Proceedings of ALGORITMY 2012
pp. 151–160

PERFORMANCE OF THE BLOCK JACOBI METHOD FOR THE SYMMETRIC EIGENVALUE PROBLEM ON A MODERN MASSIVELY PARALLEL COMPUTER*

YUUSUKE TAKAHASHI†, YUUSUKE HIROTA‡, AND YUSAKU YAMAMOTO§

Abstract. In this paper, we consider the solution of a medium-size symmetric eigenvalue problem on a massively parallel computer using the block Jacobi method. We compare parallel cyclic block Jacobi methods using 1-dimensional and 2-dimensional data distribution and show that the latter has advantages in terms of the number of processors that can be used and the frequency and volume of interprocessor communication. The 2-dimensional scheme has a disadvantage that some part of the algorithm can be executed by only $\sqrt{p}$ processors, where p is the number of processors. However, a simple analysis shows that this does not degrade weak scalability. This analysis is supported by performance evaluation on the University of Tokyo's T2K supercomputer using up to 1024 cores. We also discuss how to improve the performance of our imlementation from three viewpoints.

Key words. eigenvalue, block Jacobi method, parallel computing

AMS subject classifications. 15A18, 34L16, 65F15

1. Introduction.

Solution of the symmetric eigenvalue problem plays a vital role in many fields of science and engineering. In a certain kind of molecular dynamics simulation based on quantum mechanics, there is a need to solve a medium-size ($N \sim 10,000$) symmetric eigenvalue problem millions of times to compute time evolution of a system. Thus speeding up the solution process is strongly desired. Solution of a medium-size symmetric eigenvalue problem is also important in molecular orbital methods.

The standard procedure for solving the symmetric eigenvalue problem is based on tridiagonalization [7]. In this procedure, the input matrix is first transformed to a symmetric tridiagonal matrix by orthogonal transformations, then the eigenvalues and eigenvectors of the tridiagonal matrix are computed, and finally, the eigenvectors of the original matrix are formed by back-transformation. This approach is optimal from the viewpoint of computational cost. It is adopted by LAPACK [1] and is widely used. There are also several high performance implementations, such as ScaLAPACK [5], for massively parallel distributed-memory machines. However, these solvers are designed with very large-scale problems in mind. To fully exploit the potential of a parallel machine with thousands of cores, these solvers require matrices of order more than hundreds of thousands.

In this paper, we consider solving a medium-size symmetric eigenproblem with thousands of cores. To this end, we use the block Jacobi method. The block Jacobi

*This work was supported by Core Research for Evolutional Science and Technology (CREST) Program "Highly Productive, High Performance Application Frameworks for Post Petascale Computing" of Japan Science and Technology Agency.

†Department of Computational Science, Kobe University, Kobe, 657-8501, Japan (115x213x@stu.kobe-u.ac.jp).

‡Department of Computational Science, Kobe University, Kobe, 657-8501, Japan (hirota@stu.kobe-u.ac.jp).

§Department of Computational Science, Kobe University, Kobe, 657-8501, Japan / JST-CREST (yamamoto@cs.kobe-u.ac.jp).

method usually requires computational work several times larger than that of the tridiagonalization-based method. However, it has several advantages. First, it has large-grain parallelism; when the block size is L, each processor can perform $O(L^3)$ floating-point operations between synchronization points. Second, most of the computational work is done in the form of matrix-matrix multiplication. This feature is especially suited for modern microprocessors. Finally, the algorithm is simple and consists of only two types of computations, namely, solution of a small eigenproblem and matrix multiplication. The purpose of this paper is to evaluate the performance of the Jacobi-based symmetric eigensolver on a modern massively parallel distributed-memory computer and find out performance bottlenecks and opportunities for possible improvements.

The paper is organized as follows. In Section 2, we explain the block Jacobi method and strategies for its parallelization. We compare two parallelization schemes, based on the 1-dimensional and 2-dimensional distributions, and argue that the latter is more suited in a massively parallel environment. In Section 3, we present performance results on the University of Tokyo's T2K supercomputer with up to 1024 cores. Finally, Section 4 gives some concluding remarks.

2. The Block Jacobi Method and Its Parallel Implementation.

2.1. The cyclic block Jacobi method. Let $A \in \mathbf{R}^{N\times N}$ be a symmetric matrix. In the block Jacobi method, we divide A into square blocks of size $L \times L$. For simplicity, we assume that N is divisible by L and let $W = N/L$. Let us denote the (I, J)-th block of A by A_{IJ}. In a variant called the cyclic block Jacobi method [7], we pick up one of the upper off-diagonal blocks, say A_{IJ} $(I < J)$, and apply an orthogonal transformation on A that annihilates the block. By repeating this for the $W(W-1)/2$ off-diagonal blocks, one sweep is completed. Of course, annihilating a block will produce nonzero elements into already annihilated blocks, so the entire sweep must be repeated until the matrix becomes sufficiently close to a diagonal matrix.

Let the orthogonal matrix used to annihilate the A_{IJ} be denoted by $P^{(I,J)}$. $P^{(IJ)}$ can be found as follows. Let

$$\tilde{A} = \begin{bmatrix} A_{II} & A_{IJ} \\ A_{JI} & A_{JJ} \end{bmatrix} \in \mathbf{R}^{2L\times 2L} \tag{2.1}$$

and let $\tilde{P} \in \mathbf{R}^{2L\times 2L}$ be an orthogonal matrix that diagonalizes $\tilde{A}$, that is,

$$\tilde{P}^T \tilde{A} \tilde{P} = \tilde{\Lambda}, \tag{2.2}$$

where $\tilde{\Lambda}$ is a diagonal matrix. Now, divide $\tilde{P}$ into four $L \times L$ submatrices and embed each submatrix into an appropriate part of I_N to get an $N \times N$ orthogonal matrix $P^{(I,J)}$. Then, from Eq. (2.2), we know that the (I, J)-th and the (J, I)-th submatrix of $\left(P^{(I,J)}\right)^T AP^{(I,J)}$ is zero. Thus we have obtained a desired orthogonal matrix. By multiplying $\left(P^{(I,J)}\right)^T$ on the left, only the I-th and J-th block rows of A are updated. Similarly, by multiplying $P^{(I,J)}$ on the right, only the I-th and J-th block columns of A are updated. The eigenvectors of A can be computed by accumulating the matrix $P^{(I,J)}$ used at each step. The algorithm of the block Jacobi method is shown as Algorithm 1. Although the order of picking up off-diagonal blocks is specified here, this can be changed as long as each off-diagonal block is selected once in each sweep.

After the convergence, the diagonal elements of A are the eigenvalues and the columns of P are the eigenvectors.

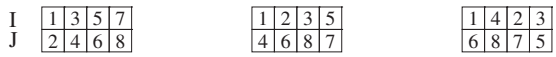
As can be seen from Algorithm 1, the algorithm consists of only two kinds of operations, namely, solution of a $2L \times 2L$ eigenproblem (step 4) and matrix multiplications (steps 6, 7 and 8). For each (I, J), the computational cost of step 4 is about $32L^3$, while the cost for steps 6, 7 and 8 is $24L^2N$. Hence, if $L \ll N$, most of the computation is done in the form of matrix multiplication. Thus the method is suited for modern high performance architectures.

[Algorithm 1: Cyclic block Jacobi method]
1: $P = I_N$
2: **for** $n = 1, 2, \ldots$ **do**
3: **for** $(I, J) = (1, 2), (1, 3), \ldots, (1, W), (2, 3), \ldots, (W - 1, W)$ **do**
4: Find a matrix $\tilde{P}$ that diagonalizes $\tilde{A} = \begin{bmatrix} A_{II} & A_{IJ} \\ A_{JI} & A_{JJ} \end{bmatrix}$.
5: Extend $\tilde{P}$ to an $N \times N$ matrix $P^{(I,J)}$.
6: $A \leftarrow \left(P^{(I,J)}\right)^T A$
7: $A \leftarrow AP^{(I,J)}$
8: $P \leftarrow PP^{(I,J)}$
9: **end for**
10: **end for**

2.2. Parallelism of the cyclic block Jacobi method. In the cyclic block Jacobi method, assume that the (I_2, J_2)-th block is annihilated right after the (I_1, J_1)-th block and all of the I_1, I_2, J_1 and J_2 are different integers. In that case, since the four blocks that determines $P^{(I_2,J_2)}$, namely, A_{I_2,I_2}, A_{I_2,J_2}, A_{J_2,I_2} and A_{J_2,J_2}, are unchanged by the orthogonal transformation by $P^{(I_1,J_1)}$, it does not matter whether $P^{(I_2,J_2)}$ is computed before or after the application of $P^{(I_1,J_1)}$. Thus the computation of $\tilde{P}$ can be done for (I_1, J_1) and (I_2, J_2) in parallel. Moreover, pre-multiplication (step 6) by $\left(P^{(I_1,J_1)}\right)^T$ and $\left(P^{(I_2,J_2)}\right)^T$ can be done in parallel. This is also true of the post-multiplication (step 7) and update of P (step 8). Thus there is a large-grain parallelism.

More generally, assume that all of the $I_1, I_2, \ldots, I_{W/2}$ and $J_1, J_2, \ldots, J_{W/2}$ are different integers. Then step 4 can be done for the $W/2$ sets, (I_1, J_1), $(I_2, J_2), \ldots$, $(I_{W/2}, J_{W/2})$, simultaneously. Steps 6, 7 and 8 can also be parallelized in the same manner. Since there are $W(W-1)/2$ off-diagonal blocks in the upper triangular part, the loop over off-diagonal blocks (step 3) can be completed in $W - 1$ steps if the sets at each step are chosen judiciously. In this study, we use the standard round-robin ordering [6]. To be concrete, let $W = 8$. In the first step, we use the pairs $(1, 2)$, $(3, 4)$, $(5, 6)$ and $(7, 8)$, as illustrated in Fig. 2.1 (a). Thus we eliminate the blocks A_{12}, A_{34}, A_{56} and A_{78} simultaneously. Before the second step, we fix the number "1" and rotate other numbers clockwise to get the arrangement in Fig. 2.1 (b). Thus the pairs at the second step is $(1, 4)$, $(2, 6)$, $(3, 8)$ and $(5, 7)$ and the blocks A_{14}, A_{26}, A_{38} and A_{57} are eliminated. Before the third step, we again fix "1" and rotate other numbers clockwise to get the arrangement in Fig. 2.1 (c), which corresponds to the elimination of A_{16}, A_{48}, A_{27} and A_{35}. By repeating this 7 ($=W - 1$) times, we can generate all the pairs corresponding to all the off-diagonal blocks. Generalization of this method to a general value of W is straightforward.

The algorithm of the parallel cyclic block Jacobi method is shown as Algorithm



(a) Step 1 (a) Step 2 (a) Step 3

FIG. 2.1. *The round-robin ordering.*

2. Here, the pairs $(I_1^{(K)}, J_1^{(K)})$, $(I_2^{(K)}, J_2^{(K)})$ ..., $(I_{W/2}^{(K)}, J_{W/2}^{(K)})$ to be eliminated at the Kth step are determined as above. In this algorithm, the innermost for loop can be executed in parallel.

[Algorithm 2: Parallel cyclic block Jacobi method]
1: $P = I_N$
2: **for** $n = 1, 2, \ldots$ **do**
3: **for** $K = 1, 2, \ldots, W-1$ **do**
4: **for** $(I, J) = (I_1^{(K)}, J_1^{(K)}), (I_2^{(K)}, J_2^{(K)}) \ldots, (I_{W/2}^{(K)}, J_{W/2}^{(K)})$ **do**
5: Find a matrix $\tilde{P}$ that diagonalizes $\tilde{A} = \begin{bmatrix} A_{II} & A_{IJ} \\ A_{JI} & A_{JJ} \end{bmatrix}$.
6: Extend $\tilde{P}$ to an $N \times N$ matrix $P^{(I,J)}$.
7: $A \leftarrow \left(P^{(I,J)}\right)^T A$
8: $A \leftarrow AP^{(I,J)}$
9: $P \leftarrow PP^{(I,J)}$
10: **end for**
11: **end for**
12: **end for**

2.3. Parallel implementation.

2.3.1. The 1-dimensional distribution. To implement the parallel cyclic block Jacobi method on a distributed memory parallel machine, one has to determine the data distribution. Most of the existing studies use the 1-dimenstional distribution [4][6][9][10]. In this approach, $W/2$ processors are used. The ℓth processor ($\ell = 1, 2, \ldots, W/2$) takes charge of the $I_\ell^{(K)}$th and $J_\ell^{(K)}$th block columns and executes lines 5 through 9 of Algorithm 2 for $(I, J) = (I_\ell^{(K)}, J_\ell^{(K)})$. Thus steps 5, 6, 8 and 9 can be done completely independently, because step 5 uses submatrices only in the $I_\ell^{(K)}$th and $J_\ell^{(K)}$th block columns and steps 8 and 9 are orthogonal transformations involving only the $I_\ell^{(K)}$th and $J_\ell^{(K)}$th block columns. To perform step 7, however, the ℓth processor needs $P^{(I,J)}$ for $(I, J) = (I_1^{(K)}, J_1^{(K)}), (I_2^{(K)}, J_2^{(K)}) \ldots, (I_{W/2}^{(K)}, J_{W/2}^{(K)})$. This incurs all-to-all broadcast. After finishing the for loop of lines 4 through 10, the processors must exchange the block columns so that the ℓth processor has the $I_\ell^{(K+1)}$th and $J_\ell^{(K+1)}$th block columns. By choosing the pairs $(I_1^{(K)}, J_1^{(K)})$, $(I_2^{(K)}, J_2^{(K)})$ ..., $(I_{W/2}^{(K)}, J_{W/2}^{(K)})$ $(K = 1, 2, \ldots, W-1)$ judiciously, one can ensure that the sets $\{I_\ell^{(K)}, J_\ell^{(K)}\}$ and $\{I_\ell^{(K+1)}, J_\ell^{(K+1)}\}$ have one element in common for all ℓ and K. Then each processor needs to send and receive only one block column at each step, instead of two block columns. Also, in the case where the interprocessor network is a ring, one can ensure that the exchange of block columns occur only between adjacent processors. Much work has been done on these aspects. See [4], [6], [9] and [10] for example.

The 1-dimensional distribution has the advantage that it can achieve nearly 100%

utilization of the processors. In fact, all the processors execute steps 5 through 9 of Algorithm 2 and there is no idle time. However, it has also several disadvantages. The greatest problem is that the number of processors p that can be used is severely limited. Of course, $p \leq N/2$ must hold. Moreover, to achieve modest performance in the matrix multiplication part, the number of columns and rows of the matrices must be at least 100. Hence, the number of processors that can be used to solve a problem of $N = 10,000$ is limited to about 100, which contradicts our objective of utilizing thousands of processors to solve a medium-size problem. The 1-dimensional distribution also has the problem that all-to-all broadcast is needed to share $P^(I,J)$ $((I,J) = (I_1^{(K)}, J_1^{(K)}), (I_2^{(K)}, J_2^{(K)}) \ldots, (I_{W/2}^{(K)}, J_{W/2}^{(K)}))$ among all the processors and that the exchange of entire block columns is needed between processors. Such communication patterns are not desirable in a massively parallel environment.

2.3.2. The 2-dimensional distribution. To avoid these problems, we decided to use the 2-dimensional distribution in this study. In this approach, $W^2/4$ processors are used and the (ℓ, m)th processor $(\ell, m = 1, 2, \ldots, W/2)$ takes charge of the four blocks, $A_{I_\ell^{(K)}, I_m^{(K)}}$, $A_{I_\ell^{(K)}, J_m^{(K)}}$, $A_{J_\ell^{(K)}, I_m^{(K)}}$ and $A_{J_\ell^{(K)}, J_m^{(K)}}$. In this approach, the processors on the diagonal $(l = m)$ execute step 5 of Algorithm 2 and broadcast the orthogonal matrix $\tilde{P}$ to processors in the same block row (processors with the same ℓ) and the same block column (processors with the same m). Then all the processors execute steps 7 through 9 using the orthogonal matrices just received. The algorithm of the parallel cyclic block Jacobi method using 2-dimensional distribution is shown as Algorithm 3. Here, the algorithm is written in an MPI-like fashion that is to be executed on all of the $W^2/4$ processors.

The 2-dimensional distribution has several advantages. First, it is easier to use more processors. Even if we put a restriction that the matrix size appearing in matrix multiplications is at least 100×100, we can use 10,000 processors to solve a problem of $N = 10,000$ [1]. Second, there is no need for all-to-all broadcast; the orthogonal matrix $\tilde{P}_\ell^{(K)}$ needs to be broadcast only to the processors within the same block row or column. Third, in the exchange phase, each processor needs to exchange only up to four blocks and not an entire block column. The second and third points greatly reduce the frequency and volume of interprocessor communication.

The disadvantage of the 2-dimensional distribution is that only the processors on the diagonal work in step 11 and other processors become idle. Thus the percentage of processor utilization is lower than that in the 1-dimensional distribution. However, from the viewpoint of solving a medium-size problem as quickly as possible, this is not a severe problem, as long as step 11 does not cause a sequential bottleneck.

To analyze the influence of step 11 on the overall performance, we perform a weak scalability analysis. More precisely, we fix the size of the matrix allocated to one processor to $2L \times 2L$ and estimate the computation time of steps 11, 18, 19 and 20 for each K as a function of p, the number of processors. In that case, the matrix size is $N = 2Lp^{\frac{1}{2}}$. The results on the computation time are given in Table 2.1. Here, "Computation time" means the computation time per one active processor. From the table, it can be seen that the ratio of the time for step 11 to the total computation time is constant that is independent of p. This means that though the processor utilization in the 2-dimensional distribution is not perfect, it does not degrade weak

[1] Note that the size of the matrices appearing in the matrix multiplications is $2L \times 2L$. Thus we require that $L = 50$ and the number of processors is $W^2/4 = N^2/4L^2 = 10,000$.

scalability. On the other hand, the results on the communication time are given in Table 2.2. Here, s is the bandwidth of interprocessor communication (Bytes/sec) and "Communication time" means the time required to complete the communication. We assume that broadcast is done using the binary tree and exchange of one block can be done in one step. We can see from the table that the ratio of the communication time to the total computation time grows only slowly (like $\frac{1}{2}\log_2 p$) with p. From these analysis, we can conclude that the 2-dimensional distribution is suited for a massively parallel environment.

[Algorithm 3: Parallel cyclic block Jacobi method using 2-d distribution]

Get my processor number (ℓ, m).
$\tilde{A} = \begin{bmatrix} A_{I_\ell^{(K)}, I_m^{(K)}} & A_{I_\ell^{(K)}, J_m^{(K)}} \\ A_{J_\ell^{(K)}, I_m^{(K)}} & A_{J_\ell^{(K)}, J_m^{(K)}} \end{bmatrix}$
if $l = m$ then
    $\tilde{P} = I_{2L}$
else
    $\tilde{P} = O_{2L}$
end if
for $n = 1, 2, \ldots$ **do**
    for $K = 1, 2, \ldots, W-1$ **do**
        if $(l = m)$ **then**
            Find a matrix $\tilde{P}_\ell^{(K)}$ that diagonalizes $\tilde{A}$.
            Send $\tilde{P}_\ell^{(K)}$ to processors with number $(\ell, *)$.
            Send $\tilde{P}_\ell^{(K)}$ to processors with number $(*, \ell)$.
        else
            Receive $\tilde{P}_\ell^{(K)}$ from processor (ℓ, ℓ).
            Receive $\tilde{P}_m^{(K)}$ from processor (m, m).
        end if
        $\tilde{A} \leftarrow \left(\tilde{P}_\ell^{(K)}\right)^T \tilde{A}$
        $\tilde{A} \leftarrow \tilde{A}\tilde{P}_m^{(K)}$
        $\tilde{P} \leftarrow \tilde{P}\tilde{P}_m^{(K)}$
        Compute $I_\ell^{(K+1)}$, $J_\ell^{(K+1)}$, $I_m^{(K+1)}$ and $J_m^{(K+1)}$.
        Exchange the blocks with other processors to obtain blocks $A_{I_\ell^{(K+1)}, I_m^{(K+1)}}$, $A_{I_\ell^{(K+1)}, J_m^{(K+1)}}$, $A_{J_\ell^{(K+1)}, I_m^{(K+1)}}$ and $A_{J_\ell^{(K+1)}, J_m^{(K+1)}}$.
    end for
end for

TABLE 2.1
Computation time of steps 11, 18, 19 and 20 of Algorithm 3.

Step	Computation	Cost	Number of processors used	Computation time
11	Diagonalization of $\tilde{A}$	$32L^3 \times p^{\frac{1}{2}}$	$p^{\frac{1}{2}}$	$32L^3$
18	$\tilde{A} \leftarrow \left(\tilde{P}_\ell^{(K)}\right)^T \tilde{A}$	$8L^2N \times p^{\frac{1}{2}}$	p	$16L^3$
19	$\tilde{A} \leftarrow \tilde{A}\tilde{P}_m^{(K)}$	$8L^2N \times p^{\frac{1}{2}}$	p	$16L^3$
20	$\tilde{P} \leftarrow \tilde{P}\tilde{P}_m^{(K)}$	$8L^2N \times p^{\frac{1}{2}}$	p	$16L^3$

TABLE 2.2
Communication time of steps 12/15, 13/16 and 22 of Algorithm 3.

Step	Operation	Data size (in Bytes)	Number of steps	Communication time
12/15	Broadcast of $\tilde{P}_\ell^{(K)}$ within row ℓ	$4L^2 \times 8$	$\frac{1}{2}\log_2 p$	$(16L^2 \log_2 p)/s$
13/16	Broadcast of $\tilde{P}_\ell^{(K)}$ within column ℓ	$4L^2 \times 8$	$\frac{1}{2}\log_2 p$	$(16L^2 \log_2 p)/s$
22	Exchange of the blocks	$2 \times 4L^2 \times 8$	1	$64L^2/s$

3. Experimental Results.

3.1. Computational environments. We implemented a program of the parallel cyclic block Jacobi method based on Algorithm 3 and evaluated its performance. The program is written with C and parallelized using MPI. Diagonalization of $\tilde{A}$'s (diagonal blocks) is done using LAPACK routine dsyev and matrix multiplications are performed using BLAS routine DGEMM. The program was run on the University of Tokyo's T2K supercomputer, whose specification is listed in Table 3.1. As a test matrix, we used a random symmetric matrix whose entries are uniform random numbers in the interval $[0, 10]$. The block size is fixed to $L = 125$ and the matrix size N is set to $2Lp^{\frac{1}{2}}$, where p is the number of processor cores used. The values of p are 4, 16, 64, 256 and 1024, so the matrix sizes used in the test are 500, 1000, 2000, 4000 and 8000. The stopping criterion of the block Jacobi method is that the absolute values of all the off-diagonal elements are less than 10^{-10}.

TABLE 3.1
Specification of the University of Tokyo's T2K Supercomputer.

Item	Specification
Node	AMD Opteron Processor (2.3GHz, 4 cores) × 4 Up to 64 nodes (1024 cores) are used.
Memory	32Gbytes / Node
OS	Red Hat Enterprise Linux
Compiler	PGI C/C++ Compiler
BLAS & LAPACK	AMD Core math Library

3.2. Results. The execution time of our program for various values of p's are shown in Fig. 3.1. As can be seen from the graph, diagonalization of $\tilde{A}$'s occupies most of the time when p is small. As p increases, the time for broadcast and exchange increases. This is in consistent with our analysis in subsection 2.3 that the diagonalization of $\tilde{A}$'s accounts for only constant portion of the total execution time, although it is executed using only $\sqrt{p}$ cores. Ideally, in the present setting, the total execution time should increase proportionally with N, because the total computational work is expected to be $O(N^3)$ and the number of cores used is $O(N^2)$. However, the graph shows that the execution time increases more than twice when N is doubled. This is due to two reasons. First, the times for broadcast and exchange increase rapidly as p increases. This can be because of contention of messages on the interprocessor network. Second, the number of iterations before convergence increases with p. In fact, the number of outer iterations (line 8 of Algorithm 3) is 5, 6, 7, 8 and 10 for $p = 4$, 16, 64, 256 and 1024, respectively. Thus it seems that the number of iterations increases as $O(\log_2 \sqrt{p})$. Based on these observations, we will discuss possible improvements in the next subsection.

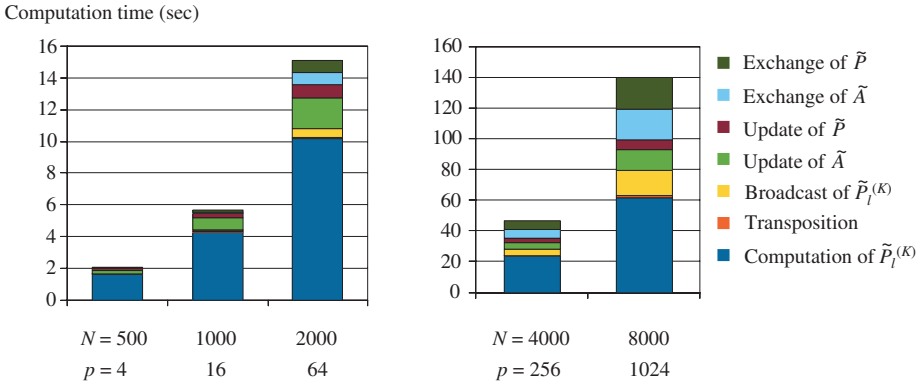


FIG. 3.1. *Execution time of the parallel cyclic block Jacobi method on the T2K supercomputer.*

3.3. Possible improvements. In view of the observations made in the previous subsection, we propose possible improvements on our program from three aspects.

3.3.1. Reducing the time for diagonalization of $\tilde{A}$. From Fig. 3.1, it can be seen that diagonalizaition of $\tilde{A}$ consumes nearly half of the execution time when $p = 1024$. In the present implementation, this part is done using LAPACK dsyev, which is based on the QR algorithm. By replacing it with a faster algorithm such as the divide-and-conquer or MR^3, the execution time of this part will be shortened. Another approach would be to parallelize each diagonalization. However, because the size of $\tilde{A}$ is small, say 100, in our target problem, it is unlikely that parallelization will produce large speedup. As an alternative, we can use the (block) Jacobi method also for this diagonalization. This choice provides us with larger parallel granularity, at the cost of increased computational work. Another advantage of using the Jacobi method is that, by using an appropriate variant of the Jacobi method, we can compute all the eigenvalues of A with high relative accuracy. We will investigate these possibilities in our future work.

3.3.2. Reducing the time of broadcast and exchange. As we mentioned in the previous subsection, the time for broadcast and exchange increases rapidly with p. This suggests existence of contentions of messages in these operations. The contentions may be avoidable if we allocate the submatrices to processors judiciously by taking the network topology into account.

3.3.3. Accelerating the convergence. Recently, it has been shown that the convergence of the block Jacobi method can be accelerated considerably by using the QR decomposition as a preprocessing stage [8][3]. Although most of the existing studies on this topic deal with Jacobi methods for the singular value decomposition, the same technique can be applied to Jacobi methods for eigenvalue problems as well. Also, the dynamic ordering approach [2][8], which chooses the off-diagonal blocks to be eliminated based on their Frobenius norm, may be useful for accelerating the convergence.

4. Conclusion. In this paper, we consider using the block Jacobi method for solving a medium-size symmetric eigenvalue problem on a massively parallel computer. We compared parallel cyclic block Jacobi methods using 1-dimensional and 2-dimensional data distribution and showed that the latter has advantages in terms of the number of processors that can be used and the frequency and volume of interprocessor communication. In contrast to the 1-dimensional scheme, the 2-dimensional scheme has a part that is executed by only $\sqrt{p}$ processors. However, a simple analysis shows that this part does not degrade weak scalability of the algorithm. This analysis is supported by performance evaluation on the University of Tokyo's T2K supercomputer using up to 1024 cores. We also discussed how to improve the performance of our program from three viewpoints. As a future work, we will port our program to the "K" supercomputer, which is the world's fastest supercomputer as of November 2011, and evaluate its performance on thousands of cores.

Acknowledgments. We would like to express our sincere gratitude for the anonymous reviewer, whose comments helped us much in improving the quality of this paper. We are also grateful to Professor Marian Vajteršic and Professor Gabriel Okša for providing valuable comments on our current research project and for pointing out important literatures on the dynamic ordering approach. This work is partially supported by Grants-in-Aid for Scientific Research from the Japan Society for the Promotion of Science, and Core Research for Evolutional Science and Technology (CREST) Program "Highly Productive, High Performance Application Frameworks for Post Petascale Computing" of Japan Science and Technology Agency (JST).

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1992.

[2] M. Bečka, G. Okša, and M. Vajteršic. Dynamic ordering for a parallel block-Jacobi SVD algorithm. *Parallel Computing*, 28:243–262, 2002.

[3] M. Bečka, G. Okša, M. Vajteršic, and R. Grigori. On iterative QR pre-processing in the parallel block-Jacobi SVD algorithm. *Parallel Computing*, 36:297–307, 2010.

[4] C. H. Bischof. *The two-sided block Jacobi method on a hypercube*, pages 612–618. SIAM, 1988.

[5] L. S. 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. C. Whaley. *ScaLAPACK User's Guide*. SIAM, 1997.

[6] R. P. Brent and F. T. Luk. The solution of the singular value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Statist. Comput.*, 6:69–84, 1985.

[7] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[8] G. Okša and M. Vajteršic. Efficient pre-processing in the parallel block-Jacobi SVD algorithm. *Parallel Computing*, 32:166–176, 2006.

[9] M. Vajteršic and M. Bečka. Block-SVD algorithms and their adaptation to hypercubes and rings. In N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms / Architecture Synthesis*, pages 175–181. IEEE Computer Society Press, 1997.

[10] B. B. Zhou and R. P. Brent. A parallel ring ordering algorithm for efficient one-sided Jacobi SVD computations. *J. of Parallel and Distributed Computing*, 42:1–10, 1997.