Progress in NUCLEAR SCIENCE and TECHNOLOGY, Vol. 2, pp.643-650 (2011)

ARTICLE

Development of a High-Performance Eigensolver on a Peta-Scale Next-Generation Supercomputer System

Toshiyuki IMAMURA[1,3,*], Susumu YAMADA[2,3] and Masahiko MACHIDA[2,3]

[1] *The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan*
[2] *CCSE, Japan Atomic Energy Agency, 6-9-3 Higashi-Ueno, Taitoh-ku, Tokyo 110-0015, Japan*
[3] *CREST, Japan Science Technology Agency*

For current supercomputer systems, multicore and multisocket processors are required in order to build a system, and choice of interconnection is essential. In addition, for effective development of new code, high-performance, scalable, and reliable numerical software is key. ScaLAPACK and PETSc are software developed for distributed memory parallel computer systems. Real computation requires software that is highly tuned for implementation on new architectures, such as many-core processors.

In the present study, we introduce a high-performance, highly scalable eigenvalue solver with the goal of realizing the K-computer system, which is a next-generation supercomputer system. We have developed two versions of this eigenvalue solver, namely, the standard version (eigen_s) and an enhanced-performance version (eigen_sx), both of which were developed on the T2K cluster system housed at the University of Tokyo. Eigen_s uses conventional algorithms, such as Householder tridiagonalization, the divide and conquer (DC) algorithm, and the Householder back-transformation. These algorithms are carefully implemented using a blocking technique and flexible two-dimensional data-distribution in order to reduce the overhead of memory traffic and data transfer, respectively. Eigen_s performs excellently on the T2K system with 4,096 cores (theoretical peak: 37.6 TFLOPS) and exhibits fine performance (3.0 TFLOPS) with a 200,000-dimensional matrix. The enhanced version, eigen_sx, uses more advanced algorithms, such as the narrow-band reduction algorithm, DC for band matrices, and the block Householder back-transformation with WY- representation. Even though this version is still in the test stage, eigen_sxhas realized 4.7 TFLOPS with a 200,000-dimensional matrix.

KEYWORDS: eigenvalue solver, high-performance computing, scalable performance, T2K supercomputer system, K-computer system

I. Introduction

In 2008, Roadrunner reached PFLOPS, and we achieved peta-scale computing power.[1)] However, for a number of reasons, high-performance computing is difficult to achieve.

Peta-scale systems usually have more than 100,000 computing cores and an interconnected multi-socket, multi-core architecture. Each computing core is a general-purpose processor-unit and provides a computational power on the order of GFLOPS. However, higher performance, especially performance exceeding 100 TFLOPS, is only available when most of the computational cores work perfectly and without parallel overhead. In other words, significantly higher parallelism is required for peta-scale algorithms. Otherwise, high performance cannot be guaranteed at the design level. Another drawback to achieving high performance with a multi-core processor system is a shortage of memory bandwidth. Since the improvement in memory bandwidth is extremely gradual in semiconductor technology, this results in the absolute bandwidth per core being reduced, even though the memory bandwidth has improved from one-third to half of the frequency of the processor. In fact, the memory bandwidth for the processor installed in recent supercomputers is smaller than the frequency or FLOPS rate of the processor. On a T2K super cluster system, which is used in the present study, quad-socket AMD Opteron Barcelona quad-cores are installed, and each processor core operates at 2.3 GHz and can perform at a rate of 9.2 GFLOPS. Theoretically, the memory bandwidth of a single node is 42.7 GB/s, which is equivalent to 0.29 Byte/Flop[a] (=42.7 [GB/s]/(9.2 [GFLOPS]*4*4)). With respect to the Fujitsu SPARC64 VIIIfx processor[2)] adopted for the K-computer, a next-generation supercomputer, the memory bandwidth is 0.5 Byte/Flop (=64 [GB/s]/128 [GFLOPS]), which is slightly higher than that on the T2K system. However, the memory bandwidth cannot reach the Byte/Flop rate of a vector processor (for example, 2.5 Byte/Flop on an NEC SX-9). This tendency will become more pronounced in the future. Therefore, lower memory usage and reduced memory communication/traffic should be considered in implementing application codes.

We should consider two factors in the present study,

*Corresponding author, E-mail: imamura@im.uec.ac.jp

[a]Byte/Flop is the unit for transferring data between the main memory and the processor per single floating operation

© 2011 Atomic Energy Society of Japan, All Rights Reserved.

namely, higher parallelism and an algorithm that has a low memory traffic property. For the eigenvalue solver for dense symmetric matrices, the memory bottleneck is the Householder transformation step, in which a full dense matrix is condensed into a compact format, a tridiagonal matrix. In the present study, we developed two eigenvalue solvers. One solver adopts the conventional approach, which has the problem of a bottleneck in memory access, and the other introduces a very efficient algorithm, namely, narrow band reduction, to refine the bottleneck of a narrower memory bandwidth.

The remainder of the present paper is organized as follows. In Section II, we present two eigenvalue solvers, namely, eigen_s and eigen_sx. In Section III, performance tests on a large-scale cluster system are described. In Section IV, conclusions are presented and future research is discussed.

II. Algorithm for Solving an Eigenvalue Problem with a Dense Symmetric Matrix

In this section, we describe the standard eigenvalue algorithm for a real dense-symmetric matrix to compute all eigenvalues and corresponding eigenvectors, as defined by Eq. (1). In other words, we focus only on the full-diagonalization algorithm in numerical simulations.

$$Ax = \lambda x, A = A^T \in \mathrm{R}^{n\times n} \quad (1)$$

1. Standard Approach: Eigen_s

The standard approach consists of three steps, which are described briefly in this section.

(1) Householder Tridiagonalization

First, we define the reflector function H, as follows:

$$(u, \beta) := H(a, k) \quad (2)$$

$$u_{[1:k]} = a_{[1:k]} + \mathrm{sign}(\|a_{[1:k]}\|, a_k)e_k \quad (3)$$

$$u_{[k+1:n]} = 0,\ \beta = 2/\|u\|^2 \quad (4)$$

where u and a are vectors of the same dimension, β returns the scalar value, and k indicates the index of reflector operation. Using this reflector function, we first compute the reflector vector u, and its factor β. Next, we construct a Householder's reflector by $I - \beta uu^T$, and we can then transform matrix A as follows:

$$A = \left[\begin{array}{c|c} A_0 & a \\ \hline a^T & \alpha \end{array}\right] \quad (5)$$

$$A \to \left[\begin{array}{c|c} (I - \beta uu^T)A_0(I - \beta uu^T) & \pm\|a\|e_{N-1} \\ \hline \pm\|a\|e_{N-1}^T & \alpha \end{array}\right] \quad (6)$$

This transformation can be applied recursively to $A_1 = (I - \beta uu^T)A_0(I - \beta uu^T)$. Finally, we obtain a tridiagonal matrix transformed from the dense matrix A. This transformation is referred to as Householder tridiagonalization.

In order to improve the performance, incorporating data blocking into the algorithm is a practical solution. Hammarling et al.[3] developed a block algorithm using a panel-partitioned data distribution. Using their blocking scheme, one of the major computational parts of the Householder transformation can be written by a pair of matrix-matrix multiplications such as $A - UV^T - VU^T$. The Householder transformation was originally written in terms of vector-vector outer products. Since the Householder transformation was accelerated dramatically on a parallel computer system, we herein adopt this blocking strategy in the proposed solvers.

(2) Divide and Conquer Method

The second step of the standard approach is to compute the eigenpairs for the tridiagonal matrix transformed in the first step. Here, we adopt the divide-and-conquer algorithm developed by Cuppen.[4] This algorithm exhibits very natural parallelism in the internal processes and can also be applied to subproblems recursively, as shown in **Fig. 1**. Thus, this algorithm provides perfect parallel performance. This step of both the standard approach and enhanced approach is presented elsewhere.[5]

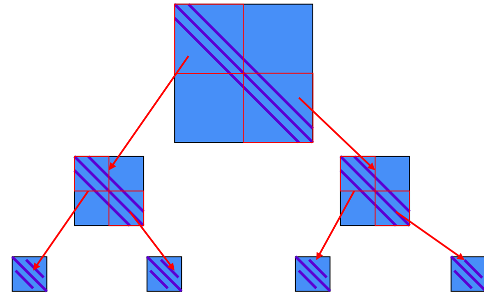

Fig. 1 Schematic diagram of the divide and conquer algorithm

(3) Block-Householder Back-Transformation

The final step is back-transformation of the computed eigenvectors, because the eigenvectors computed in the second step correspond to the transformed tridiagonal matrix. Back-transformation is the completely opposite one-sided procedure of the Householder transformation. Thus, back-transformation involves reverse-order matrix-multiplication of the Householder transformation matrices, such as $(I - \beta_n u_n u_n^T)\cdots(I - \beta_2 u_2 u_2^T)(I - \beta_1 u_1 u_1^T)$. The eigenvectors are multiplied by this matrix from the left-hand side.

As described in the forward Householder transformation, data blocking improves the performance. Back-transformation is also accelerated by data blocking. Two consecutive operations of the Householder transformation can be written as follows:

$$(I - \beta_1 u_1 u_1^T)(I - \beta_2 u_2 u_2^T) = I - UCU^T \quad (7)$$

where $U = [u_1, u_2]$. This blocking is referred to as compact WY-blocking representation. Generally, matrix C in this blocking scheme $(I - \beta_1 u_1 u_1^T)(I - \beta_2 u_2 u_2^T)\cdots(I - \beta_k u_k u_k^T) = I - UCU^T$ is recursively computed as follows:

$$C_j := \left[\begin{array}{cc} C_{j-1} & 0 \\ -\beta_{j-1} u_j^T U_{1:j-1} C_{j-1} & \beta_j \end{array}\right] \quad (8)$$

Matrix C can be also represented by the following implicit scheme:

$$C = \left(\mathrm{diag}^{-1}(\beta_1, \cdots, \beta_k) + \mathrm{Lower}(U^T U)\right)^{-1} \quad (9)$$

Since the above equation is represented as the inverse of a triangular matrix, we need no explicit data for C. When the blocking factor k is small, the cost of Eqs. (8) and (9) are similar. However, as k increases, Eq. (9) has a smaller cost (almost half) than Eq. (8). Thus, we can choose both equations according to blocking factor k.

2. Advanced and Enhanced Approach: Eigen_sx

In the standard approach, the performance bottleneck was determined to be hidden in the Householder transformation, specifically, in matrix-vector multiplication.[6)] We also pointed out that modern microprocessors have a low Byte/Flop rate (used herein to transfer data from main memory to the processor), which causes a degradation in performance. Furthermore, this suggests that matrix-vector multiplication does not exhibit good scalability unless the memory bandwidth is improved at least twofold.

(1) Narrow-Band Reduction

In order to improve the performance in the first step, Householder tridiagonalization, we proposed another approach by which to reduce a dense matrix to a banded matrix. This algorithm is referred to as narrow-band reduction. **Figure 2** shows the full algorithm of narrow-band reduction. In Step (ii), matrices U and C, which are necessary in order to construct a block Householder's reflector, are defined by Householder QR factorization using panel W. Specifically, the outputs U and C are computed as follows:

$$(U, C) := H_{\mathrm{block}(K)}(W) \tag{10}$$

$$(u_i, \beta_i) := H(W_{:,i}, N - i + 1) \tag{11}$$

$$W_{:,i+1:K} := (I - \beta_i u_i u_i^T) W_{:,i+1:K} \tag{12}$$

$$U = [U, u_i] \tag{13}$$

where H and C are computed by Eqs. (2) and (8), respectively. The above scheme is applied recursively, for $i = 1, 2..., K$. From Steps (iii) through (ix) in Fig. 2, the block algorithm for two-sided transformation is constructed as follows using Hammarling's panel-partitioned data distribution, in the same manner as the standard Householder tridiagonalization.

$$(I - UCU^T)A(I - UC^T U^T) = A - UV^T - VU^T \tag{14}$$

$$V = (AU)C^T - \frac{1}{2}U(CU^T(AU)C^T) \tag{15}$$

where W is a preserved variable for panel-partitioning. In addition, considering the symmetry of A, except for symmetric matrix S, Step (v) uses an upper triangular matrix S'.

In Fig. 2, each capital letter denotes a matrix. In many cases, matrices are square matrices. However, U and V are tall-skinny matrices in this case. All of the operations are written in matrix-matrix multiplication form. As described in the previous subsection, one of the major parts of the Householder transformation is $A - UV^T - VU^T$, which is referred to as a rank k update. Another major part is the matrix-vector product is $(I - \beta uu^T)A_0(I - \beta uu^T)$. Substituting matrix-vector operations with matrix-matrix operations improves the performance, because matrix-matrix multiplication, referred to as DGEMM in the BLAS library, performs excellently (often yielding 70-80% of the theoretical peak performance). If a fast DGEMM routine tuned for a specific processor can be used in the implementation, a significant performance improvement can be expected. Since the performance of a matrix-vector product is bounded by memory bandwidth, this improvement in performance must be large on lower Byte/Flop rate processors, such as T2K and the K-computer (0.29 and 0.5, respectively). In fact, in a previous study,[6)] we found that the acceleration can be increased by two to three times using this algorithm.

for $j = N, \ldots, 1$ **step** $-M$; $i = j - M$
  (i) $U \leftarrow \emptyset, V \leftarrow \emptyset, W \leftarrow A_{(*,i+1:j)}$
  for $k = 0, \ldots, M-1$ **step** K
    (ii) Construct a Householder block reflector
      $(C, U^{(k)}) := H_{\mathrm{Block}(K)}(W_{[*,j-k]})$
    (iii) Matrix-Vectors multiplication
      $\bar{V}^{(k)} \leftarrow A_{[1:j-k-1,1:j-k-1]}U^{(k)}$
    (iv) $\bar{\bar{V}}^{(k)} \leftarrow \bar{V}^{(k)} - (UV^T + VU^T)U^{(k)}$
    (v) $V^{(k)} \leftarrow \bar{\bar{V}}^{(k)}C^T - U^{(k)}S'$,
      $S = \frac{1}{2}CU^{(k)T}\bar{\bar{V}}^{(k)}C^T$,
      $S' = (\mathrm{Upper}(2S) + \mathrm{diag}(S))$
    (vi) $U \leftarrow [U, U^{(k)}], V \leftarrow [V, V^{(k)}]$
    (vii) $W_{[*,j-k:j]} \leftarrow W_{[*,j-k:j]} - (U^{(k)}V^{(k)T} + V^{(k)}U^{(k)T})_{[*,j-k:j]}$
  endfor
  (viii) $A_{[*,j-M+1:j]} \leftarrow W$
  (ix) $2M$ rank-update
    $A_{[1:i,1:i]} \leftarrow A_{[1:i,1:i]} - (UV^T + VU^T)_{[1:i,1:i]}$
endfor

Fig. 2 Narrow-band reduction algorithm

(2) Divide and Conquer Method for a Banded Matrix

Narrow-band reduction does not return a tridiagonal matrix, but rather a banded matrix. We also need an alternative algorithm for the second step.

In the second step, we introduce the enhanced algorithm of the divide and conquer method, which is also reported by Pham et al.[5)] This algorithm is a natural enhancement of the original Cuppen's algorithm and can be easily implemented by very reliable libraries, for example, LAPACK and ScaLAPACK. The outline of the algorithm is shown in **Fig. 3**, and numerical aspects and performance test are reported in a previous study .[5)]

(3) Block-Householder Back-Transformation

The final step in the advanced solver is also back-transformation. This is similar to the standard approach except that the length of reflector vector is shorter in one element. Most of the algorithm for the block-Householder back-transformation was discussed in the previous subsection.

III. Performance Test

1. Parallel Implementation

(1) Parallel Programming and Parallelization Model

Since the proposed solvers, eigen_s and eigen_sx, are to be used on the K-computer next-generation supercomputer

function $[\Lambda, X]$=BAND_DC(B)
  if(dim(B) is small)
    Compute $[\Lambda, X] = \text{eigen}(B)$ and **return**
  endif
  Decompose B into B_1 and B_2 with K-rank perturbations by using SVD.
    $B = \text{diag}(B_1, B_2) + UU^T, U \in \mathrm{R}^{N\times K}$
  $[\Lambda_1, X_1] = \text{BAND_DC}(B_1)$
  $[\Lambda_2, X_2] = \text{BAND_DC}(B_2)$
  Let $\Lambda^{(0)} = \text{diag}(\Lambda_1, \Lambda_2)$ and $X^{(0)} = \text{diag}(X_1, X_2)$.
  for $j = 1, \ldots, K$
    Let $B^{(j)} = \Lambda^{(j-1)} + v_j v_j^T, v_j = {X^{(j-1)}}^T u_j$.
    Solve eigenvalue problem for $B^{(j)}$

$$f(\lambda) = 1 + \sum_i \frac{(v_{j[i]})^2}{\lambda_i - \lambda} = 0$$

$$x = X^{(j-1)}\text{Normal}((\Lambda^{(j-1)} - \lambda I)^{-1} v_j),$$

    then set $\Lambda^{(j)} = \text{diag}(\lambda_1, \ldots, \lambda_N)$,
    and $X^{(j)} = [x_1, \ldots, x_N]$.
  endfor
  Set $[\Lambda, X] = [\Lambda^{(K)}, X^{(K)}]$ and **return**.
end

Fig. 3 Divide and Conquer method for a banded matrix

system, these solvers are developed with three types of parallelism.

1. MPI: message passing parallelism (for interconnected distributed memory)
2. OpenMP: thread parallelism (for shared memory)
3. SIMD: instruction level parallelism

We elaborately do MPI and OpenMP hybrid parallel programming, and the proposed solvers run in thread parallel mode, message passing parallel mode, or combined mode. We can specify the running mode for runtime options (ex. `export OMP_NUM_THREADS=4; mpirun -np 3 ./a.out`). In addition, we thoroughly tune the codes in SIMD fashion and use highly tuned BLAS libraries. Therefore, most of the internal loop structures are expected to perform at theoretical peak performance. In particular, the blocking algorithm consists of several matrix-matrix multiplications, which can run also rather quickly.

(2) Data Distribution, Data Layout, and Process Mapping

We adopt only one matrix data distribution, namely, the two-dimensional cyclic distribution depicted in **Fig. 4**. Processor mapping on two-dimensional Cartesian coordinates is very flexible, and users can specify any pattern to be used during execution. For example, when NP = 24, {(1,24), (2,12), (3,8), ..., (24,1) } patterns are available (square or near-square mapping is selected by default). In particular, the K-computer adopts the Tofu interconnect,[8)] which is a 6-D torus network. This flexible feature is preferable for a brand-new network.

(3) Core Implementation and 2D-Communication

Basically, all of the matrices and vectors are distributed in a two-dimensional distribution. Column vectors and row vectors are distributed over the subgroup at the same row and column, respectively. When the specific vector on a matrix, for example, `A(1:6,6)` in Fig. 4, is required, the data is replicated with simultaneously broadcasting over the subgroups (:,column) or (row,:), namely, multicasting. Collective communication is designed over column or row groups in the two-dimensional distribution. Collective communication is quite scalable when the number of computational nodes increases.

In order to compute the inner product or norm, after partial and local summation, collective communication (`MPI_Allreduce` with `MPI_SUM` option) is called. Matrix-vector multiplication and rank-k update, that is, Steps (iii) and (ix) in Fig. 2, require the row vectors transposed from the column vectors. Thus, the Householder tridiagonalization and the narrow-band reduction routines are designed to have two duplicated vectors for U and V.

(4) Miscellaneous Information

The environment for development and experiments in this work is shown in **Table 1**. We use the T2K super cluster system housed at University of Tokyo, which has quad-core AMD Barcelona processors, largest computational partition is 512 nodes (8,192 cores). In this study, 256 nodes (4,096 cores) are allowed to use, in which theoretical peak performance reaches 37.6 TFLOPS. As presented, our solver is developed in a hybrid parallel programming style. Thus, quite flexible execution or assignment onto computational cores is available. With the consideration of the conflict on memory bus among cores (in other words, in order to avoid unnecessary traffic on hyper transport bridges), we assign each computational process onto a single socket, statically.

2. Parallel Performance for Large-Scale Problems

(1) Performance and Scalability

Varying the dimensions of the test matrix (We use the Frank matrix which is a well-known benchmark, the eigenvalues and corresponding eigenvectors of which can be calculated analytically.) and the number of nodes, the elapsed time of our two solvers is measured on the T2K super cluster system. **Figures 5** and **6** show the results on a logarithm scale. The proposed solvers are scalable when the dimension of the matrix is large, e.g., 20,000. Although the graphs in these figures appear quite similar, there are differences. Both solvers have similar parallel scalability. However, the solvers do not exhibit good scalability in the case of smaller problems. Since parallel overhead cannot be concealed behind the computational cost when the problem size is smaller, we can find the trade-off point. In addition, eigen_sx is slightly faster than eigen_s when the dimension of the matrix is large.

Figure 7 shows the performance scalability of eigen_s. Even though the dimension of the test matrix is relatively small, acceptable performance scalability is achieved for the case in which N = 38,400. The performance degrades approximately 50% from the ideal performance. In fact, the ideal performance reaches approximately 20% of the theoretical peak performance, and the actual performance remains at 10% of the

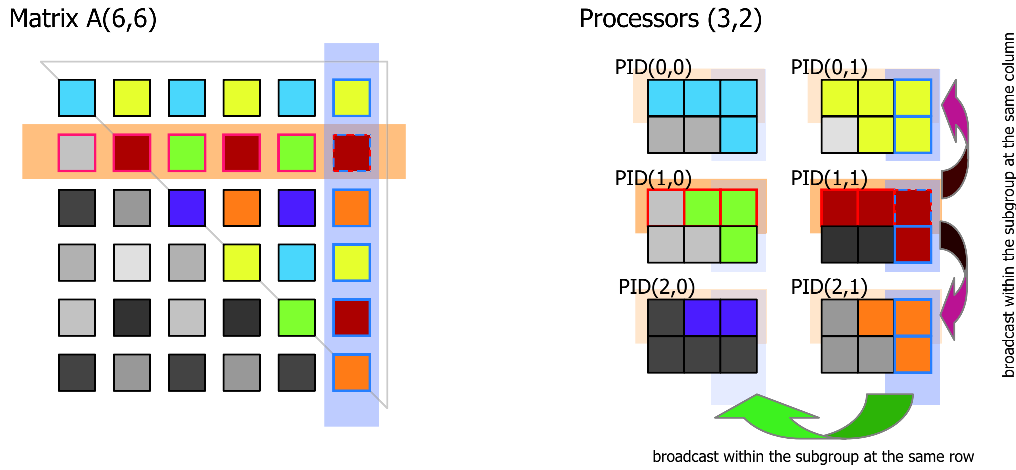

Fig. 4 2D data distribution (left) and data layout and process mapping (right)

Table 1 Hardware and software configuration for our development on the T2K cluster system

Hardware	
CPU:	AMD Barcelona quad core 2.3 GHz
	4 processor sockets/node
	256 nodes are allowed on this study
Main memory:	DDR2 32 GB/node
	(28 GB is available for users)
	42.7 GB/s/node
Theoretical peak:	147.2 GFLOPS/node
	27.683 TFLOPS (256 nodes)
Interconnection:	Myrinet 5 GB/s×2 (full-bisection)
Software	
OS:	Redhat Enterprise Linux 5
Compiler:	Intel Fortran compiler 11.0.074
	options `-g -axW -openmp`
MPI:	MPICH-MX (MPI-1.2)
BLAS:	Intel MKL 11.0.074
LAPACK:	Intel MKL 11.0.074
ScaLAPACK:	version 1.8.0 (compiled)
Affiliation control:	`numactl`
	`--physcpubind=`*4*MPIID*:*4*MPIID+3*

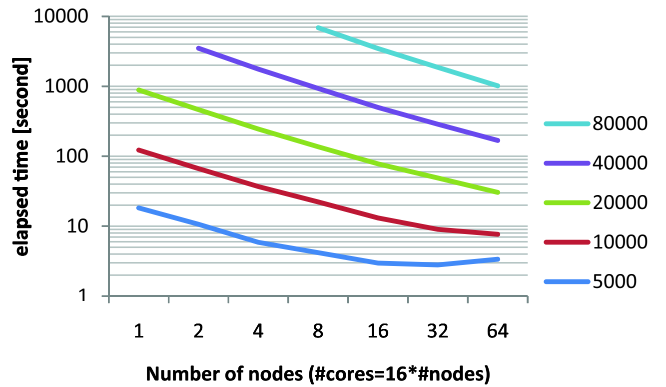

Fig. 5 Elapsed time for full-diagonalization using eigen_s

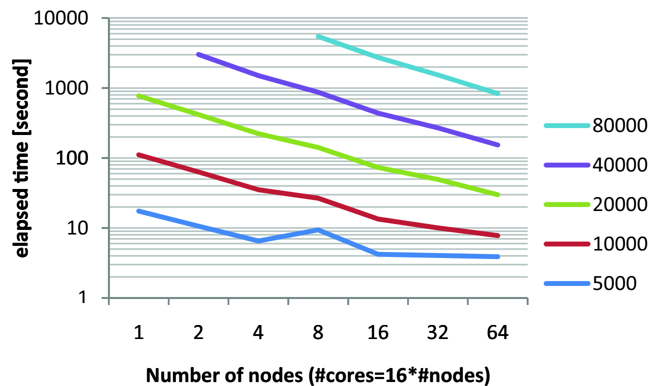

Fig. 6 Elapsed time for full-diagonalization using eigen_sx

theoretical peak. This suggests that, even if eigen_s solves a larger eigenvalue problem, the performance is bounded by 20% of the theoretical peak.

Based on previous studies,[7] we found that when the dimension of the problem becomes large, the cost of the divide and conquer method becomes smaller than the costs of the other two steps.

(2) Performance Prediction with eigen_sx

As described in the previous section, the performance of eigen_sx is easily improved using fast BLAS implementations. In addition, we can expect to reduce the communication overhead because data blocking also agglomerate couple of message passing. This suggests that for larger-scale problems, for example, diagonalization for a more than100,000-dimensional matrix, eigen_sx acts as a eigenvalue solver. Considering the first step of both eigen_s and eigen_sx,we perform another large numerical experiment, as shown in **Figs. 8** and **9**. In this experiment, we use 256 nodes, and 4,096

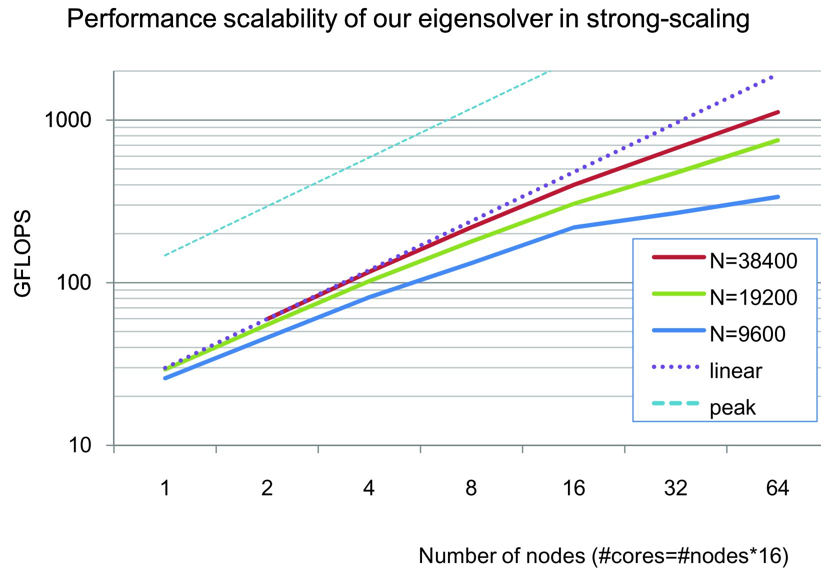

Fig. 7 Performance scalability of eigen_s on the T2K cluster system

```
N=       200000 M=         128  NM=        6256
NUM.OF.PROCESS=        1024 (          32          32 )
NUM.OF.THREADS=           4
calc (u,beta)       258.480158567429[s]
mat-vec (Au)        2783.94904828072[s]        1915.74387348326[GFLOPS]
2update (A-uv-vu)   268.623162031174[s]        19854.3315960014[GFLOPS]
calc v              61.3560543060303[s]
v=v-(UV+VU)u        120.128936290741[s]
UV post reduction   1.13441467285156[s]
COMM_STAT
   BCAST  ::     96.1706650257111[s]
   REDUCE ::     248.613152980804[s]
   REDIST ::    0.000000000000000E+000[s]
   GATHER ::     14.1590096950531[s]
TRD-BLK       200000      3513.90630698204[s]          3035.55807548775       [GFLOPS]
```

Fig. 8 Computing log 1: the largest performance test for eigen_s on the T2K cluster system

```
N=       200000 M=         128  NM=        6256
NUM.OF.PROCESS=        1024 (          32          32 )
NUM.OF.THREADS=           4
calc (u,beta)       208.912147521973[s]
mat-vec (Au)        1510.60236501694[s]        3530.60041268606[GFLOPS]
2update (A-uv-vu)   247.699535608292[s]        21531.4627871059[GFLOPS]
calc v              63.7259461879730[s]
v=v-(UV+VU)u        177.137169361115[s]
UV post reduction   3.22072529792786[s]
COMM_STAT
   BCAST  ::     86.7426195144653[s]
   REDUCE ::     100.455729246140[s]
   REDIST ::    0.000000000000000E+000[s]
TRD-BLK       200000      2227.20995497704[s]          4789.25062400624       [GFLOPS]
```

Fig. 9 Computing log 2: the largest performance test for eigen_sx on the T2K cluster system

cores. Unfortunately, we were not able to obtain the results in the other two steps due to the limitation of the CPU time. Eigen_s and eigen_sx achieve performances of 3.0 TFLOPS and 4.7 TFLOPS, respectively, which are equivalent to 8% and 12% of the theoretical peak performance, respectively.

We predict the overall performance based on these log files. Since the greater part of the third step consists of DGEMM in BLAS, back-transformation is expected to perform fine. If back-transformation exhibits the performance listed in the GFLOPS rate at third column of the line of '2update (A-uv-vu)' in Fig. 8, the computational time in the third step can be expected to be approximately 800 seconds. (The computational cost discussed herein is $\frac{2}{3}N^3$. In contrast, that of the third step of back-transformation is $2N^3$.) According to previous studies, the cost of the second step becomes much smaller than that of the third step. We estimate the cost of this part to be 10% of the cost of the third step. Therefore, the overall cost of eigen_s and eigen_sx are roughly estimated to be 4,473 seconds (=3,513+160+800) and 3,507 seconds (=2,227+160*3+800), respectively. We can save approximately 1,000 seconds by using eigen_sx on the T2K system.

(3) Toward the Development of the K-Computer

As described in the introduction, the memory bandwidth of the K-computer is approximately twice as wide as that of the T2K system. The performance ratio between 'mat-vec' and '2update' clearly becomes smaller on the K-computer. The ratio for eigen_s on the K-computer is expected to be similar to the ratio for eigen_sx on the T2K system. The performance on 'mat-vec' improves by the replacement to the matrix-matrix product at most two times. The performance ratio between 'mat-vec' and '2update' may improve from 1:5 to 1:4. If the DGEMM routine is assumed to account for 80% of the theoretical performance of the K-computer, we predict the performance more than 16 to 20% of the theoretical peak. Hence, the performance of eigen_sx on the K-computer exceeds that on the T2K system because the memory bandwidth of the K-computer is wider. If a narrower Byte/Flop system is required, we can accelerate the narrow-band reduction step by using a wider intermediate-band matrix.[6)] Although this increases the computational cost of the divide-and-conquer step, and this trade-off must be considered, this approach is effective for a system in which the Byte/Flop rate is smaller. The resulting performance improvement is significant in large-scale simulations.

Eigen_s and eigen_sx are developed for the next-generation supercomputer K-computer. The K-computer is assumed to possess more than 100,000 interconnected cores. In addition to the innovation to the memory bandwidth, improved effectiveness of multicore processors and reduced communication are key consideration in large-scale system. The communication costs reported in the logs (Figs. 8 and 9) demonstrate that eigen_sx requires less communication, thus it is also expected to perform well on K-computer.

IV. Related Research

The target system of eigen_s is a typical homogenous system in the peta-scale computing era. It is thought that the K-computer and similar next-generation systems will be the last generation of systems that have a homogeneous architecture. In an effort to realize next-generation or next-next-generation supercomputing, a number of computer architectures have been proposed, and several of these have been ranked in top positions according to the top500 benchmark. In this section, we discuss research on eigenvalue solvers or linear algebra software related to next-generation computing.

1. Many-Core and GPGPU: General Purpose Computing on GPU's

In November 2010, the GPU cluster system was ranked highest and two other GPU systems were ranked in the top five according to the Linpack benchmark.[1)] Although it is unclear how long this situation will continue in the future, GPUs and many-core processors are one of the key issues to exe-scale era. PLASMA[9)] and MAGMA[10)] are well-known software packages for performing numerical linear algebra operations. PLASMA is a new-generation LAPACK implementation that introduced effective task scheduling and communication avoiding algorithms. MAGMA focuses on the GPGPU and provides the GPU version of LAPACK with a similar scheduling mechanism and a highly tuned BLAS, namely, MAGMABLAS.

2. Massively Parallel Approach

Since the performance on a single processor is at most 1 to 10 TFLOPS, a PFLOPS class system must be massively parallel and have 1,000 or more interconnected nodes. As the memory wall becomes more severe, the network latency and throughput walls also merge. Naturally, an effective communication algorithm that takes into consideration the network bottleneck is important. Eigen_s and eigen_sx adopt flexible 2D data-distribution and CPU-mapping. Katagiri and Itoh[11)] also proposed a grid-free algorithm for a massively parallel dense symmetric eigensolver with a communication splitting multicasting algorithm. Furthermore, their algorithm can be applied to small matrices in massively parallel processing that takes an appropriate process mapping.

V. Conclusion

In the present paper, we have introduced eigen_s and eigen_sx, which are high-performance, high-scalable eigenvalue solvers that were developed for the K-computer, a next-generation supercomputer. We introduced several optimization techniques, such as a blocking scheme to overcome the shortage of memory bandwidth. This approach yields a good performance improvement. Furthermore, this approach can avoid unnecessary communication overhead. Although we have not sufficiently examined the performance of the proposed eigensolvers with respect to large-scale problems involving a large number of computational cores, the results of the present study indicate that eigen_sx will be able to perform well on the K-computer. We therefore conclude that eigen_s and eigen_sx are promising eigensolvers for present and future parallel computer systems.

In future studies, we intend to examine the performance of eigen_s and eigen_sx on larger computer systems having more than 10,000 cores. We have thus far confirmed that eigen_s

works with 8,192 processor cores on the T2K cluster system. Furthermore, communication algorithms that yield fewer data transfers must be investigated in order to efficiently apply eigen_s and eigen_sx to a huge number of processor cores.

Finally, the authors would like to thank Huu Phuong Pham for his support of the present study. The present research was supported in part by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B), 21300013, 21300007, 20300007, 20500044, and Scientific Research on Priority Areas, 21013014.

References

1) See the past records in top 500 Homepage, http://www.top500.org.
2) T. Maruyama, T. Yoshida, R. Kan *et al.*, "Sparc64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing," *Micro, IEEE*, **30**[2], 20–40 (2010).
3) S. Hammarling, D. Sorensen, J. Dongarra, "Block reduction of matrices to condensed forms for eigenvalue computations," *J. Comput. Appl. Math.*, **27**, 215–227 (1987).
4) J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem," *Numer. Math.*, **36**, 177–195 (1981).
5) H. P. Pham, T. Imamura, S. Yamada, M. Machida, "Novel approach in a divide and conquer algorithm for eigenvalue problems of real symmetric band matrices," *Proc. Joint Int. Conf. Supecomputing in Nuclear Applications + Monte Carlo 2010 (SNA+MC2010)*, Tokyo, Japan, Oct., 17–21, 2010 (2010), [USB-memory].
6) T. Imamura, S. Yamada, M. Machida, "Narrow-band reduction approach of a DRSM eigensolver on a multicore-based cluster system," B. Chapman, *et al.*, (eds.) Parallel Computing: From Multicores and GPU's to Peatascale, *Advances in Parallel Computing*, **19**, 91–98 (2010).
7) S. Yamada, T. Imamura, T. Kano, M. Machida, "High-performance computing for exact numerical approaches to quantum many body problems on the earth simulator," *Proc. IEEE/ACM SC06 conference*, Tampa, USA, Nov. 11-17, 2006 (2006), [CD-ROM].
8) Y. Ajima, S. Sumimoto, T. Shimizu, "Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers," *IEEE Computer*, **42**[11], 36–40 (2009).
9) A. Buttari, J. Dongarra, J. Kurzak *et al.*, "The Impact of Multicore on Math Software," *Proc. PARA 2006*, Umea, Sweden, June, 18–21, 2006, 1–10 (2006).
10) S. Tomov, R. Nath, H. Latif, J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," *Proc. 2010 IEEE Int. Symp. Parallel&Distributed Processing, Workshop and PhD Forum(IPDPSW)*, Anchorage, USA, April, 19–23, 2010, 1–8 (2010), [Electrical].
11) T. Katagiri, S. Itoh, "A massively parallel dense symmetric eigensolver with communication splitting multicasting algorithm," J. M. Laginha *et al.*, (eds.) *High Performance Computing for Computational Science —VECPAR2010*, LNCS **6449**, 139–150 (2011).