A Massively Parallel Dense Symmetric Eigensolver with Communication Splitting Multicasting Algorithm

Takahiro Katagiri[1] , Shoji Itoh[2]

[1] Information Technology Center, The University of Tokyo,
2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-8658, JAPAN
katagiri@cc.u-tokyo.ac.jp
[2] Advanced Center for Computing and Communication, RIKEN,
2-1 Hirosawa, Wako-shi, Saitama 351-0198, JAPAN
itosho@riken.jp

Abstract. In this paper, we propose a process grid free algorithm for a massively parallel dense symmetric eigensolver with a communication splitting multicasting algorithm. In this algorithm, a tradeoff exists between speed and memory space to keep the Householder vectors. As a result of a performance evaluation with the T2K Open Supercomputer (U. Tokyo) and the RX200S5, we obtain the performance with 0.86x and 0.95x speed-downs and 1/2 memory space compared to the conventional algorithm for a square process grid. We also show a new algorithm for small-sized matrices in massively parallel processing that takes an appropriately small value of p of the process grid p x q. In this case, the execution time of inverse transformation is negligible.

Keywords: parallel and distributed computing, numerical algorithms for CS&E

1 Introduction

A parallel dense symmetric eigensolver is a crucial tool for scientific computing. Some applications need few eigenvalues and eigenvectors for a sparse matrix. But other cases need all eigenvalues and all eigenvectors for a dense matrix. For example, all eigenvalues and eigenvectors for a dense matrix are needed in the density functional calculation of the electronic structure of an insulin hexamer [1]. Hundreds of computations of eigenvalues and eigenvectors are required to optimize the structure in some applications. Hence, we need to optimize the eigensolver to match both the dense and small sizes of the matrix in massively parallel processing because of the time restriction of computer services, such as at a supercomputer center.

In addition, current computer architectures are increasing in complexity. Therefore, we need to administrate deep hierarchical caches, non-uniform memory accesses, and increase of the number of cores. Due to the features of current computer architectures, a cache-aware algorithm, that is, a blocking algorithm, is used in many numerical libraries. As an example in eigensolvers, a blocking algorithm for the reduction of dense matrices was proposed by Dongarra *et al.* [2]. After that, to reduce the communication time, a two-step reduction algorithm was proposed by Bischof *et al.* [3].

These algorithms, however, are not aimed at small matrices. Instead, the target is a huge matrix to obtain high "computational" efficiency for one-time computation. This efficiency does not consider the actual execution limit of computer services. In current massively parallel machines, more than 10,000 cores are implemented. In this massively parallel environment, the conventional approach cannot work well because the actual matrix size that can be solved is very limited. For a dense eigensolver, the computation complexity grows to $O(n^3)$, and hence the computation time increases on the order of 100x of execution time with one core in weak scaling when we use 100,000 cores.

Katagiri *et al.* [4] proposed a massively parallel algorithm with a communication splitting multicast algorithm. The algorithm established more than a 5x speedup compared to that of ScaLAPACK [4], although the algorithm does not implement a blocking algorithm. The drawback of this algorithm was the restriction of process grid construction. In this paper, we propose an algorithm for a square process grid configuration. The goal of this paper is two-fold.

First, we propose a process grid free algorithm based on [4]. This enables us not only many opportunities to adapt our algorithm, but also another tuning approach.

Second, the process grid free algorithm has a tradeoff between memory space and execution speed. We evaluate the tradeoff by using two kinds of parallel machines. The execution speed with a small-sized matrix is examined in this paper. We focus on the small dimension of 10,000, which represents the real usage for a chemical simulation. The execution time for a small size of 10,000 is shown in the performance evaluation.

This paper is organized as follows. Section 2 explains sequential and parallel algorithms for symmetric eigensolvers. Section 3 proposes a new algorithm with communication-splitting multicasts. Section 4 is a performance evaluation with the T2K Open Supercomputer (U. Tokyo) and the RX200S5. Finally, we summarize our findings in Section 5.

2 Symmetric Dense Eigensolver

To calculate the symmetric standard eigenproblem $Ax = \lambda x$, where $A \in \Re^{n \times n}, \lambda \in \Re^1, x \in \Re^n$, we need to reduce the dense matrix A to a tridiagonal matrix T. This transformation is called "tridiagonalization." After solving the new eigenproblem for T, we obtain an eigenvalue and an eigenvector. The eigenvector, which is y in this example, is not the eigenvector for matrix A. Therefore, we need a transform from y to x, which is the eigenvector of A. This transformation is called "inverse transformation."

Non-blocking Sequential Algorithm

We describe the tridiagonalization processes shown in Figure 1.

In this figure, the notation $A^{(k)}{}_{a:b,c:d}$ indicates the sub-matrix of $A^{(k)}$, which consists of rows from a to b and columns from c to d. Figure 1 includes a dense vector matrix multiplication in line <3>, a dot product in line <4>, a copy in line <5>, and a

matrix update in line <6>. To perform inverse transformation, we need a workspace to store the Householder (pivot) vectors $u_1, u_2, \cdots, u_{n-2}$.

Inverse transformation is described in Figure 2. Figure 2 includes a dot product in line <3>, and a matrix update in line <4>.

<1> $do\ k = 1,\ n-2$
<2> $A^{(k)}{}_{k:n,k} \mapsto (\alpha_k, u_k)$, $where\ \alpha_k \in \Re^1, u_k \in \Re^n$.
<3> ${y_k}^T = \alpha_k {u_k}^T A^{(k)}{}_{k:n,k:n}$
<4> $\mu_k = \alpha_k {y_k}^T u_k$
<5> $x_k = y_k$
<6> $H_k A^{(k)} H_k = A^{(k)} - (x_k - \mu_k u_k){u_k}^T - u_k {y_k}^T$
<7> $enddo$

Figure 1. Tridiagonalization Processes.

<1> $do\ k = n-2,\ 1,\ -1$
<2> $do\ i = k,\ n$
<3> $\sigma_i = \alpha_k {u_k}^T A^{(k)}{}_{k:n,i}$
<4> $H_k A^{(k)}{}_{k:n,i} = A^{(k)}{}_{k:n,i} - \sigma_i u_k$
<5> $enddo;\ \ enddo;$

Figure 2. Inverse Transformation Processes.

3 The Communication Splitting Multicasting Algorithm

3.1 The Data Distribution

Let the number of MPI processes be *np* = *px* x *py*, and the process grid be *p* x *q*. The process identification is also defined in a 2D manner, that is, (*myidx*, *myidy*), which ranges from 0 to *p-1* for *myidx* and from 0 to *q-1* for *myidy*.

The symmetric dense matrix *A* is distributed to each process in a cyclic-cyclic manner with a non-compressed form; thus, it does not use symmetry. In the cyclic-cyclic distribution, indexes of the row and column for matrix *A* are distributed as follows.

$$\Pi = \{myidx\ \ + 1 + (i-1)\,p\},\ \Gamma = \{myidy\ \ + 1 + (j-1)q\}, \qquad \textbf{(1)}$$

where,

$$\begin{matrix} i = 1,2,\cdots, last\ (myidx\ +1+\lfloor n/p \rfloor p, \lfloor n/p \rfloor) \\ j = 1,2,\cdots, last\ (myidx\ +1+\lfloor n/q \rfloor q, \lfloor n/q \rfloor) \end{matrix}, \quad last\ (a,b) = \begin{cases} b+1\ (if\ a \le n) \\ b\ \ (if\ a > n) \end{cases}$$

By using the index sets in Equation (1), we can denote the distributed matrix and vectors. For example, $A^{(k)}_{\Pi,k}$ is a vector that consists of a cyclic distribution for the first dimension and an entire k-th column for matrix $A^{(k)}$, and $u_{k\Pi}$ is a vector in which the elements are distributed with the cyclic manner of vector u_k.

3.2 The Square Grid Algorithm for Tridiagonalization

For the parallel algorithm for a square grid [4], that is, the case of $p=q$, tridiagonalization is used, as described in Figure 3.

In Figure 3, all communications can be implemented by using multiple MPI_BCASTs or MPI_ALLREDUCEs on the splitted communicator of MPI. The communication time can be reduced in massively parallel execution, in contrast to the time needed by a conventional algorithm that cannot split the communication. To perform matrix updating in parallel in lines <25>-<29> in Figure 3, we need the partial elements of x_k and u_k. The copies can perform 2x the tridiagonal multicasting operations in lines <5>-<7> to obtain the elements from $u_{k\Pi}$ to $u_{k\Gamma}$, and in lines <12>-<14> to transpose y to x. This enables us to dramatically reduce the communication time for the operation in comparison to that of the conventional algorithm [2].

Figure 4 shows the data distribution of vectors u_k, x_k, and y_k in the square grid algorithm. The data distribution of their duplications is also shown.

3.3 The Process Grid Free Algorithm for Tridiagonalization

The Case of a Rectangle Grid ($p < q$)

In [4], there is no description of rectangle process grid algorithms for the rectangle grid ($p < q$). However, the algorithm can be constructed by exchanging MPI_BCAST in Figure 3 with MPI_ALLREDUCE. We consequently establish a multicasting algorithm, but the communication time increases compared to the case of the square process grid.

The key point of the change is the transpose operations in lines <5>-<7> and <12>-<14> in Figure 3. In the case of $p < q$, there is no data for the elements of y_k with a p cyclic distribution to go with the elements of x_k with a q cyclic distribution. To avoid this situation, we add a copy process of $u_{k\Gamma}=u_{k\Pi}$ stridden (*myidy / px*) with offset (*py / px*), before line <6> in Figure 3 for u_k operation. This is the key implementation technique of this distribution. Figure 5 shows the data distribution of y_k and its multicastings based on this implementation.

The Case of a Rectangle Grid ($p > q$)

There is also no description and no performance evaluation in [4] for the rectangle grid ($p > q$). According to our verification, this algorithm can be described by exchanging MPI_BCASTs with MPI_ALLREDUCE, as in the case of $p < q$. But the stride and offset are changed to (*myidx / py*) and (*px / py*), respectively. This is also a

key implementation technique to establish a grid free algorithm. Figure 6 shows the data distribution of y_k and its multicastings based on the above implementation.

The number of multicasting is reduced to the case of $p < q$, but the number of processes by MPI_ALLREDUCEs is increased. Hence, the best grid configuration depends on the communication performance of the tridiagonalization process.

<1> *do k=1, n-2*
<2> if (k $\in \Gamma$) MPI_BCAST ($A^{(k)}_{\Pi,k}$) to Cores sharing rows Π.
<3> else Receive data with MPI_BCAST ($A^{(k)}_{\Pi,k}$) endif;
<4> Computation of ($a_k, u_{k\Pi}$) with MPI_ALLREDUCE;
<5> if (I have diagonal elements of *A*)
<6> MPI_BCAST ($u_{k\Pi}$) to Cores sharing columns Γ.
<7> else Receive data with MPI_BCAST ($u_{k\Gamma}$) endif;
<8> *do j=k, n*
<9> if (j $\in \Gamma$) $y_{k\Pi}{}^T = y_{k\Pi}{}^T + \alpha_k u_k{}^T{}_j A^{(k)}_{\Pi,j}$
<10> *enddo*
<11> MPI_ALLREDUCE of $y_{k\Pi}{}^T$ to Cores sharing rows Π.
<12> if (I have diagonal elements of *A*)
<13> MPI_BCAST ($y_{k\Pi}{}^T$) to Cores sharing columns Γ.
<14> else Receive data with MPI_BCAST ($x_{k\Gamma}$) endif;
<15> *do j=k, n*
<16> $\mu_k = \alpha_k y^T_{k\Pi} u_{k\Pi}$ *enddo*
<17> MPI_ALLREDUCE of μ_k to Cores sharing rows Π.
<25> *do j=k, n*
<26> *do i=k, n*
<27> if (i $\in \Pi$ .and. j $\in \Gamma$) then
<28> $A^{(k+1)}_{i,j} = A^{(k+1)}_{i,j} - u_{k_i}(x_k{}^T{}_j - \mu u_k{}^T{}_j) - u_{k_i} y_k{}^T{}_j$ endif;
<29> *enddo; enddo;*
<30> if (k $\in \Gamma$) $\Gamma = \Gamma - \{k\}$ endif
<31> if (k $\in \Pi$) $\Pi = \Pi - \{k\}$ endif
<32> *enddo*

Figure 3. Parallel Tridiagonalization Algorithm with Square Process Grid Proposed in [4].

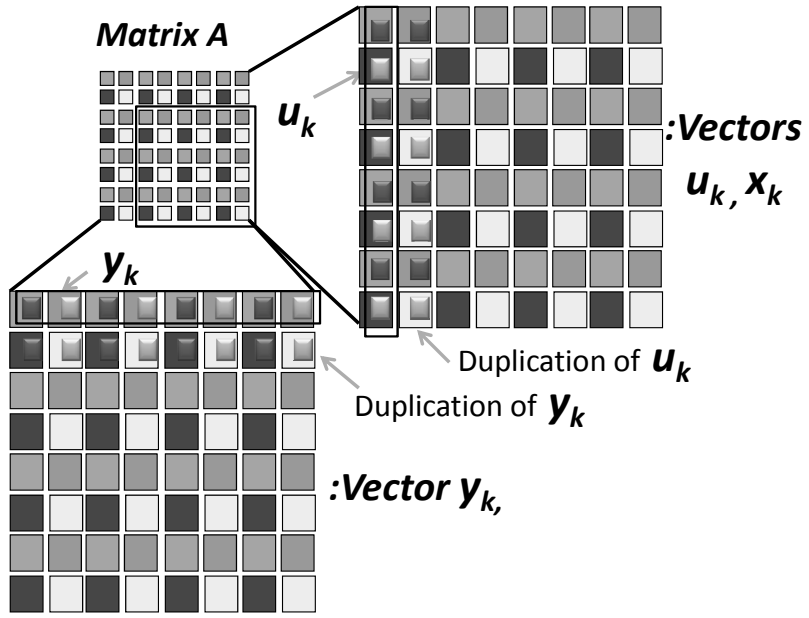


Figure 4. Data Distribution of Vectors u_k, x_k, and y_k , and Vectors of Their Duplications.

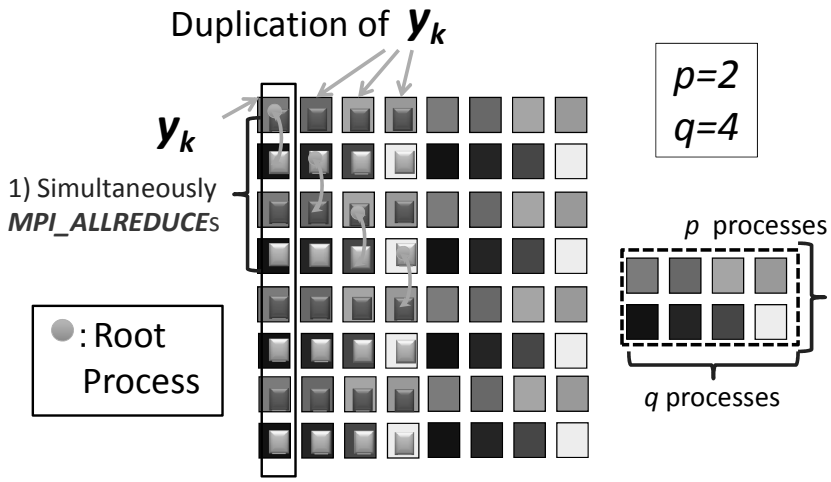


Figure 5. Data Distribution of Vector y_k and Its Multicastings on the Process Grid Free Algorithm for Tridiagonalization. (the Case of Rectangle Grid ($p < q$), $p = 2$ and $q = 4$).

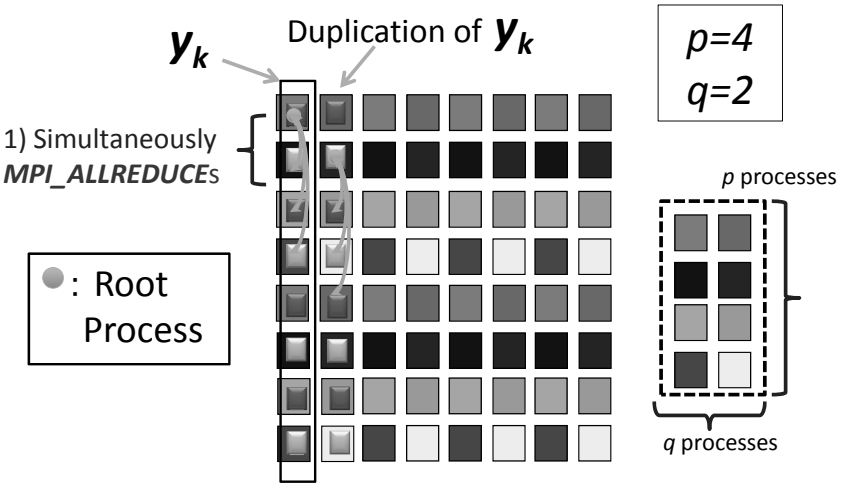


Figure 6. Data Distribution of Vector y_k and Its Multicastings on the Process Grid Free Algorithm for Tridiagonalization. (the Case of Rectangle Grid ($p > q$), $p = 4$ and $q = 2$).

3.4 The Process Grid Free Algorithm for Inverse Transformation

Figure 7 shows the parallel algorithm for inverse transformation.

```
<1> do k=n-2, 1, -1
<2>     Gather the vector  u_k  and  scalar  α_k  by using
          p-times of MPI_BCAST for  u_kΠ  with sharing . columns  Γ .
<3>     do i=kstart, kend
<4>         σ_i = α_k u_k^T A^(k)_k:n,i
<5>         A^(k)_k:n,i = A^(k)_k:n,i − σ_i u_k_i
<6>     enddo
<7> enddo
```

Figure 7. The Parallel Inverse Transformation Algorithm with Square Grid Proposed in [4].

The algorithm in Figure 7 also can be described with multiple MPI_BCASTs. The process grid affects the execution performance since p is the number of MPI_BCASTs for the p x q grid. The small p seems to perform better, but it depends on network performance. The data distribution of u_k and its multicastings are shown in Figure 8.

However, the memory requirement to store the Householder vector u_k varies according to p. If $p=2$, it needs 2x memory space compared to $p=4$ to keep the

Householder (pivot) vectors. Therefore, this algorithm is a tradeoff between execution time and memory space. This algorithm is process grid free. The process grid of the inverse transformation is the same as that of tridiagonalization. From this point of view, the entire performance is determined by the communication performance between tridiagonalization and inverse transformation.

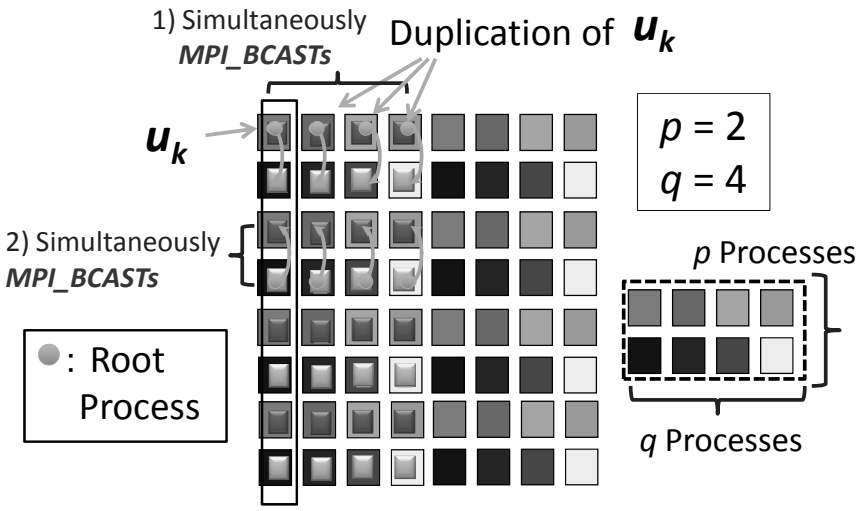


Figure 8. Data Distribution of u_k and Its Multicastings on the Process Grid Free Algorithm for Inverse Transformation.

4 Performance Evaluation

4.1 Machine Environment

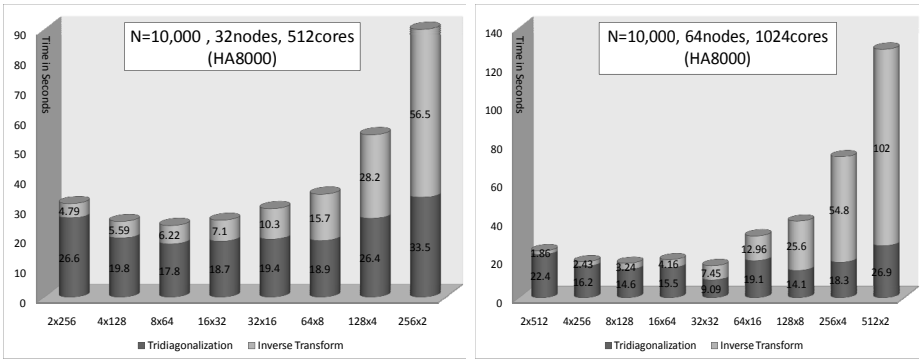
We used the T2K Open Supercomputer (TODAI), which is a HITACHI HA8000 installed at the Information Technology Center, The University of Tokyo. Each node contains 4 sockets of the AMD Opteron 8356 (Quad core, 2.3 GHz). The L1 cache is 64 KB/core, the L2 cache is 512 KB/core, and the L3 cache is 2 MB/4 cores. The memory on each node is 32 GB with DDR2-667 MHz. The theoretical peak is 147.2 GFLOPS/node. Inter-node connection is 4 lines of the Myri-10G with a full bisection connection. The inter-node connection attains 5 GB/sec in both directions. We used the HITACHI Fortran90 Compiler version V01-00-/B with option "-opt=ss -noparallel." Users can use a maximum of 64 nodes (1,024 cores) for a personal application in normal service, but a maximum of 256 nodes (4,096 cores) is available for a special service, which can be performed once per month.

We also used the RICC PRIMERGY RX200S5 installed in the Advanced Center for Computing and Communication, RIKEN. Each node contains 2 sockets of the Intel Xeon X5570 (Quad core, 2.93 GHz). The L1 cache is 256 KB/core, the L2 cache is 1 MB/core, and the L3 is 8 MB/4 cores. The memory on the node is 12 GB with DDR3-1333 MHz. The theoretical peak is 93.0 GFLOPS/node. Inter-node connection is one line of the DDR InfinitiBand. We used the Fujitsu Fortran90 Compiler version 3.2 with the option "-pc –high." In this experiment, 32 nodes (256 cores) were used.

We used ABCLib_DRSSED version 1.04 [5][6]. No automatic tuning was used in this experiment; hence, the default parameters were set.

4.2 Performance on Different Process Grids

Figure 9 shows the execution time. Table 1 shows the speedups and memory spaces in the cases of the square and rectangle grids on the T2K.



(a) 512 Cores (p != q) (b) 1024 Cores (p = q)

Figure 9. Execution Time on Different Process Grids on the T2K.

In Figure 9, the execution time of $p > q$ increases because the gathering time for u_k increases according to p. In Table 1, speedup, memory space, speedup per memory (SPM) based on conventional execution are calculated. The conventional executions are 16x32 (512 cores) and 32x32 (1024 cores). If SPM is more than 1.0, it performs with good efficiency with respect to the ratio of speedup based on unit memory space.

Table 1 shows that the case of $p>q$ has high efficiency with respect to SPM. Especially, the speed-down is only 0.86x, but memory space is reduced to 1/2 in Table 1 (a). In the $p = q$ case, the algorithm of 32x32 is very fast compared to the others. If users accept the 0.41x speed-down, the memory space can be reduced to 1/4.

Table 1. Speedups and Memory Spaces on the T2K. The Memory Space Is Calculated by the Memory Requirement to Keep Householder Vectors u_k.

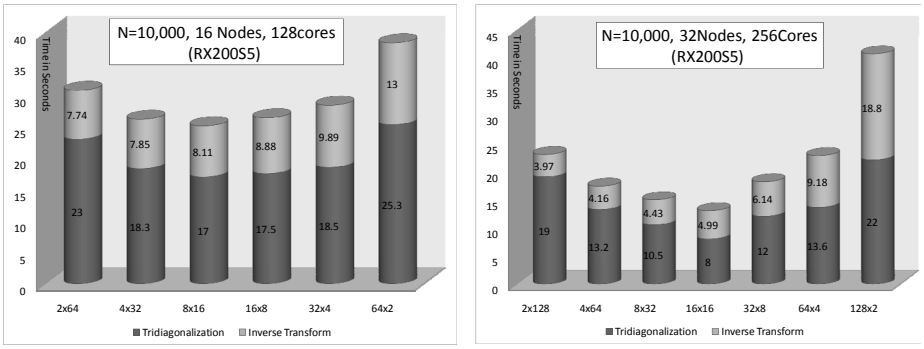
(a) 512 Cores (p != q)

Grid (pxq)	Time [sec.]	Speed UP	Mem.	SPM
16x32	25.8	1x	1x	1
8x64	24.0	1.07x	2x	0.5
4x128	25.3	1.01x	4x	0.2
2x256	31.3	0.82x	8x	0.1
32x16	29.7	0.86x	0.5x	1.7
64x8	24.6	0.74x	0.25x	2.9
128x4	54.6	0.47x	0.125x	3.7
256x2	90.0	0.28x	0.062x	4.5

(b) 1024 Cores (p = q)

Grid (pxq)	Time [sec.]	Speed UP	Mem.	SPM
32x32	16.5	1x	1x	1
64x16	32.0	0.51x	0.5x	1.02
128x8	39.7	0.41x	0.25x	1.6
256x4	73.1	0.22x	0.125x	1.7
512x2	128	0.12x	0.062x	1.9

Figure 10 shows the execution time. Table 2 shows the speedups and memory spaces in the cases of the square and rectangle grids on the RX200S5.



(a) 128 Cores (p != q)

(b) 256 Cores (p=q)

Figure 10. Execution Time in Different Processor Grids on the RX200S5.

For Table 2 (a), (b), the ratios of SPM are better than those of the T2K. The speed-down is only 0.95x, but the memory space is reduced to 1/2 in Table 2 (a). In Table 2 (a), the speed-down is only 0.71x with 32x8 compared to the case of 16x16.

Table 2. Speedups and Memory Spaces on the RX200S5. The Memory Space Is Calculated by the Memory Requirement for Householder Vectors u_k.

(a) 128 Cores (p != q)

Grid (pxq)	Time [sec.]	Speed UP	Mem.	SPM
8x16	25.1	1x	1x	1
4x32	26.1	0.96x	2x	0.48
2x64	30.7	0.81x	4x	0.20
16x8	26.3	0.95x	0.5x	1.9
32x4	28.3	0.88x	0.25x	3.5
64x2	38.3	0.65x	0.125x	5.2

(b) 1024 Cores (p = q)

Grid (pxq)	Time [sec.]	Speed UP	Mem.	SPM
16x16	12.9	1x	1x	1
32x8	18.1	0.71x	0.5x	1.4
64x4	22.7	0.56x	0.25x	2.2
128x2	40.8	0.31x	0.125x	2.4

4.3 Execution Performance in a Massively Parallel Environment

Figure 11 shows the execution time with 4,096 cores (256 nodes) on the T2K. In this experiment, 11 kinds of processor-grid configurations were tested.

Figure 11 indicates following interesting phenomena:

1. The conventional square grid, 64x64 (total execution time is 11.7+22.4=34.1 seconds), is not the fastest. In this case, 8x512 (the total execution time is 2.05+26.8=28.8 seconds) is the fastest.
2. The ratio between the time of inverse transformation and the total time decreases when p is reduced. The fastest case of inverse transformation takes only 0.283 seconds of execution in the process grid 2x2048. The ratio of execution time of inverse transformation to the total time is 0.283/59.7=0.47%. This is negligible time.
3. When p increases, the execution time of inverse transformation greatly increases. In addition, it causes a bottleneck due to the increase of processes according to multicastings. In contrast, the time of tridiagonalization only slightly affects the total time compared to the time of inverse transformation. This is due to good load balancing for the heavy computational part of tridiagonalization.

Phenomena 2 and 3 provide another possibility of optimization for symmetric eigensolvers. If we use much memory to store pivot vectors, we take small values of p. Because the heavy computational part is only tridiagonalization in this case, we can tune the routine in a simple manner. Conventional tuning is very complex since there is a tradeoff between tridiagonalization and inverse transformation in the communication, especially in a conventional square process grid. Again, our target is small-sized matrices. There is a room for memory space in our target. Hence, the algorithm with small values of p is a candidate for an efficient parallel algorithm to be considered in the future.

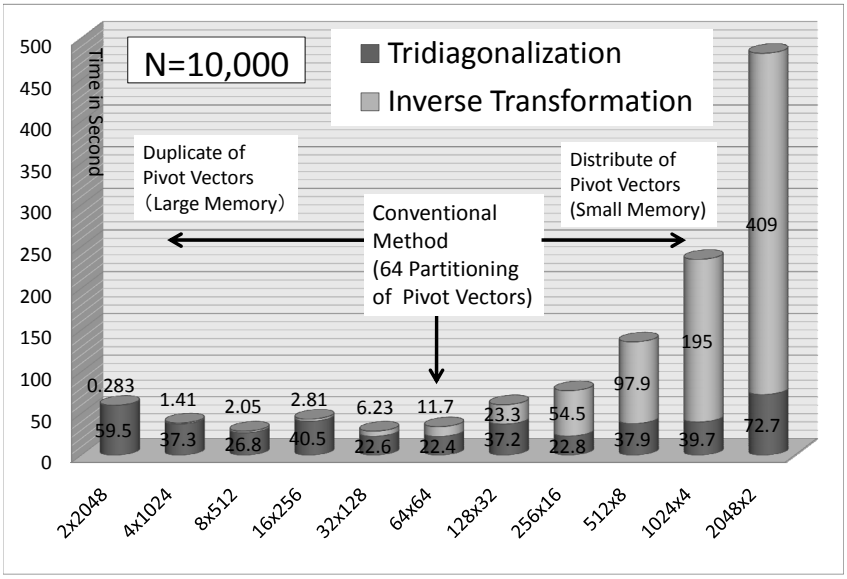


Figure 11. Execution Time with 4,096 Cores (256 Nodes) on the T2K Using Different Processor Grids.

5 Conclusion

In this paper, we propose a process grid free algorithm for a massively parallel dense symmetric eigensolver with a communication splitting multicasting algorithm. A tradeoff exists between speed and memory space in this algorithm. As a result of the performance evaluation with the T2K Open Supercomputer (HITACHI HA8000) and RICC RX200S5, we found that 0.86x and 0.95x speed-downs with 1/2 memory space allows us to keep the Householder vectors.

We showed the possibility of this new algorithm for small-sized matrices on massively parallel processing to take appropriately small values of p of process grid p x q. In this case, the execution time of inverse transformation is negligible.

The blocking parallel algorithm is now being studied in [7] and takes into account the communication reduction for the symmetric dense eigensolver. Implementing a communication-hiding algorithm with previous sending for the next-step Householder vector is important future work for small-sized matrices on massively parallel processing.

Acknowledgments We thank the RIKEN Cluster of Clusters (RICC) at RIKEN for the computer resources used for the experiment. This work is partially supported by Grant-in-Aid for Scientific Research (B) "Development of the Framework to Support Large-scale Numerical Simulation on Multi-platform," No. 21300017, and Grant-in-

Aid for Scientific Research (B) “Development of Auto-tuning Specification Language Towards Manycore and Massively Parallel Processing Era,” No. 21300007.

References

1. Inaba, T., Tsunekawa, N., Hirano, T., Yoshihiro,T., Kashiwagi, H. and Sato, F.: Density Functional Calculation of the Electronic Structure on Insulin Hexamer, Chemical Physics Letters, Vol.434, Issues 4-6, pp. 331-335 (2007).
2. Dongarra, J. J., Hammarling, S. J. Sorensen, D. C.: Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations, Journal of Computational and Applied Mathematics, Vol. 27, pp. 215–227 (1989).
3. Bischof C.H., Marques, M. and Sun, X.: Parallel Bandreduction and Tridiagonalization, Proceedings of Sixth SIAM Conference on Parallel Processing for Scientific Computing, pp. 22-24 (1993)
4. Katagiri, T., Kanada, Y. : An Efficient Implementation of Parallel Eigenvalue Computation for Massively Parallel Processing, Parallel Computing, vol.27, no.14, pp.1831-1845 (2001)
5. Katagiri, T., Kise, K., Honda, H., Yuba, T.: ABCLib_DRSSED: A Parallel Eigensolver with an Auto-tuning Facility, Parallel Computing, Vol.32, Issue 3, pp.231-250 (2006)
6. ABCLib_DRSSED home page: http://www.abc-lib.org/main1.html
7. Imamura, T.,: How To Develop The Eigenvalue Solver Which Organizes Beyond Hundred Thousand Cores, IPSJ SIG Notes, Vol.2009-HPC-121, No.19 (2009) In Japanese.