An Efficient Implementation of Parallel Eigenvalue Computation for Massively Parallel Processing

Takahiro Katagiri [a,b,*], Yasumasa Kanada [b]

[a] *Research Fellow of the Japan Society for the Promotion of Science*

[b] *Computer Centre Division, Information Technology Center, The University of Tokyo, 2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-8658, JAPAN*

Abstract

This article describes an efficient implementation and evaluation of a parallel eigensolver for computing all eigenvalues of dense symmetric matrices. Our eigensolver uses a Householder tridiagonalization method, which has higher parallelism and performance than conventional methods when problem size is relatively small, e.g. the order of 10,000. This is very important for relevant practical applications, where many diagonalizations for such matrices are required so often. The routine was evaluated on the 1024 processors HITACHI SR2201, and giving speedup ratios of about 2–5 times as compared to the ScaLAPACK library on 1024 processors of the HITACHI SR2201.

Key words: Parallel eigensolver; Parallel tridiagonalization; Householder algorithm; Linear algebra; Massively Parallel Processing;

1 Introduction

Parallelizing eigensolvers for symmetric dense matrices have been researched by many individuals [17,5,7,19,10,1,15,3,20,13]. However, parallelization for massively parallel processing (MPP) has not received much attention because (1) there were only few real MPP machines; (2) efficient MPP implementations are hard to attain. It is especially difficult to attain high performance when

* Corresponding author.

Email addresses: katagiri@pi.cc.u-tokyo.ac.jp (Takahiro Katagiri), kanada@pi.cc.u-tokyo.ac.jp (Yasumasa Kanada).

matrix size is relatively small and processor elements (PEs) are large numbers. Such a situation is not rarely seen on an MPP system. This is why this paper will focus on high performance algorithm on MPP systems.

Many kinds of MPP architectures which include hundreds or more PEs are becoming more available. On these MPP systems, many classical algorithms such as Jacobi method or divide-and-conquer method [6] will work poorly, since they cannot be readily adapted to a parallel environment. The reasons are explained as the following. Jacobi method has the problem of high message set-up overheads when PEs increase, even when blocking and over-lapping techniques [9] are applied. For the divide-and-conquer method, approach to tridiagonal matrices or QR decomposition can be implemented at high performance on shared memory parallel machines [8]. Nevertheless, the classical divide-and-conquer approaches cause heavy data communications on the distributed memory parallel machines when eigenvectors are calculated. Above all the classical algorithms with the problems above, we want to focus on one of the well-known classical algorithms, the Householder algorithm. The main reason is because Householder algorithm can decrease communication complexity according to the increase of the number of PEs in a certain data distribution.

Conventional parallel Householder algorithms based on a sequential Householder algorithm [19,10,1] have the problem of increasing communication complexity. Conventional algorithms based on the symmetry of the matrix, increase communication complexity in comparison with algorithms that assume non-symmetry, even though the conventional algorithms have half the computational complexity of non-symmetric ones. In MPP environments, the increase of communication complexity must be considered. For instance, typical conventional Householder tridiagonalizations using symmetry (called HTS hereafter) have computational complexity of $4/3 \cdot n^3/p \cdot \delta_1$, and communication complexity of $\gamma_1 \cdot n^2 \log_2 p$, where n is the problem size, p is the number of PEs, δ_1 is an execution time per floating-point computation, and γ_1 is the time for communications in the HTS. In the same way as in the HTS algorithms, we can estimate execution time for Householder tridiagonalizations based on the non-symmetry (HTN hereafter). These algorithms have a computational complexity of $8/3 \cdot n^3/p \cdot \delta_2$, and communication complexity of $\gamma_2 \cdot n^2 \log_2 p$, where δ_2 is an execution time per floating-point computation, and γ_2 is the time for communications in the HTN. From these relations, we will obtain a threshold size n_{thd} of the problem where the HTN is faster than the HTS:

$$n_{thd} < \frac{3}{4} C_{\gamma/\delta} \cdot p \log_2 p, \tag{1}$$

where $C_{\gamma/\delta} \equiv (\gamma_1 - \gamma_2)/(2\delta_2 - \delta_1)$. The n_{thd} depends on the number of PEs and the factor $C_{\gamma/\delta}$ for communication. For example, if $p = 4$, then $n_{thd} < 6 \cdot C_{\gamma/\delta}$,

and if $p = 1024$, then $n_{thd} < 7680 \cdot C_{\gamma/\delta}$. This relation shows that (i) n_{thd} grows with increasing p; (ii) a lower γ_2 gives us large values for n_{thd}. The conclusion is that decreased communication times is important even if the HTN algorithms have twice the computational complexity in MPP environments. From the consequence of (i) and (ii), we propose a reduced communication algorithm assuming non-symmetry. In addition, we can obtain high efficiency even if matrix size of the proposed algorithm is relatively small.

This paper is organized as follows. The parallel processing environment and the mathematical notations used throughout this paper is described in Section 2. Section 3 describes an efficient parallel algorithm for computing all eigenvalues on MPP systems. In Section 4, execution time for our routine on the HITACHI SR2201 along with comparisons to the ScaLAPACK routine on the same machine is shown. Finally, Section 5 is conclusions regarding this work.

2 Parallel processing environment and several notations

Let us assume that our target parallel computers are constructed with homogeneous PEs in processing speed, memory size and communication speed, and their PEs are labeled in a two-dimensional mesh of size $q \times r = p$, where p is the number of PEs. Let $P_{myidx,myidy}$, $(myidx = 0, 1, ..., q-1, myidy = 0, 1, ..., r-1)$ be a two-dimensional index for the PEs. In addition, we assume that all PEs are connected with a network in order to broadcast messages and perform reduction operations, such as global summation for local data.

In our implementation, the Householder transformation is used for the similarity transformation. That is,

Theorem 1 *Given a vector $z \in \Re^n$, the following vector $u \in \Re^n$ and scalar $\alpha \in \Re$ exist (See [6]):*

$$(I - \alpha u u^T)z = (\xi_1, \cdots, \xi_k, \pm\sigma, 0 \cdots, 0)^T, \text{where} \quad \sigma = ||z_{k+1:n}||_2. \tag{2}$$

The vector $u \equiv (0, \cdots, 0, \xi_{k+1} \pm \sigma, \xi_{k+2}, \cdots, \xi_n)^T$ and the scalar $\alpha \equiv 1/(\sigma^2 + |\xi_{k+1}\sigma|)$ are a pair of quantities which satisfy the above theorem, and $\alpha u^T u = 2$ since $||u||_2^2 = (\xi_{k+1} \pm \sigma)^2 + \xi_{k+2}^2 + \cdots + \xi_n^2 = \sigma^2 + \sigma^2 + 2|\xi_{k+1}\sigma| = 2(\sigma^2 + |\xi_{k+1}\sigma|) = 2/\alpha$. The sign of the scalar σ is same as that of ξ_{k+1} to minimize catastrophic cancellation of significant digits when calculating the elements of the vector u. The reflection $(I - \alpha u u^T)z$ in the theorem 1 is called *Householder transformation*. We represent the reflection $(I - \alpha u u^T)z$ by $H^{(k)}(z)$. This reflection does not affect the elements $\xi_1, \cdots, \xi_k$. (u, α) is the pair to be required to perform the above reflection $H^{(k)}(z)$ in the formula (2).

Next in this section is the explanation of the notation for the matrix data distribution. Here, let Π be a set of row indices of matrix A, and Γ be a set of column indices, where $j \in \Pi, \Gamma$ satisfies the relation of $1 \leq j \leq n$, and n is a matrix dimension. These sets are different among each data distribution. With these sets, we define 2-D distributions called the grid-wise distribution (Cyclic, Cyclic) as follows:

$$\begin{aligned}
\Pi = &\{myidx + 1 + (j-1)q\}, \\
&\quad j = 1, 2, \cdots, \mathbf{last_c}(myidx + 1 + q \times \lfloor n/q \rfloor, \lfloor n/q \rfloor), \\
\Gamma = &\{myidy + 1 + (j-1)r\}, \\
&\quad j = 1, 2, \cdots, \mathbf{last_c}(myidy + 1 + r \times \lfloor n/r \rfloor, \lfloor n/r \rfloor),
\end{aligned} \tag{3}$$

where the value of $\mathbf{last_c}(a,b)$ is defined as

$$\mathbf{last_c}(a,b) = \begin{cases} \text{if } a \leq n \text{ then } b+1, \\ \text{if } a > n \text{ then } b. \end{cases} \tag{4}$$

Finally, we tabulate the notation used in this paper in Table 1 in order to define algorithms.

Table 1
Mathematical notation and its explanation.

Notation	Explanation
α, μ, σ	scalars $\in \Re$
x, y, u	vectors $\in \Re^n$
$\chi_i, \eta_i, \upsilon_i$	i-th elements in the above vectors x, y, u
x_Π	a partial vector constructed from the arguments which are indexed by a set Π on the above vector x
A	a matrix $\in \Re^{n \times n}$
$A_{i:j,k}$	a partial vector constructed from the $i, \cdots, j$-th rows and the k-th column of the above matrix A
$A_{\Pi,j}$	a partial matrix constructed by rows indexed by a set Π and the j-th column of the above matrix A
$A^{(k)}$	k-th iteration of matrix A

3 Outline of our parallel eigenvalue computation process

3.1 Outline of the entire process

Here, we assume that the data of our matrix are already distributed over the PEs. Under this condition, our parallel eigensolver calculates eigenvalues using the following well-known three steps:

(1) Transforming a dense symmetric matrix to a tridiagonal matrix in parallel (*The tridiagonalization routine*).
(2) Re-distributing the non-zero elements of the tridiagonal matrix over all PEs (*The re-distribution routine*).
(3) Computing all eigenvalues for the gathered tridiagonal matrix in the step (2) by the bisection method in parallel (*The eigenvalue computation routine*).

We use the Householder transformation in the process (1), which is called Householder-bisection method.

3.2 The tridiagonalization process

We consider the following transformation: $A^{(1)} \equiv A$ to tridiagonal form $A^{(n-2)}$, where $A^{(k)}$ is defined as in Table 1. This transformation is denoted by $H^{(k)}(z)$ $= H^{(k)}(A^{(k)}_{k:n,k})$. By substituting $H^{(k)} = I - \alpha uu^T$ for $H^{(k)}(z)$ in $(k+1)$-th iteration, the following equations are obtained:

$$\begin{aligned} A^{(k+1)} &= H^{(k)} A^{(k)} H^{(k)} \\ &= A^{(k)} - \alpha A^{(k)} uu^T - \alpha uu^T A^{(k)} - \alpha^2 uu^T A^{(k)} uu^T \\ &= A^{(k)} - xu^T - uy^T + \alpha uu^T xu^T \\ &= A^{(k)} - uy^T + u\mu u^T - xu^T \\ &= A^{(k)} - u(y^T - \mu u^T) - xu^T, \end{aligned} \tag{5}$$

where

$$x = \alpha A^{(k)} u, \quad y^T = \alpha u^T A^{(k)}, \quad \mu = \alpha u^T x. \tag{6}$$

For tridiagonalization process, the matrix A is symmetric. Therefore, $x = y$, and we can obtain the following formula:

$$A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T. \tag{7}$$

Note that to execute k-th iteration, we need the column vector $A_{k:n,k}$ which is obtained from the partial matrix $A_{k:n,k:n}$.

We have already developed the tridiagonalization and Hessenberg reduction routines [14] by the Householder transformation. Figure 1 shows our parallel tridiagonalization algorithm. The routine of Figure 1 reduces communication

```
c     P_{myidx,myidy} owns row set Π and
c       column set Γ of n × n matrix A.
⟨1⟩   do k=1, n − 2
⟨2⟩     if (k ∈ Γ) then
⟨3⟩       broadcast(A^(k)_{Π,k}) to
              PEs sharing rows Π
⟨4⟩     else
⟨5⟩       receive(A^(k)_{Π,k})
⟨6⟩     endif
⟨7⟩     computation of (u_Π, α)
⟨8⟩     if (I have diagonal elements of A)
        then
⟨9⟩       broadcast(u_Π) to
              PEs sharing columns Γ
⟨10⟩    else
⟨11⟩      receive(u_Γ)
⟨12⟩    endif
⟨13⟩    do j=k, n
⟨14⟩      if (j ∈ Γ) x_Π = x_Π + α A^(k)_{Π,j} υ_j
          endif
⟨15⟩    enddo
⟨16⟩    global summation of x_Π to PEs
          sharing rows Π
⟨17⟩    if (I have diagonal elements of A)
        then
⟨18⟩      broadcast(x_Π) to
              PEs sharing columns Γ
⟨19⟩    else
⟨20⟩      receive(x_Γ)
⟨21⟩    endif
⟨22⟩    do j=k, n
⟨23⟩      μ = α u_Π^T x_Π   enddo
⟨24⟩    global summation of μ to
            PEs sharing rows Π
⟨25⟩    do j=k, n
⟨26⟩      do i=k, n
⟨27⟩        if (i ∈ Π .and. j ∈ Γ) then
⟨28⟩          update A^(k+1)_{i,j} =
              A^(k)_{i,j} − υ_i (χ_j^T − μ υ_j^T) − χ_i υ_j^T
⟨29⟩        endif   enddo   enddo
c       remove k from active columns
          and rows
⟨30⟩    if (k ∈ Γ)  Γ = Γ − {k}   endif
⟨31⟩    if (k ∈ Π) Π = Π − {k}   endif
⟨32⟩  enddo
```

Fig. 1. Parallel algorithm for the tridiagonalization (the (Cyclic, Cyclic) grid-wise distribution).

and broadcast time for vector reduction to a ratio of $1/\sqrt{p}$. The same idea appears in [4,11,10]. Table 2 summarizes the communication complexity for the algorithm in Figure 1.

Table 2 shows that our algorithm requires at most 5 times reduction operations per iteration for the tridiagonalization. This means that parallelizing the tridiagonalization is more difficult than parallelizing other numerical decompositions, such as LU decomposition. In addition, we can see that the communication complexity of our algorithm depends on the topology of PE grids.

Our process does not depend on the symmetry of the matrix, so that our rou-

Table 2
Communication complexities of reduction operation for the tridiagonalization of Figure 1 in k-th iteration.

Line No.	Total communication setup times	Message length per one communication	Comments
$\langle 7 \rangle$	$\lceil \log_2(r) \rceil$	1	
$\langle 8 \rangle$–$\langle 12 \rangle$	$\lceil \log_2(r) \rceil$	$\lceil (n-k+1)/q \rceil$	if $r = q$, broadcast
$\langle 16 \rangle$	$\lceil \log_2(q) \rceil$	$\lceil (n-k+1)/r \rceil$	
$\langle 17 \rangle$–$\langle 21 \rangle$	$\lceil \log_2(r) \rceil$	$\lceil (n-k+1)/q \rceil$	if $r = q$, broadcast
$\langle 24 \rangle$	$\lceil \log_2(r) \rceil$	1	

tine has twice the computational complexity ($8/3n^3$) as much as that of the standard process ($4/3n^3$) [11,1,10]. However, our process has a lower communication complexity than the routine in [11,1,10], since data structures and data access patterns are simple. This is explained by the following reasons. To decrease computational complexity, the processes of $\langle 25 \rangle$–$\langle 29 \rangle$ in Figure 1 are improved to update upper tridiagonal part of A. After the improvement, we have to implement either of the following two methods since lower tridiagonal part of A is not calculated.

(1) *Method for compressed data form*: Additional communications and re-structuring the calculation processes of $\langle 13 \rangle$–$\langle 15 \rangle$ are needed.
(2) *Method for non-compressed data form*: After the modified processes of $\langle 25 \rangle$–$\langle 29 \rangle$ in Figure 1, data re-distribution for lower tridiagonal part of A is needed.

For instance, the routine in [21] (the ScaLAPACK's tridiagonalization routine [1]) is implemented according to the above method (1). By the method (1), the routine in [21] requires 4 reduction operations to execute the processes $\langle 13 \rangle$–$\langle 15 \rangle$, while our routine requires only 1 reduction operation. Note that blocked algorithm [21] needs additional communications to perform block update. For instance, the routine in [21] needs 4 spread communications and 1 reduction operation, while our routine needs no communication.

Table 3 shows the communication complexities of each implementation method. The communication complexities depend on the implementation of communication methods in general. The communication complexities of Table 3 are calculated by using the implementation of ScaLAPACK[21] for the method (1), and all-to-all communication for the method (2).

The communication complexities of Table 3 indicate that the method (2) has $O(n^3)$ total communication amount, and this is much higher complexity in comparison with the other implementations. Therefore, the implementation

Table 3
Communication complexities of matrix-vector product for the tridiagonalization.

Implementation	Communication times	Total communication volume
Method (1)	$4\ n\ \lceil\ \log_2(r)\ \rceil$	$2\ n^2\ \lceil\ \log_2(r)/q\ \rceil$
Method (2)	$n\ \lceil\ \log_2(r)\ \rceil + n(p-1)$	$n^2/2\ \lceil\ \log_2(r)/q\ \rceil$ $+\ (p-1)(n^3/(3p) + n^2/(2p))$
Figure 1	$n\ \lceil\ \log_2(r)\ \rceil$	$n^2/2\ \lceil\ \log_2(r)/q\ \rceil$

of the method (2) is not addressed.

Let ϕ_1 be a time per floating-point operation in the method of Figure 1, and ϕ_2 be a time per floating-point operation in the implementation method (1). Let α be a message setup time, and β be a communication time per floating-point data. By using these variables, we can estimate a condition that the execution time for the method of Figure 1 is faster than that of the method (1):

$$4n^2/3 \cdot (2\phi_1 - \phi_2) < \log_2(r) \cdot (3\alpha + n\beta/q), \tag{8}$$

where the values of n, r, q, α and β are positive. The condition of (8) will be achieved when $2\phi_1 - \phi_2 < 0$. This indicates that the floating-point operation factor of ϕ_1 for the method (1) can greatly affect the condition. Hence, if the floating-point operation of Figure 1 is 2 times faster than that of the method (1), the method of Figure 1 is always faster than that of the method (1). Note that we can implement the method of Figure 1 at high performance with comparison to the method (1), since the kernel of the method of Figure 1 does not use the special data structure for symmetric matrices.

For these reasons, we expect that our routine is faster than conventional routines when the number of PEs increases, and adaptability of the real problems is high. In addition, our routine does not support block-cyclic distribution. The block-cyclic distribution causes a poor load balance when n/p is small. This condition easily occurs on MPP, hence, such distribution is not suitable for parallel routines for MPP.

3.3 The re-distribution process

In order to have the entire tridiagonal matrix on each PE, the grid-wise (Cyclic, Cyclic) distributed elements are re-distributed in this process.

3.4 The eigenvalue computation process

The bisection method is implemented to compute the eigenvalues. Implementation of the bisection method is the same as the routine `BISECT` in reference [18].

In our implementation, *nbi*, which is the number of the iterations for improving the accuracy when an eigenvalue is isolated, is set to 200 to calculate all eigenvalues. Note that the routine does not iterate 200 times, since the routine is finished when narrowing section is small enough (such as machine epsilon). Parallelizing the routine is easy because we can parallelize the routine with different initial values.

4 Performance evaluation

We have implemented our parallel eigensolver on the HITACHI SR2201, and evaluated its performance. The HITACHI SR2201 system is a distributed memory, message-passing parallel machine of the MIMD class. It is composed of 1024 PEs, each having a pseudo vector processor [2] and 256 Megabytes of main memory, interconnected via communication network that has the topology of a three-dimensional hyper-crossbar. The peak interprocessor communications bandwidth is 300 Mbytes/s in each direction. The communication library used for the SR2201 is the MPI (Message Passing Interface). We used the HITACHI Optimized Fortran90 V02-06-/D compiler. The compile options we used were *-rdma -W0,'OPT(O(SS))'*.

4.1 Performances in the Frank matrix

To evaluate performances and to check the program, we calculated eigenvalues and eigenvectors for the following matrix:

$$A_n = (a_{ij}),\ a_{ij} = n - \mathbf{max}(i,j) + 1. \tag{9}$$

Its eigenvalues are known to be:

$$\lambda_k = \frac{1}{2\left(1 - \cos\frac{(2k-1)}{(2n+1)}\pi\right)},\ k = 1, 2, \ldots, n. \tag{10}$$

4.1.1 Performance for calculating all eigenvalues

We measured the execution time of an order 8000 all eigenvalue problem with the SR2201 on 4–1024 PEs. Table 4 shows the execution time.

Table 4
Calculation times of 8000 eigenvalues in seconds. ($nbi = 200$, the maximal relative error with respect to the analytical values is 0.2493×10^{-7})

PEs	4	8	16	32
(Grid)	(2×2)	(2×4)	(4×4)	(4×8)
Tridiagonalization	1962	989.5	490.3	254.9
(Ratio %)	(95.1%)	(94.6%)	(94.0%)	(93.8%)
Re-distribution	0.002	0.004	0.005	0.006
Bisection	98.57	55.61	30.86	16.79
(Ratio %)	(4.7%)	(5.3%)	(5.9%)	(6.1%)
Total time	2061	1045	521.2	271.7
Speedup	1.00	1.97	3.95	7.58
PEs	64	128	256	1024
(Grid)	(8×8)	(8×16)	(16×16)	(32×32)
Tridiagonalization	119.0	70.42	47.90	63.16
(Ratio %)	(92.1%)	(92.7%)	(93.9%)	(98.6%)
Re-distribution	0.011	0.013	0.025	0.082
Bisection	10.15	5.469	3.060	0.783
(Ratio %)	(7.8%)	(7.2%)	(6.0%)	(1.2%)
Total time	129.2	75.90	50.99	64.02
Speedup	15.9	27.1	40.4	32.1

From Table 4, the tridiagonalization process occupies the largest part (more than 90%) of the total parallel execution time, while the time for re-distribution took 0.2% at most. Therefore, improvement of the re-distribution routine is not an issue. Notwithstanding its scalar implementation, the time for calculation of the eigenvalues took only about 8% of the total time. Therefore, improvement of the routine for the eigenvalue calculation is not necessary on this machine. Finally, we can conclude that fast parallel tridiagonalization is the crucial part and the efficiency of tridiagonalization will decide the total performance for computing all eigenvalues.

Next, Figure 2 shows the performances in FLOPS by the tridiagonalization

routine for 4–1024 PEs in Figure 2. From Figure 2 it is clear that the per-

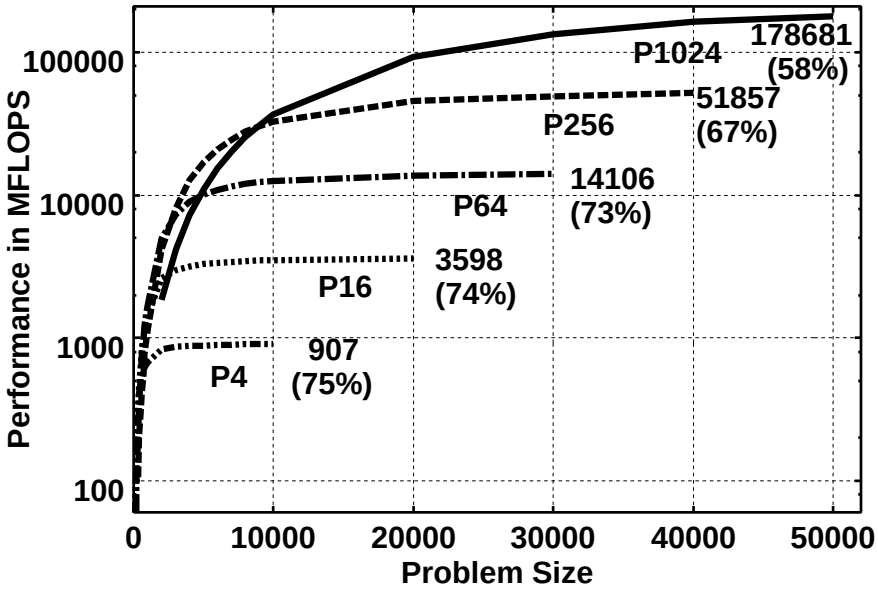


Fig. 2. Performances in parallel tridiagonalization (MFLOPS, the percentages in parentheses are percentages of the theoretical peak performance). Total calculation amount of $8/3n^3$ is used for computing the MFLOPS values.

formance for 4 PEs saturates at 75% of the theoretical peak performance of 300×4 MFLOPS. The saturated performance for 1024 PEs was 178 GFLOPS with our algorithm. Figure 2 also shows that the efficiency decreases as the number of PEs increases. One of the reasons is because when the number of PEs increases, the ratio of calculation time to the total execution time decreases.

4.2 Comparison to the ScaLAPACK

4.2.1 Experimental results

Table 6 and Table 7 show execution time of ScaLAPACK tridiagonalization (hereafter SLP TRD) routine and ours (hereafter our TRD) respectively. We use the HITACHI optimized ScaLAPACK version 1.2 [12]. Its communication library used is PVM, and the PBLAS which is computational kernel for ScaLAPACK is optimized by HITACHI limited.

The SLP TRD is implemented by using block-cyclic distribution, blocked algorithm, and symmetry of the matrix [21]. By using blocked algorithm, size of blocking (BL) can greatly affect performance of the ScaLAPACK. Table 5 shows this fact. From the result of Table 5, we found that varying BL speeds up 3.2 times with respect to the execution time in $BL = 1$.

According to [12], if the problem size n is less than 4000, BL should be 60, and if n is over 4000, desirable BL of 100 is a good choice on the SR2201. Considering these values, we evaluated the performance of SLP TRD routines under $BL = \{40, 60, 80, 100, 120\}$ to find which BL gives the best performance. For PE grid, squared grid of $\sqrt{p} \times \sqrt{p}$ is best according to [12]. We tried to measure execution time in the PE grid when number of PEs is large. When the number of PE is small, such as 4, 32, and 64, we measured execution time in all combinations for the PE grid to find which PE grid gives the best performance. The size of BL and the execution time are included in Tables 6 and 7.

Table 5
Execution time of the ScaLAPACK for varying BL in seconds. (SR2201, $n = 8000$, 256 PEs (PE grid:16×16))

BL	1	2	5	10	15	20	25	30
SLP TRD	517.37	296.53	201.97	174.02	171.50	156.86	157.28	155.70
Speedup	1.00	1.74	2.56	2.97	3.01	3.29	3.28	3.32

Figure 3 shows execution time for various number of PEs. From Figure 3(a), our TRD was about 2–6 times as fast as SLP TRD for a 2000 × 2000 matrix. On the other hand, from Figure 3(b), when matrix dimension was 8000, SLP TRD was faster than our TRD on 4–16 PEs. However, when the number of PEs increases, our TRD was faster than SLP TRD. The threshold number of PEs was about 16.

Figure 4 shows speedup ratio for a 8000 × 8000 matrix. From Figure 4, we could find the fact that our TRD obtained 40 times speedup, while SLP TRD obtained 10 times speedup only.

4.2.2 Discussion

From the results in Table 6 and Table 7, the conclusion is that if local matrix sizes were small enough, execution time of our TRD were 2–5 times faster than the SLP TRD. This is caused by the following two reasons:

(I) Our TRD has better load balance than the SLP TRD, since our TRD is permanently set to $BL = 1$ while SLP TRD allows to the arbitrary values;

Table 6
Performance for tridiagonalization I (SR2201). Unit is in second.

(a) Case of $PE = 4$

Size	SLP TRD (Grid,BL)	Our TRD (Grid)	SLP / Ours
100	0.02 (1×4, 100)	0.056 (2×2)	0.35
200	0.48 (1×4, 100)	0.133 (2×2)	3.6
400	1.73 (1×4, 40)	0.475 (2×2)	3.6
800	6.01 (1×4, 40)	2.454 (2×2)	2.4
1000	9.32 (2×2, 40)	3.785 (2×2)	2.4
2000	41.90 (2×2, 40)	26.937 (2×2)	1.5
4000	231.10 (2×2, 40)	242.010 (2×2)	0.95
8000	1422.69 (2×2, 100)	1962.512 (2×2)	0.72

(b) Case of $PE = 16$

Size	SLP TRD (Grid,BL)	Our TRD (Grid)	SLP /Ours
100	0.03 (1×16, 100)	0.082 (4×4)	0.36
200	0.82 (8×2, 100)	0.195 (4×4)	4.2
400	1.92 (1×16, 40)	0.419 (4×4)	4.5
800	5.48 (4×4, 40)	1.733 (4×4)	3.1
1000	7.53 (2×8, 40)	1.824 (4×4)	4.1
2000	23.00 (4×4, 40)	8.649 (4×4)	2.6
4000	92.21 (4×4, 60)	56.239 (4×4)	1.6
8000	474.49 (4×4, 60)	490.346 (4×4)	0.96

(c) Case of $PE = 64$

Size	SLP TRD (Grid,BL)	Our TRD (Grid)	SLP / Ours
100	0.21 (4×16, 100)	0.153 (8×8)	1.3
200	0.98 (1×64, 100)	0.278 (8×8)	3.5
400	2.82 (4×16, 100)	0.638 (8×8)	4.4
800	6.60 (8×8, 40)	1.402 (8×8)	4.7
1000	8.79 (8×8, 40)	1.612 (8×8)	5.4
2000	20.73 (8×8, 40)	5.105 (8×8)	4.0
4000	57.6 (8×8, 40)	19.631 (8×8)	2.9
8000	210.49 (8×8, 60)	119.065 (8×8)	1.7

Table 7
Performance for tridiagonalization II (SR2201). Unit is in second.

(a) Case of $PE = 128$

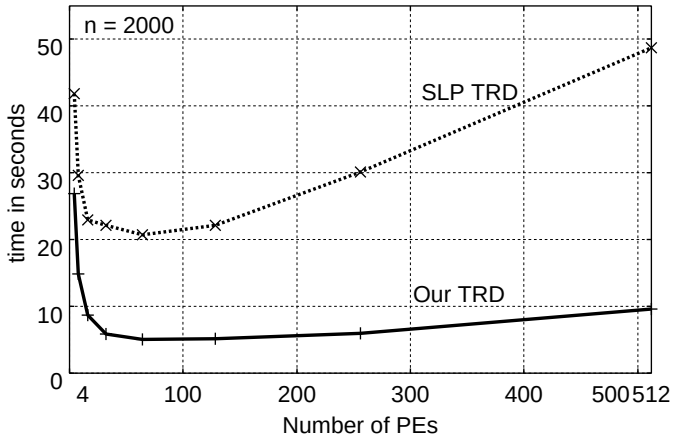
Size	SLP TRD (Grid,BL)	Our TRD (Grid)	SLP / Ours
200	1.93 (8×16, 100)	0.462 (8×16)	4.1
400	3.78 (8×16, 60)	0.860 (8×16)	4.3
800	7.68 (8×16, 80)	1.650 (8×16)	4.6
1000	9.74 (8×16, 100)	2.109 (8×16)	4.6
2000	22.17 (8×16, 40)	5.122 (8×16)	4.3
4000	54.13 (8×16, 40)	15.420 (8×16)	3.5
8000	162.01 (8×16, 40)	70.422 (8×16)	2.3
10000	245.60 (8×16, 40)	123.891 (8×16)	1.9

(b) Case of $PE = 256$

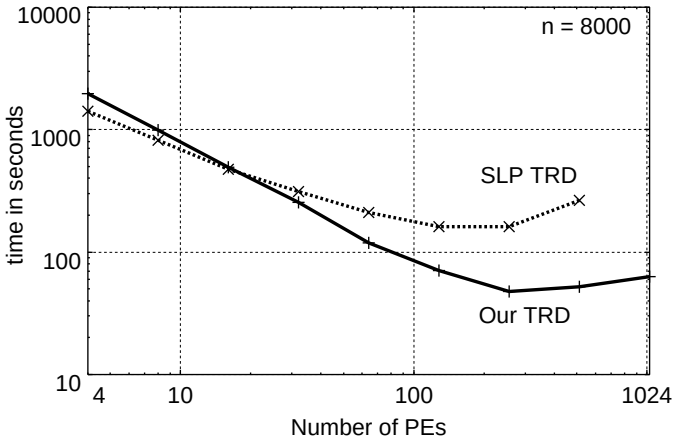
Size	SLP TRD (Grid,BL)	Our TRD (Grid)	SLP / Ours
400	5.69 (16×16, 60)	1.373 (16×16)	4.1
800	10.17 (16×16, 80)	2.480 (16×16)	4.1
1000	12.89 (16×16, 100)	3.217 (16×16)	4.0
2000	30.12 (16×16, 40)	5.964 (16×16)	5.0
4000	67.29 (16×16, 40)	14.338 (16×16)	4.6
8000	161.76 (16×16, 100)	47.906 (16×16)	3.3
10000	226.11 (16×16,40)	79.889 (16×16)	2.8
20000	774.10 (16×16, 60)	454.267 (16×16)	1.7

(c) Case of $PE = 512$

Size	SLP TRD (Grid,BL)	Our TRD (Grid)	SLP / Ours
1000	26.34 (16×32, 40)	4.552 (16×32)	5.7
2000	48.76 (16×32, 80)	9.613 (16×32)	5.0
4000	111.93 (16×32, 40)	20.484 (16×32)	5.4
8000	265.77 (16×32, 40)	52.397 (16×32)	5.0
10000	325.76 (16×32, 60)	81.450 (16×32)	3.9
20000	827.96 (16×32, 40)	302.541 (16×32)	2.7



(a) Case of $n = 2000$



(b) Case of $n = 8000$

Fig. 3. Execution time for SLP TRD and Our TRD in the tridiagonalization (SR2201).

(II) Our TRD has a lower communication complexity than the SLP TRD because of the non-symmetry in our TRD.

As for the reason (I), Katagiri and Kanada [16], Stranzdings [22], and Hendrickson *et al.* [10] point out the following. Parallel libraries must be constructed by using different blocking factors of block length in data distribution (BDD) and block length in blocking algorithm (BBA). The BBA does not depend on the data distribution. Therefore, the value of BDD must be taken as small as possible from the sake of parallel performance. In ScaLAPACK, however, the value of BDD is equal to BBA because of easy implementation of their libraries. This implementation policy causes poor load balancing on MPP environments. Our process supports (Cyclic, Cyclic) distribution (the value of BDD is 1) only. Therefore, our routine never suffers from poor load balancing.

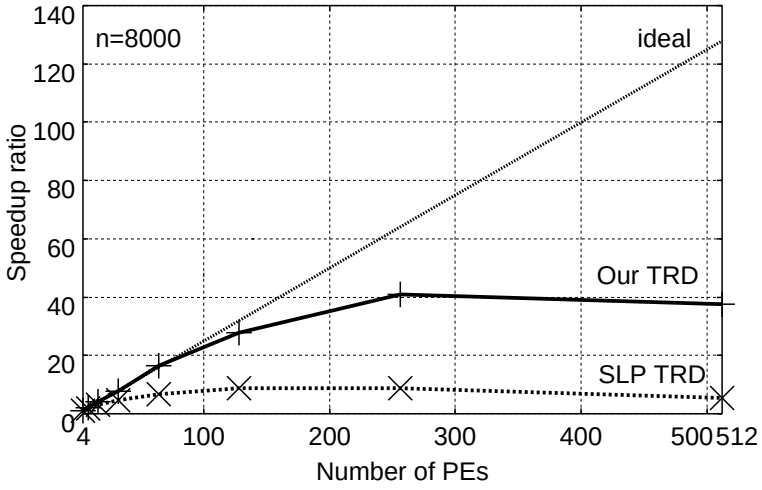


Fig. 4. Speedup ratios between SLP TRD and Our TRD for the tridiagonalization ($n = 8000$, SR2201).

The reason (II) shows the fact that when n/p is very small, algorithms by the HTN will be faster than the HTS. As already mentioned, the HTN algorithms have less communication with respect to the HTS. Of course, when n/p is large enough, execution time will mainly depend on computation time since computational complexity is $O(n^3)$. This can be explained by Table 6(a), $p = 4$, $n = 8000$ case. In this case, SLP TRD was about 1.3 times faster than our TRD. However, when the number of PEs increases, a larger matrix is needed to exceed the performance of our TRD. For example, Table 6(a) shows that the execution time of a 4000×4000 matrix was 231 seconds (SLP TRD) vs. 242 seconds (our TRD), and the ratio of executions was 0.95, and Table 6(b) shows that the execution time of a 8000×8000 matrix were 474 seconds (SLP TRD) vs. 490 seconds (our TRD), and the ratio of executions was 0.96. This means that to exceed our TRD, matrix sizes must be large in SLP TRD, and hence, the execution time will also be large.

From Table 6(a) ($p = 4$, $n = 800$, and $n = 8000$), we can determine the factors in the formula (1): $\delta_1 \approx 4268/(8000)^3$, $\gamma_1 \approx 3.58 \times 10^{-6}$, $\delta_2 \approx 5888/(8000)^3$, and $\gamma_2 \approx 3.83 \times 10^{-7}$. By using these factors, we will obtain the factor of $C_{\gamma/\delta} \approx 1011$, and this shows a fact that the $C_{\gamma/\delta}$ can have a considerable value. This factor affects execution time for an application. In applications in the chemical field, 6000 or more consecutive diagonalizations are required [23], which cannot be executed in parallel because each diagonalization depends on previous ones. Therefore in such applications, the execution time per diagonalization is limited, since total execution time becomes enormous. Even if execution time per one diagonalization is 100 seconds, total execution time will be about 166 hours (about 7 days!) From the results in Table 6 and Table 7, diagonalizations time under 100 seconds were about 2–5 times faster than SLP TRD. This shows that 35 days diagonalizations by using SLP TRD will be reduced to 7 days by our approach. Therefore, executing small size

diagonalization at high speed is very important. For such applications, our routine will be a more effective tool than the conventional routines.

5 Conclusion

The authors formulated, implemented and evaluated a parallel routine which can calculate all eigenvalues on an MPP system. By using the Householder tridiagonalization based on a non-symmetry algorithm and the (Cyclic, Cyclic) data distribution, we could obtain a better performance than the ScaLAPACK routine which is widely used as a parallel library. This effect becomes stronger when the number of PEs increases. Therefore, our process is very efficient on MPPs.

Our algorithm can attain high performance on an MPP system with as many as 1024 processors, even when the order of the matrices is relatively small, e.g. the order of 10,000. This is very important for relevant practical applications, where many diagonalizations for such matrices are required so often. Performing each diagonalization on one PE is not an essential solution because it can be a hot-spot by the sequential processing of the diagonalization. Therefore, performing parallel diagonalization with small problem at high performance is crucial.

In RISC based processors, it is known that blocking algorithms are more efficient than non-blocking algorithms [21]. However, blocking algorithms increase communication complexity for tridiagonalization [16]. In addition, using symmetry requires complex communication. Therefore, efficient inner processor algorithms and inter processor algorithms are different. To accomplish high performance, we have to analyze the parallel performance theoretically. The analysis and implementation of non-blocking vs. blocking algorithms, and symmetry vs. non-symmetry algorithms are the parts of future work.

Acknowledgments

The authors are much obliged to Dr. Daisuke TAKAHASHI now at Saitama university, and Dr. Aad van der Steen at Utrecht university for helpful comments. We also would like to express our sincere thanks to the anonymous reviewer of this paper.

This research is partly supported by Grant-in-Aid for Scientific Research on Priority Areas “Discovery Science” from the Ministry of Education, Science

and Culture, Japan, and Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

References

[1] 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 Users' Guide* (SIAM, USA, 1997).

[2] T. Boku, K. Itakura, H. Nakamura and K. Nakazawa, CP-PACS: A Massively Parallel Processor for Large Scale Scientific Calculations, *Proceedings of International Conference on Supercomputing'97* (Vienna, Austria, 1997) 108–115.

[3] R. Brent, L. Grosz, D. HarrarII, M. Hegland, M. Kahn, G. Keating, G. Mercer, O. Nielsen, M. Osborne and B. Zhou, Development of a Mathematical Subroutine Library for Fujitsu Vector Parallel Processors, *Proceedings of International Conference on Supercomputing'98* (Melbourne, Australia, 1998) 13–20.

[4] H. Chang, S. Utku, M. Sakama and D. Rapp, A Parallel Householder Tridiagonalization Stratagem Using Scattered Square Decomposition, *Parallel Computing* **6** (1988) 297–311.

[5] S. Chinchalkar and T. Coleman, Computing Eigenvalues and Eigenvectors of a Dense Real Symmetric Matrix on the NCUBE 6400, Technical Report 91-074, Cornell University, Theory Center, 1991.

[6] J. Demmel, *Applied Numerical Linear Algebra* (SIAM, USA, 1997).

[7] J. Demmel and K. Stanley, The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers, Technical Report UT-GS-94-254, University of Tennessee, Knoxville, 1994.

[8] E. Elmroth and F. Gustavson, New Serial and Parallel Recursive QR Factorization Algorithms for SMP Systems, *Lecture Notes in Computer Science* No.**1541**, (springer-verg, 1998) 120–128.

[9] D. Gimenez, V. Hernandez, R. van de Geijn and A. Vidal, A Jacobi Method by Blocks on a Mesh of Processors, *Concurrency : Practice and Experience* **9(5)** (1997) 391–411.

[10] B. Hendrickson, E. Jessup and C. Smith, Toward an Efficient Parallel Eigensolver for Dense Symmetric Matrices, *SIAM J. Sci. Comput.* **20(3)** (1999) 1132–1154.

[11] B. A. Hendrickson, and D. E. Womble, The Tours-Wrap Mapping for Dense Matrix Calculation on Massively Parallel Computers, *SIAM Sci. Comput.* **15(5)** (1994) 1201–1226.

[12] HITACHI Ltd., Using ScaLAPACK and PBLAS Libraries for the HITACHI SR2201, *Computer Centre News, the University of Tokyo* **30(2)** (1998) 36–58. in Japanese.

[13] T. Katagiri, A Study on Large Scale Eigensolvers for Distributed Memory Parallel Machines, Ph.D. Thesis, The University of Tokyo, 2001.

[14] T. Katagiri and Y. Kanada, Performance Evaluation of Householder Method for the Eigenvalue Problem on Distributed Memory Architecture Parallel Machine, *IPSJ SIG Notes 96-HPC-62* (1996) 111–116. in Japanese.

[15] T. Katagiri and Y. Kanada, A Parallel Implementation of Eigensolver and its Performance, *IPSJ SIG Notes, 97-HPC-69* (1997) 49–54. in Japanese.

[16] T. Katagiri and Y. Kanada, Performance Evaluation of Blocked Householder Algorithm on Distributed Memory Parallel Machine, *Trans. IPS. Japan* **39(7)** (1998) 2391–2394. in Japanese.

[17] S. Lo, B. Philippe and A. Sameh, A Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem, *SIAM J. Sci. Stat. Comput.* **8(2)** (1987) s155–s165.

[18] K. Murata, M. Natori and Y. Karaki, *Large Scale Numerical Simulations* (Iwanami Shoten, Japan, 1990) 157–165. in Japanese.

[19] K. Naono, M. Igai and Y. Yamamoto, Development of a Parallel Eigensolver and its Evaluation, *Proceedings of Joint Symposium on Parallel Processing'96* (Tokyo, Japan, 1996) 9–16. in Japanese.

[20] J. Reeve and M. Heath, An Efficient Parallel Version of the Householder-QL Matrix Diagonalization Algorithm, *Parallel Computing* **25** (1999) 311–319.

[21] K. S. Stanley, Execution Time of Symmetric Eigensolver, Ph.D. Thesis, The University of California at Berkeley, 1997.

[22] P. Strazdins, Optimal Load Balancing Techniques for Block-Cyclic Decompositions for Matrix Factorization, Technical Report TR-CS-98-10, The Australian National University, Joint Computer Science Technical Report Series, 1998.

[23] S. Tajima, T. Katagiri, Y. Kanada and U. Nagashima, Parallel Processing of Molecular Geometry Parameter Optimization by Extended Huckel Method – An Attempt of Simple Fast Generation of Molecular Structure – , *The Journal of Chemical Software* **6(2)** (2000). in Japanese.