM}}(N,L) + E_{\mathrm{DCOPY}}(N,L)\right), \qquad (6)$$

where $E_{\mathrm{DGEMM}}(N,L)$ means the time to execute DGEMM, and $E_{\mathrm{DCOPY}}(N,L)$ means the time to execute DCOPY. As is described in Section 2.1.3, the DGEMM routine executes the product of a $2L \times 2L$ matrix and a $2L \times N$ matrix. The DCOPY routine executes data copy whose length is $L \times N$. Thus it is considered that $E_{\mathrm{DGEMM}}(N,L) = O(4NL^2)$ and $E_{\mathrm{DCOPY}}(N,L) = O(NL)$. However, when L is small, it often occurs that the computation times $E_{\mathrm{DGEMM}}(N,L)$ and $E_{\mathrm{DCOPY}}(N,L)$ are not proportional to the computational costs.

We therefore estimate $E_{\mathrm{DGEMM}}(N,L)$ and $E_{\mathrm{DCOPY}}(N,L)$ from the measured values at sample points in the (N,L) plane. More specifically, we first measure the actual compuation time of DGEMM for several pairs of parameters $\{(N_i, L_j)\}$ $(i = 1,\ldots,n_N, j = 1,\ldots,n_L)$ (**Fig. 5**, Step 1). Then, for each j, we approximate the measured values at (N_i, L_j) $(i = 1,\ldots,n_N)$ with a linear function of N (Fig. 5, Step 2). Here, the least squares method is used to determine the coefficients of the linear function. Finally, from the values of the linear functions at (N, L_j) $(j = 1,\ldots,n_L)$, a quadratic function of L is computed by the least squares method (Fig. 5, Step 3). $E_{\mathrm{DGEMM}}(N,L)$ is estimated as the value of this quadratic function at L. Similarly, $E_{\mathrm{DCOPY}}(N,L)$ is estimated from the actual computation time of DCOPY. The only difference is that a linear function of L is used instead of a quadratic function in Step 3.

Once $E_{\mathrm{DGEMM}}(N,L)$ and $E_{\mathrm{DCOPY}}(N,L)$ have been estimated in this way, $E_{\mathrm{BM}}^{(2)}(N,L)$ can be calculated from Eq. (6).

3.2.2.3 The Other Processes

In the Bischof preprocessing and postprocessing, the execution time of DGEMM routines dominates most of the computation time. The computational cost of each DGEMM routine in these two stages is $O(LN^2)$, as can be seen from Algorithm 1 and Algorithm 2. This implies that as L increases, the computation speed of the DGEMM routine increases rapidly and stays constant. This can be confirmed from Figs. 3 and 4, which shows that the computation time of Bischof preprocessing and postprocessing decreases rapidly as L increases and then stays constant.

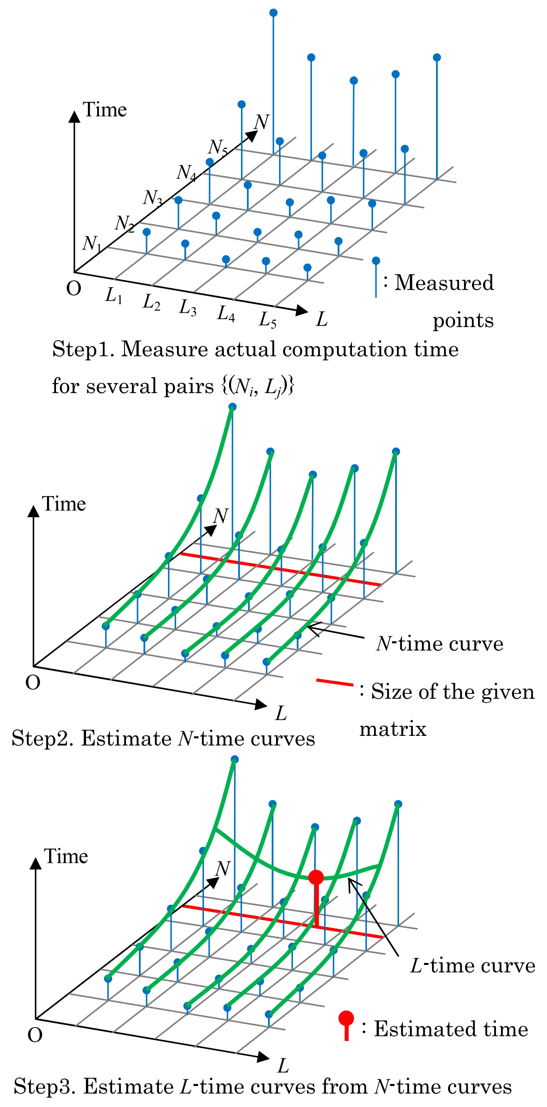

Time
N
N_1 N_2 N_3 N_4 N_5
O
L_1 L_2 L_3 L_4 L_5 L
: Measured points
Step1. Measure actual computation time for several pairs $\{(N_i, L_j)\}$
Time
N
O
L
N-time curve
: Size of the given matrix
Step2. Estimate N-time curves
Time
N
O
L
L-time curve
: Estimated time
Step3. Estimate L-time curves from N-time curves

Fig. 5 Process flow to estimate the computation time of the given pair of parameters (N, L).

To model the computation time of these parts, $E^{(3)}_{\mathrm{BM}}(N, L)$, we also use the procedure described in Fig. 5. In Step 2, we use a cubic function of N, while in Step 3, we use a natural cubic spline function of L.

3.2.2.4 Total Time

The total computation time $E_{\mathrm{BM}}(N, L)$ can be estimated by Eq. (5). In summary, we can estimate $E_{\mathrm{BM}}(N, L), \forall N, L > 0$ as follows:

(1) Determine several sampling points $\{(N_i, L_j)\}_{i,j}$ and obtain actual values of $E^{(1)}_{\mathrm{BM}}(N_i, L_j)$, $E_{\mathrm{DGEMM}}(N_i, L_j)$, $E_{\mathrm{DCOPY}}(N_i, L_j)$ and $E^{(3)}_{\mathrm{BM}}(N_i, L_j)$ at each sampling point. (Step 1 in Fig. 5). Once we have obtained the data and saved them, we need not do this step and we only have to load the saved data instead.

(2) Estimate N-time curves of $E^{(1)}_{\mathrm{BM}}(N, L_j)$, $E_{\mathrm{DGEMM}}(N, L_j)$, $E_{\mathrm{DCOPY}}(N, L_j)$ and $E^{(3)}_{\mathrm{BM}}(N, L_j)$ at several fixed $\{L_j\}$ (Step 2 in Fig. 5).

(3) Estimate L-time curves of $E^{(1)}_{\mathrm{BM}}(N, L)$, $E_{\mathrm{DGEMM}}(N, L)$, $E_{\mathrm{DCOPY}}(N, L)$ and $E^{(3)}_{\mathrm{BM}}(N, L)$ for the size N of the given matrix from the estimated N-time curves (Step 3 in Fig. 5).

(4) Calculate $E^{(2)}_{\mathrm{BM}}(N, L)$ from Eq. (6) and $E_{\mathrm{BM}}(N, L)$ from Eq. (5).

Here, N-time curve means a curve for which the horizontal axis is N and the vertical axis is the computation time. L-time curve means a curve for which the horizontal axis is L and the vertical axis is the computation time. The N-time and L-time curves are obtained by applying the least squares method to the functions shown in **Table 3**.

3.3 Implementation of Auto-Tuning

In order to utilize this auto-tuning function, we have to measure several actual values of $E^{(1)}_{\mathrm{BM}}(N_i, L_j)$, $E_{\mathrm{DGEMM}}(N_i, L_j)$, $E_{\mathrm{DCOPY}}(N_i, L_j)$ and $E^{(3)}_{\mathrm{BM}}(N_i, L_j)$ (Step 1 in Fig. 5). For this purpose, we prepare matrices of several sizes $\{N_i\}$ by random numbers, measure the computation time for several band widths $\{L_j\}$, and save the results (*preparation process*). This process has to be done only once, for example on installing the I-SVD library. If the computation environment such as CPU, OS, BLAS and so on is changed, this process should be done once again.

After the preparation process, we can use the auto-tuning function. Given an $N \times N$ input matrix, we compute its singular value decomposition as follows:

Table 3 Interpolation and extrapolation curves.

	N-time curve (L is fixed)	L-time curve (N is fixed)
The Murata preprocessing $E^{(1)}_{\rm BM}(N,L)$	Quadratic in N	Linear in L
Applying compact WY repr. $E^{(2)}_{\rm BM}(N,L)$	$\left\{\left(\frac{N}{L}\right)^2+\frac{N}{L}\right\}\times(E_{\rm DGEMM}(N,L)+E_{\rm DCOPY}(N,L))$	
$(E_{\rm DGEMM}(N,L))$	Linear in N	Quadratic in L
$(E_{\rm DCOPY}(N,L))$	Linear in N	Linear in L
The others $E^{(3)}_{\rm BM}(N,L)$	Cubic in N	Natural cubic spline of L
The Bischof-Murata algorithm $E_{\rm BM}(N,L)$	$E^{(1)}_{\rm BM}+E^{(2)}_{\rm BM}+E^{(3)}_{\rm BM}$	
The Dongarra algorithm $E_{\rm D}(N)$	Cubic in N	-

(1) Estimate the computation time $E_{\rm BM}(N,L)$ of the Bischof-Murata algorithm for several values of $\{L_j\}$, and choose a proper value of L.
(2) Estimate the computation time $E_{\rm BM}(N,L)$ of the Bischof-Murata algorithm at the parameter (N,L).
(3) Estimate the computation time $E_{\rm D}(N)$ of the Dongarra algorithm.
(4) Compare the 2 estimated time and apply the faster preprocessing.
(5) Apply the parallel I-SVD algorithm.
(6) Apply the postprocessing corresponding to the preprocessing used in Step (4).

4. Numerical Experiments

4.1 Experimental Conditions

In this section, we check the effectiveness of the auto-tuning function described in Section 3. Table 2 shows the test environment.

There is a published I-SVD library[16]. This library contains the Dongarra and the Bischof-Murata algorithms. Therefore users can utilize the library for dense square matrices. The I-SVD algorithm for dense square matrices is implemented as DGESLV. The function of auto-tuning described in this article is also implemented. We use the source code of the I-SVD library for the numerical experiments in this section.

This library is parallelized. The I-SVD algorithm is parallelized with *pthread*[17),18)]. The Murata preprocessing is parallelized with OpenMP[19)]. The Murata postprocessing, the Bischof pre/postprocessing, and the Dongarra algorithm are parallelized with GotoBLAS 2[20)]. Six cores are used on the computer G and two cores are used on the computer I.

For the experiments in this section, random matrices of several size are adopted. Before the experiments, actual computation time of the Dongarra and the Bischof-Murata algorithms is measured at 30 pairs of parameters $\{(N,L)\}, N=\{1{,}000, 2{,}000, 3{,}000, 4{,}000, 5{,}000\}$ and $L=\{50,100,150,200,250,300\}$, in order to estimate $E_{\rm BM}(N,L)$ and $E_{\rm D}(N)$. Actual computation time of DGEMM and DCOPY is also measured at these parameters.

4.2 Experimental Results

Figures 6 and **7** show interpolation curves for $N=2{,}000, 4{,}000$ and an extrapolation curve for $N=8{,}000$. According to the figures, the interpolation and extrapolation curves approximate the L-time curve well. Using the interpolation and extrapolation curves, we can guess a proper band width L and estimate the computation time.

Figures 8, **9** and **Table 4** show the auto-tuned band width L, estimated computation time and actual computation time with the band width. As can be seen from the graph, the estimated computation time approximates the actual computation time well. On the computer G, the Bischof-Murata algorithm is faster than the Dongarra algorithm. On the contraty, the Dongarra algorithm is faster than the Bischof-Murata algorithm on the computer I. This difference is also predicted by the estimated computation time.

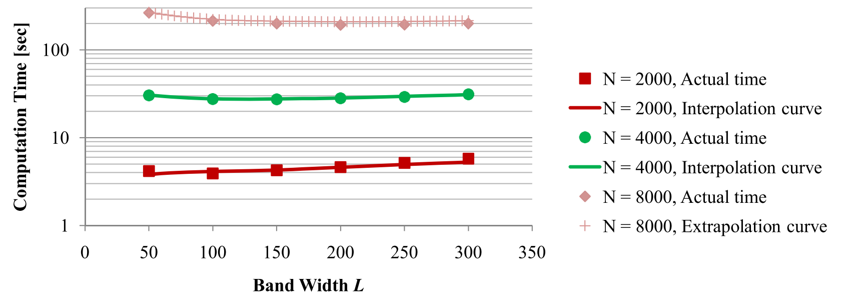


Fig. 6 Estimation curve of L-time and actual computation time (computer: G).

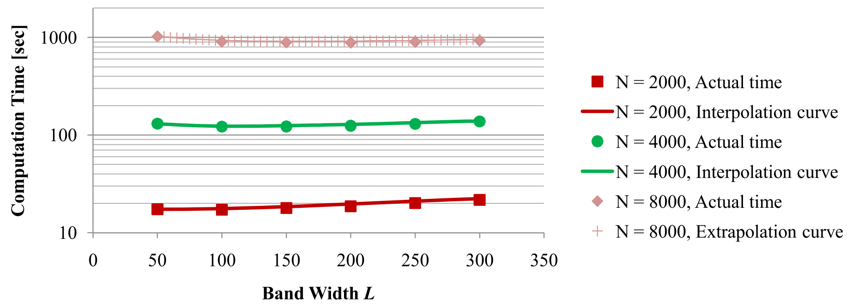


Fig. 7 Estimation curve of L-time and actual computation time (computer: I).

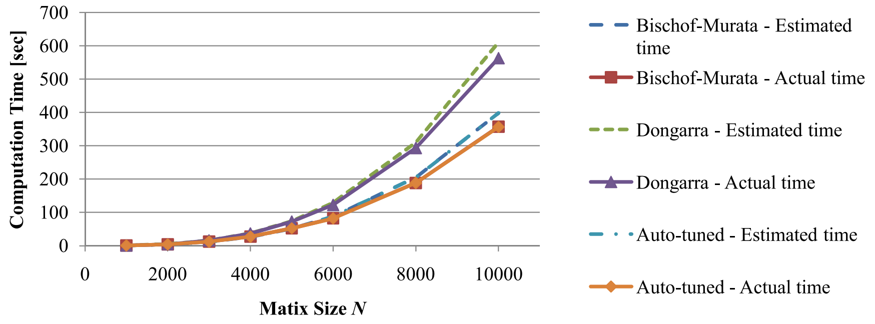


Fig. 8 Estimated and actual computation time (computer: G).

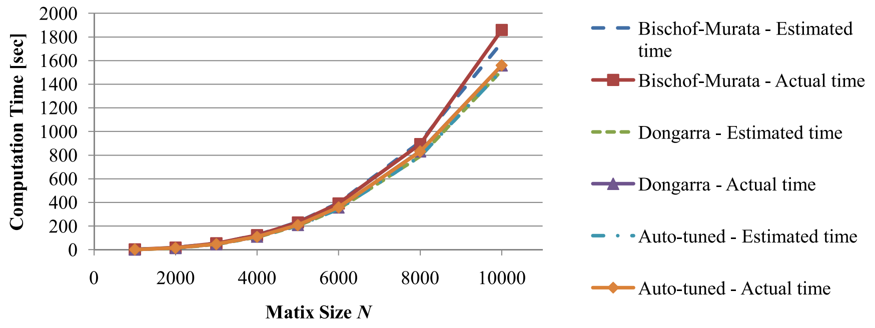


Fig. 9 Estimated and actual computation time (computer: I).

4.3 Performance Analysis

4.3.1 The Preferred Algorithm and the Performance Difference between DGEMV and DGEMM

In this subsection, we investigate why different pre/postprocessing algorithms are preferred on computers G and I. As is shown in Table 1, the Bischof-Murata algorithm is theoretically more expensive than the Dongarra algorithm. In particular, the computational work of Bischof-Murata postprocessing is twice that of Dongarra postprocessing. However, in the Dongarra algorithm, half of the computational work in the preprocessing phase is done with the slower level-2 BLAS, or DGEMV. In contrast, in the Bischof-Murata algorithm, most of the computations can be done with the level-3 BLAS, or DGEMM. Considering these characteristics, it is likely that which algorithm is faster depends on performance difference between DGEMV and DGEMM.

To confirm this, we measured the performance of the DGEMV and DGEMM routines on both computers. The sizes of the matrices for DGEMV are those used in Eq. (1) in Dongarra preprocessing. The sizes of the matrices for DGEMM are those used in Steps 3 and 7 of Bischof preprocessing (Algorithm 1). The results on computer G and I are shown in **Figs. 10** and **11**, respectively. It can be seen from the graph that the performance advantage of DGEMM over DGEMV is more than 15 times on computer G, while it is about 6 times on computer I. This explains why the Bischof-Murata algorithm, which replaces DGEMV with DGEMM by permitting increase in the computational work, is advantageous on computer G.

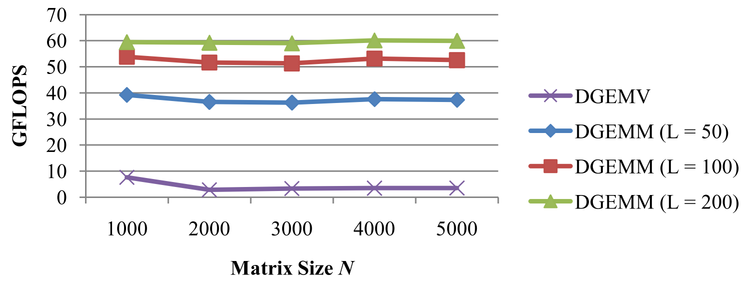


Fig. 10 Performance of the DGEMV and DGEMM routines (computer: G).

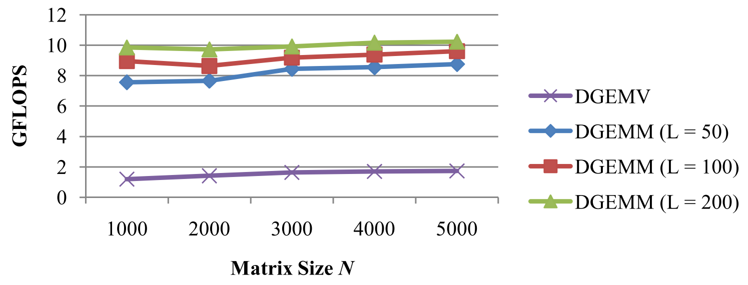


Fig. 11 Performance of the DGEMV and DGEMM routines (computer: I).

4.3.2 The Origin of Performance Difference between DGEMV and DGEMM

Next, we consider why the performance difference between DGEMV and DGEMM differs on computer G and I based on the hardware specification shown in Table 2. As is well known [8),9)], on most computers, a properly implemented DGEMM routine that fully utilizes the cache memory can achieve near-peak performance of the hardware. On both of computers G and I, each core can perform four double-precision floating-point operations per cycle using SSE2 instructions. Thus we can estimate the peak performance of computer G as 80 GFLOPS (4 operations $\times$ 3.33 GHz $\times$ 6 cores), and that of computer I as 21.33 GFLOPS (4 operations $\times$ 2.66 GHz $\times$ 2 cores).

On the other hand, the performance of DGEMV is usually bounded by the memory bandwidth. When the vector is in the cache and the matrix is out of the cache, the DGEMV routine requires loading one floating-point data (8 Bytes) to perform two floating-point operations. Thus the ratio of data transfer to floating-point operations is 4 Bytes/FLOP. Hence, the maximum achievable performance on computer G is 6.4 GFLOPS ((25.6 GBytes/s) $\div$ (4 Bytes/FLOP)), while it is 2.13 GFLOPS ((8.5 GBytes/s) $\div$ (4 Bytes/FLOP)) on computer I. Accordingly, if both DGEMM and DGEMV achieve maximum performance, the performance difference between DGEMV and DGEMM is 12.5 times ($= 80 \div 6.4$) on computer G and 10 times ($= 21.33 \div 2.133$) on computer I. Thus the performance difference is slightly larger on computer G.

According to Fig. 10, on computer G, DGEMM achieves more than 75% of the hardware peak performance, while DGEMV achieves only 55% of the maximum performance bounded by the memory bandwidth. On the contrary, on computer I, DGEMM achieces only 50% of the hardware peak performane, while DGEMV achieves nearly 80% of the maximum performance bounded by the memory bandwidth (see Fig. 11). Such variation in the efficiency of the BLAS routines, along with the performance limit by hardware specification, gives rise to the performance difference between DGEMV and DGEMM.

Table 4 Auto-tuned parameters and computation times.

	Matrix	Auto-tuned algorithm/parameter			Actual time		
Computer	size *N*	Algorithm	Band width *L*	Estimated time	BM	D	Error
G	1,000	BM	50	0.6	0.7	0.5	+23%
	2,000	BM	50	3.9	4.2	4.3	+7%
	3,000	BM	107	11.6	12.2	15.7	+5%
	4,000	BM	117	26.5	27.3	36.9	+3%
	5,000	BM	126	51.3	52.4	72.1	+2%
	6,000	BM	192	87.6	82.1	122.2	−7%
	8,000	BM	200	204.8	188.6	293.4	−9%
	10,000	BM	202	398.4	356.9	563.5	−12%
I	1,000	D	50	1.9	2.8	1.9	−1%
	2,000	D	66	14.7	17.4	14.9	+2%
	3,000	D	97	46.9	54.0	47.5	+1%
	4,000	D	111	107.0	123.0	110.2	+3%
	5,000	D	123	203.1	230.4	208.6	+3%
	6,000	D	133	343.6	391.2	358.6	+4%
	8,000	D	155	791.0	892.6	831.9	+5%
	10,000	D	172	1,515.4	1,859.3	1,560.9	+3%

BM: the Bischof-Murata algorithm, D: the Dongarra algorithm

Note that DGEMV can achieve higher performance when the matrix is stored in the cache. This is the case of $N = 1{,}000$ on computer G, where the matrix is of 8 MBytes and can be stored in the L3 cache. However, such a situation occurs only for relatively small matrices. We therefore assumed that the matrix is out of the cache in the above analysis.

4.4 Cost of the Auto-tuning

Finally, we discuss the cost of auto-tuning. As explained in Sections 3.2.2.4 and 3.3, the cost of auto-tuning consists of the following two parts:

- The cost to measure the actual values of $E^{(1)}_{\mathrm{BM}}(N, L)$, $E_{\mathrm{DGEMM}}(N, L)$, $E_{\mathrm{DCOPY}}(N, L)$ and $E^{(3)}_{\mathrm{BM}}(N, L)$ at sampling points in the (N, L) plane and measure the actual values of $E^{(N)}_{\mathrm{D}}$ at sampling points on the N axis (preparation process).
- The cost to model the execution time $E_{\mathrm{BM}}(N, L)$ and $E_{\mathrm{D}}(N)$ based on the sampled data, evaluate their values for given N and several values of L, choose the best value of L for the Bischof-Murata algorithm, and decide which algorithm is faster.

As noted in Section 3.2.2.4, the former part needs to be done only once for each computational environment. The latter part is done each time the bidiagonalization routine is called.

The time required for the former part is 769 seconds on computer G and 3,181 seconds on computer I. This is comparable to the time required to install ATLAS [21)], a BLAS library with auto-tuning facility. On the other hand, the time required for the latter part is 3.4×10^{-4} second and 7.1×10^{-4} second on computer G and I, respectively. Thus it is negligible compared with the time for pre/postprocessing.

5. Conclusion

The computation time for the Bischof-Murata and the Dongarra algorithms depends on the specification of computers, the matrix size N, and the band width L. For fast computation, we have to select an appropriate algorithm and a band width L. In this article, a method is devised to estimate faster pre/postprocessing and a proper band width L. For accurate estimation, the Bischof-Murata algorithm is divided into 3 parts and the computation time are estimated individually. Furthermore N-time curves and L-time curves are made separately.

Numerical experiments are carried out to examine the effectiveness of our method. The result of the experiments shows that the proposed manner is effective.

References

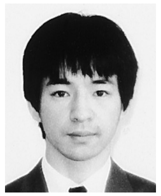
1) Iwasaki, M. and Nakamura, Y.: Accurate computation of singular values in terms of shifted integrable schemes, *Japan Journal of Industrial and Applied Mathematics*, Vol.23, pp.239–259 (2006).
2) Nakamura, Y.: *Functionality of Integrable System* (*in Japanese*), Kyoritsu Publishing, Tokyo (2006).
3) Takata, M., Kimura, K., Iwasaki, M. and Nakamura, Y.: Performance of a new scheme for bidiagonal singular value decomposition of large scale, *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks* (*PDCN2006*), pp.304–309 (2006).
4) Takata, M., Kimura, K., Iwasaki, M. and Nakamura, Y.: Implementation of library for high speed singular value decomposition, *J. IPS Japan*, Vol.47, No.SIG7 (ACS 14), pp.91–104 (2006).
5) Dongarra, J.J., Hammarling, S.J. and Sorensend, D.C.: Block reduction of matrices to condensed forms for eigenvalue computations, *Journal of Computational and Applied Mathematics*, Vol.27, pp.215–227 (1989).
6) Bischof, C.H., Marques, M. and Sun, X.: Parallel bandreduction and tridiagonalization, *Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp.22–24 (1993).
7) Murata, K. and Horikoshi, K.: A new method for the tridiagonalization of the symmetric banded matrices, *J. IPS Japan*, Vol.16, pp.93–101 (1975).
8) Golub, G.H. and van Loan, C.F.: *Matrix Computations*, 3rd ed., Johns Hopkins University Press (1996).
9) Demmel, J.W.: *Applied Numerical Linear Algebra*, SIAM, Philadelphia (1997).
10) Dackland, K. and Kågström, B.: A hierarchical approach for performance analysis of ScaLAPACK-based routines using the distributed linear algebra machine, *Proc. Applied Parallel Computing, Industrial Computation and Optimization, 3rd International Workshop* (*PARA 96*), pp.186–195 (1996).
11) Cuenca, J., Giménez, D. and González, J.: Architecture of an automatically tuned linear algebra library, *Parallel Computing*, Vol.30, No.2, pp.187–210 (2004).
12) Katagiri, T., Kise, K., Honda, H. and Yuba, T.: ABCLibScript: A directive to support specification of an auto-tuning facility for numerical software, *Parallel Computing*, Vol.32, No.1, pp.92–112 (2006).

13) Yamamoto, Y.: Performance modeling and optimal block size selection for a BLAS-3 based tridiagonalization algorithm, High Performance Computing and Grid in Asia Pacific Region, *International Conference on Eighth International Conference on High-Performance Computing in Asia-Pacific Region* (*HPC-ASIA'05*), pp.249–256 (2005).
14) Schreiber, R. and Van Loan, C.: A storage-efficient WY representation for products of Householder transformations, *SIAM J. Sci. Stat. Comput.*, Vol.10, No.1, pp.53–57 (1989).
15) LAPACK: http://www.netlib.org/lapack/
16) I-SVD Library: http://www-is.amp.i.kyoto-u.ac.jp/lab/isvd/download/
17) Toyokawa, H., Kimura, K., Takata, M. and Nakamura, Y.: On parallelism of the I-SVD algorithm with a multi-core processor, *JSIAM Letters*, Vol.1, pp.48–51 (2009).
18) Toyokawa, H., Kimura, K., Takata, M. and Nakamura, Y.: On parallelization of the I-SVD algorithm and its evaluation for clustered singular values, *Proc. International Conference on Parallel and Distributed Processing Techniques and Application*, pp.711–717 (2009).
19) OpenMP: http://openmp.org/
20) GotoBLAS2: http://www.tacc.utexas.edu/tacc-projects/gotoblas2/
21) Whaley, R.C., Petitet, A. and Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project, *Parallel Computing*, Vol.27, No.1-2, pp.3–25 (2001).

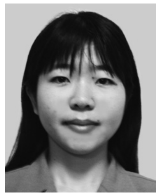
(Received October 1, 2010)
(Accepted January 5, 2011)

Hiroki Toyokawa received his B.E. and M.I. degrees from Kyoto University, in 2008 and 2010, respectively. He has been working for NS Solutions Corporation since 2010. Since 2011, he has also been a doctoral course student at Kyoto University while working.

Kinji Kimura received his Ph.D. degree from Kobe University in 2004. He became a PRESTO, COE, and CREST researcher in 2004 and 2005. He became an assistant professor at Kyoto University in 2006, an assistant professor at Niigata University in 2007, a lecturer at Kyoto University in 2008, and has been a research associate professor at Kyoto University since 2009. He is an IPSJ member.



Yusaku Yamamoto received his master's degree in Material Physics from the University of Tokyo in 1992. He started working at the Central Research Laboratory, Hitachi, Ltd. from 1992. He became a visiting scholar at business school of Columbia University in 2001. In 2003, he moved to Nagoya University as an assistant professor. He received his Ph.D. degree in Engineering from Nagoya University in 2003. He became a lecturer at Nagoya University in 2004, associate professor at Nagoya University in 2006. He is now a professor at the Department of Computational Science, Kobe University. His research interests include numerical algorithms for large-scale matrix computation and financial engineering problems.



Masami Takata is an assistant professor of the Department of Advanced Information and Computer Sciences at Nara Women's University. She received her Ph.D. degree from Nara Women's University in 2004. Her research interests include parallel algorithms for distributed memory systems and numerical algebra.

Akira Ajisaka received his B.E. and M.I. degrees from Kyoto University, in 2009 and 2011, respectively. Since 2011, he has been working at NTT DATA Corporation.

Yoshimasa Nakamura was born in 1955. After getting Ph.D. thesis in 1983 from Kyoto University, he worked for Mathematics Departments of Gifu University, Doshisha University and Osaka University. From 2001 he has been a faculty member of the Graduate School of Informatics, Kyoto University. His research interests include integrable dynamical systems. Integrable systems originally appear in classical mechanics. But they have a rich mathematical structure. His recent subject is to find possible applications of integrable systems to various areas such as numerical algorithms, combinatorics and mathematical statistics. He is a member of JSIAM, MSJ, SIAM and AMS.