Rio Yokota
Weigang Wu (Eds.)

LNCS 10776

Supercomputing Frontiers

4th Asian Conference, SCFA 2018
Singapore, March 26–29, 2018
Proceedings

Springer Open

Contents

Performance Tools

Linear Algebra

Linear Algebra

High Performance LOBPCG Method for Solving Multiple Eigenvalues of Hubbard Model: Efficiency of Communication Avoiding Neumann Expansion Preconditioner

Susumu Yamada[1(✉)], Toshiyuki Imamura[2], and Masahiko Machida[1]

[1] Japan Atomic Energy Agency, Kashiwa, Chiba, Japan
yamada.susumu@jaea.go.jp
[2] RIKEN, Kobe, Hyogo, Japan

Abstract. The exact diagonalization method is a high accuracy numerical approach for solving the Hubbard model of a system of electrons with strong correlation. The method solves for the eigenvalues and eigenvectors of the Hamiltonian matrix derived from the Hubbard model. Since the Hamiltonian is a huge sparse symmetric matrix, it was expected that the LOBPCG method with an appropriate preconditioner could be used to solve the problem in a short time. This turned out to be the case as the LOBPCG method with a suitable preconditioner succeeded in solving the ground state (the smallest eigenvalue and its corresponding eigenvector) of the Hamiltonian. In order to solve for multiple eigenvalues of the Hamiltonian in a short time, we use a preconditioner based on the Neumann expansion which uses approximate eigenvalues and eigenvectors given by LOBPCG iteration. We apply a communication avoiding strategy, which was developed considering the physical properties of the Hubbard model, to the preconditioner. Our numerical experiments on two parallel computers show that the LOBPCG method coupled with the Neumann preconditioner and the communication avoiding strategy improves convergence and achieves excellent scalability when solving for multiple eigenvalues.

1 Introduction

Since the High-T_c superconductor was discovered many physicists have tried to understand the mechanism behind the superconductivity. It is believed that strong electron correlations underlie the phenomenon, however the exact mechanism is not yet fully understood. One of the numerical approaches to the problem is the exact diagonalization method. In this method the eigenvalue problem is solved for the Hamiltonian derived exactly from the Hubbard model [1,2], which is a model of a strongly-correlated electron system. When we solve the ground state (the smallest eigenvalue and its corresponding eigenvector) of the

© The Author(s) 2018
R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 243–256, 2018.
https://doi.org/10.1007/978-3-319-69953-0_14

$\boldsymbol{x}_0 :=$ an initial guess, $\boldsymbol{p}_0 := 0$
$\boldsymbol{x}_0 := \boldsymbol{x}_0/|\boldsymbol{x}_0|,\ X_0 := A\boldsymbol{x}_0,\ P_0 := 0,\ \mu_{-1} := (\boldsymbol{x}_0, X_0),$
$\boldsymbol{w}_0 := X_0 - \mu_{-1}\boldsymbol{x}_0$
do k=0, ... until convergence
$W_k := A\boldsymbol{w}_k$
$S_A := \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}^T\{W_k, X_k, P_k\}$
$S_B := \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}^T\{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}$
Solve the generalized eigenvalue problem $S_A\boldsymbol{v} = \mu S_B\boldsymbol{v}$ to obtain the smallest eigenvalue μ_k and the corresponding eigenvector $\boldsymbol{v} = (\alpha, \beta, \gamma)^T$,
$\boldsymbol{x}_{k+1} := \alpha\boldsymbol{w}_k + \beta\boldsymbol{x}_k + \gamma\boldsymbol{p}_k,\ \boldsymbol{x}_{k+1} := \boldsymbol{x}_{k+1}/|\boldsymbol{x}_{k+1}|,$
$\boldsymbol{p}_{k+1} := \alpha\boldsymbol{w}_k + \gamma\boldsymbol{p}_k,\ \boldsymbol{p}_{k+1} := \boldsymbol{p}_{k+1}/|\boldsymbol{p}_{k+1}|$
$X_{k+1} := \alpha W_k + \beta X_k + \gamma P_k,\ X_{k+1} := X_{k+1}/|\boldsymbol{x}_{k+1}|,$
$P_{k+1} := \alpha W_k + \gamma P_k,\ P_{k+1} := P_{k+1}/|\boldsymbol{p}_{k+1}|$
$\boldsymbol{w}_{k+1} := T^{-1}(X_{k+1} - \mu_k\boldsymbol{x}_{k+1}),\ \boldsymbol{w}_{k+1} := \boldsymbol{w}_{k+1}/|\boldsymbol{w}_{k+1}|$
enddo

Fig. 1. Algorithm of LOBPCG method for matrix A. Here the matrix T is a preconditioner.

Hamiltonian, we can understand its properties at absolute zero (−273.15 °C). Many computational methods for the problem have been proposed. Since the Hamiltonian from the Hubbard model is a huge sparse symmetric matrix, an iteration method, such as the Lanczos method [3] or the LOBPCG method (see Fig. 1) [4,5], is usually utilized for solving the eigenvalue problem.

The convergence of the LOBPCG method depends strongly on the use of a preconditioner. We previously confirmed that the zero-shift point Jacobi preconditioner, which is a shift-and-invert preconditioner [6] using an approximate eigenvalue, gives excellent convergence properties for the Hubbard model with the trapping potential [7]. However, we also reported that the benefit of the preconditioner strongly depends on the characteristics of the non-zero elements of the Hamiltonian and that the preconditioner does not always improve the convergence [8]. Therefore we proposed a novel preconditioner using the Neumann expansion for solving the ground state of the Hamiltonian and demonstrated that this preconditioner improves convergence for a Hamiltonian that is difficult to solve with the zero-shift point Jacobi preconditioner [8]. Moreover we applied a communication avoiding strategy, which was developed considering the properties of the Hubbard model, to the preconditioner.

In order to understand more details of strongly correlated electron systems in particular properties at temperatures near absolute zero, we must solve for the several smallest eigenvalues and corresponding eigenvectors of the Hamiltonian. The LOBPCG method can solve multiple eigenvalues by using a block of vectors.

In this paper, we extend the Neumann expansion preconditioner to the LOBPCG method for solving multiple eigenvalues and corresponding eigenvectors. Moreover, we demonstrate that the preconditioner improves the convergence properties and can achieve excellent parallel performance.

The paper is structured as follows. In Sect. 2 we briefly introduce related work for solving the ground state of the Hubbard model using the LOBPCG method. Section 3 describes the use of the Neumann expansion preconditioner with the communication avoiding strategy for solving for multiple eigenvalues and their corresponding eigenvectors. Section 4 demonstrates the parallel performance of the algorithm on the SGI ICE X and K supercomputers. A summary and conclusions are given in Sect. 5.

2 Related Work

2.1 Hamiltonian-Vector Multiplication

When solving the ground state of a symmetric matrix using the LOBPCG method, the most time-consuming operation is the matrix-vector multiplication. The Hamiltonian derived from the Hubbard model (see Fig. 2) is

$$H = -t \sum_{i,j,\sigma} c^{\dagger}_{j\sigma} c_{i\sigma} + \sum_{i} U_i n_{i\uparrow} n_{i\downarrow}, \tag{1}$$

where t is the hopping parameter from one site to another, and U_i is the repulsive energy for double occupation of the i-th site by two electrons [1,2,7]. Quantities $c_{i,\sigma}$, $c^{\dagger}_{i,\sigma}$ and $n_{i,\sigma}$ are the annihilation, creation, and number operator of an electron with pseudo-spin σ on the i-th site, respectively. The indices in formula (1) for the Hamiltonian denote the possible states for electrons in the model. The dimension of the Hamiltonian for the n_s-site Hubbard model is

$$\binom{n_s}{n_\uparrow} \times \binom{n_s}{n_\downarrow},$$

where $n_\uparrow$ and $n_\downarrow$ are the number of the up-spin and down-spin electrons, respectively.

The diagonal element in formula (1) is derived from the repulsive energy U_i in the corresponding state. The hopping parameter t affects non-zero elements with

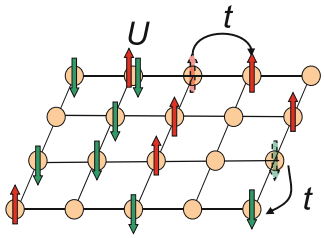


Fig. 2. A schematic figure of the 2-dimensional Hubbard model, where t is the hopping parameter and U is the repulsive energy for double occupation of a site. Up arrows and down arrows correspond to up-spin and down-spin electrons, respectively.

column-index corresponding to the original state and row-index corresponding to the state after hopping. Since the ratio U/t greatly affects the properties of the model, we have to execute many simulations varying this ratio to reveal the properties of the model.

When considering the physical properties of the model, we can split the Hamiltonian-vector multiplication as

$$Hv = Dv + (I_{\downarrow} \otimes A_{\uparrow})v + (A_{\downarrow} \otimes I_{\uparrow})v, \tag{2}$$

where $I_{\uparrow(\downarrow)}$, $A_{\uparrow(\downarrow)}$ and D are the identity matrix, a sparse symmetric matrix derived from the hopping of an up-spin electron (a down-spin electron), and a diagonal matrix from the repulsive energy, respectively [7]. Since there is no regularity in the state change by electron hopping, the distribution of non-zero elements in matrix $A_{\uparrow(\downarrow)}$ is irregular.

Next, a matrix V is constructed by the following procedures from a vector v. First, decompose the vector v into n blocks, and order in the two-dimensional manner as follows,

$$v = (\underbrace{v_{1,1}, v_{2,1}, \ldots, v_{m_{\uparrow},1}}_{\text{the first block}}, \underbrace{v_{1,2}, v_{2,2}, \ldots, v_{m_{\uparrow},2}}_{\text{the second block}}, \cdots, \underbrace{v_{1,m_{\downarrow}}, v_{2,m_{\downarrow}}, \ldots, v_{m_{\uparrow},m_{\downarrow}}}_{\text{the } m_{\downarrow}\text{-th block}})^T,$$

where $m_{\uparrow}$ and $m_{\downarrow}$ are the dimensions of the Hamiltonian for up-spin and down-spin electrons, i.e.

$$m_{\uparrow} = \binom{n_s}{n_{\uparrow}}, m_{\downarrow} = \binom{n_s}{n_{\downarrow}}.$$

The subscripts on each element of v formally indicate the row and column within the matrix V. Therefore V is a dense matrix. The k-th elements of the matrix D, d_k, are used in the same manner to define a new matrix $\bar{D}$. The multiplication in Eq. (2) can then be written as

$$V_{i,j}^{new} = \bar{D}_{i,j} V_{i,j} + \sum_k A_{\uparrow i,k} V_{k,j} + \sum_k V_{i,k} A_{\downarrow j,k} \tag{3}$$

where the subscript i, j of the matrix is represented as the (i, j)-th element and V and $\bar{D}$. Accordingly we can parallelize the multiplication $Y = HV (\equiv Hv)$ as follows:

CAL 1: $Y^c = \bar{D}^c \odot V^c + A_{\uparrow} V^c$,
COM 1: all-to-all communication from V^c to V^r,
CAL 2: $W^r = V^r A_{\downarrow}^T$,
COM 2: all-to-all communication from W^r to W^c,
CAL 3: $Y^c = Y^c + W^c$.

where superscripts c and r denote column wise and row wise partitioning of the matrix data for the parallel calculation. The operator $\odot$ means an element wise multiplication. The parallelization strategy requires two all-to-all communication operations per multiplication.

2.2 Preconditioner of LOBPCG Method for Solving the Ground State of Hubbard Model

Zero-Shift Point Jacobi Preconditioner. A suitable preconditioner improves the convergence properties of the LOBPCG method. As a consequence many preconditioners have been proposed. Preconditioners for the Hamiltonian derived from the Hubbard model also have been proposed. For the Hubbard model, the zero-shift point Jacobi (ZSPJ) preconditioner, which is a shift-and-invert preconditioner using an approximate eigenvalue obtained during LOBPCG iteration, has excellent convergence properties for Hamiltonians where the diagonal elements predominate over the off-diagonal elements, i.e. cases where the repulsive energy U is large [7,8].

Neumann Expansion Preconditioner. For the Hubbard model with a small repulsive energy, a preconditioner using the Neumann expansion was previously proposed [8]. The expansion is

$$(I - M)^{-1} = I + M + M^2 + M^3 + \cdots . \tag{4}$$

The expansion converges when the operator norm of the matrix M is less than 1 ($||M||_{op} < 1$) [9]. Here the matrix M is

$$M = I - \frac{2}{\lambda_{\max} - \lambda_{\min}}(H - \lambda_{\min} I),$$

where $\lambda_{\min}$ and $\lambda_{\max}$ are the smallest and largest eigenvalues, respectively. When the exact eigenvalues are utilized for $\lambda_{\min}$ and $\lambda_{\max}$, $||M||_{op}$ is equal to 1. Since the LOBPCG method calculates an approximation of the smallest eigenvalue, we consider the residual error of this approximation and make a low estimate of $\lambda_{\min}$. The Gershgorin circle theorem is used to assign a $\lambda_{\max}$ that is an overestimate of the true value. The inequality $||M||_{op} < 1$ is hence obeyed and the expansion (4) can converge, i.e. the inverse matrix of $\frac{2}{\lambda_{\max} - \lambda_{\min}}(H - \lambda_{\min} I)$. The expansion is an effective preconditioner for the smallest eigenvalue $\lambda_{\min}$. In practice the Gershgorin circle theorem may give estimates for $\lambda_{\max}$ that are much too large. Multiplying by a damping factor α can help alleviate this inefficiency. We found 0.9 to be an appropriate α in numerical tests.

2.3 Communication Avoiding Neumann Expansion Preconditioner for Hubbard Model

When we execute the LOBPCG method with the s-th order Neumann expansion preconditioner, we calculate $s + 1$ matrix-vector multiplications, Hv, H^2v, ..., and $H^{s+1}v$, per iteration. As the multiplications $(I_\downarrow \otimes A_\uparrow)$ and $(A_\downarrow \otimes I_\uparrow)$ are commutative Yamada et al. proposed a communication avoiding strategy for the Hamiltonian-vector multiplication [8]. Then, H^2 is given as

$$\begin{aligned} H^2 &= (I_\downarrow \otimes A_\uparrow)(D + (I_\downarrow \otimes A_\uparrow)) + (A_\downarrow \otimes I_\uparrow)(D + (A_\downarrow \otimes I_\uparrow)) \\ &\quad + D(D + (I_\downarrow \otimes A_\uparrow) + (A_\downarrow \otimes I_\uparrow)) + (I_\downarrow \otimes A_\uparrow)(A_\downarrow \otimes I_\uparrow) + (A_\downarrow \otimes I_\uparrow)(I_\downarrow \otimes A_\uparrow) \\ &= (I_\downarrow \otimes A_\uparrow)(D + (I_\downarrow \otimes A_\uparrow) + 2(A_\downarrow \otimes I_\uparrow)) \\ &\quad + (A_\downarrow \otimes I_\uparrow)(D + (A_\downarrow \otimes I_\uparrow)) + D(D + (I_\downarrow \otimes A_\uparrow) + (A_\downarrow \otimes I_\uparrow)). \end{aligned}$$

As a result, $Y_1 = Hv$ and $Y_2 = H^2 v$ can be calculated by the following algorithm:

CAL 1: $Y^c = \bar{D}^c \odot V^c + A_{\uparrow} V^c$,
COM 1: all-to-all communication from V^c to V^r,
CAL 2: $W^r = V^r A_{\downarrow}^T$,
COM 2: all-to-all communication from W^r to W^c,
CAL 3: $Y_1^c = Y^c + W^c$,
CAL 4: $Y^c = Y_1^c + W^c$,
CAL 5: $Y^c = \bar{D}^c \odot Y_1^c + A_{\uparrow} Y^c$,
CAL 6: $W^r = \bar{D}^r \odot V^r + W^r$,
CAL 7: $W^r = W^r A_{\downarrow}^T$,
COM 3: all-to-all communication from W^r to W^c,
CAL 8: $Y_2^c = Y^c + W^c$.

The algorithm requires three all-to-all communication operations. On the other hand, when using the original algorithm described in Sect. 2.1, four all-to-all communication operations are required to calculate the same multiplication. However, the new algorithm has extra calculations, CAL 4 and CAL 6, as compared to the original one. Therefore when the cost of one all-to-all communication operation is larger than that of the extra calculations, we expect to achieve speedup with the communication avoiding strategy. The algorithm can not be directly applied to the multiplication H^{s+1} for $s \geq 2$. In this case, the multiplication H^{s+1} is calculated by appropriately combining Hv and $H^2 v$ operations.

3 Neumann Expansion Preconditioner for Multiple Eigenvalues of Hubbard Model

3.1 How to Calculate Multiple Eigenvalues Using LOBPCG Method

The LOBPCG method for solving the m smallest eigenvalues and corresponding eigenvectors carries out recurrence with m vectors simultaneously (see Fig. 3). In this algorithm, the generalized eigenvalue problem has to be solved. We can solve the problem using the LAPACK function `dsyev`, if the matrix S_B is a positive definite matrix. Although theoretically S_B is always a positive definite matrix, numerically this is not always the case. The reason is that the norms of the vectors $w_k^{(i)}$ and $p_k^{(i)}$ $(i = 1, 2, \ldots, m)$ become small as the LOBPCG iteration converges, and it is possible that trailing digits are lost in the calculation of S_B. Therefore we set the matrix S_B to the identity matrix by orthogonalizing the vectors per iteration. In the following numerical tests, we utilize the TSQR method for the orthgonalization [10,11].

$\boldsymbol{x}_0^{(i)}$:= an initial guess, $\boldsymbol{p}_0^{(i)} := 0$, $i = 1, \ldots, m$
$\boldsymbol{x}_0^{(i)} = \boldsymbol{x}_0^{(i)} / ||\boldsymbol{x}_0^{(i)}||$, $i = 1, \ldots, m$
$\boldsymbol{X}_0^{(i)} := A\boldsymbol{x}_0^{(i)}$, $\boldsymbol{P}_0^{(i)} := 0, i = 1, \ldots, m$
$\mu_{-1}^{(i)} := (\boldsymbol{x}_0^{(i)}), \boldsymbol{X}_0^{(i)}$, $\boldsymbol{w}_0^{(i)} := \boldsymbol{X}_0^{(i)} - \mu_{-1}^{(i)} \boldsymbol{x}_0^{(i)}$, $i = 1, \ldots, m$
do k=0, ... until convergence
$\boldsymbol{W}_k^{(i)} := A\boldsymbol{w}_k^{(i)}$, $i = 1, \ldots, m$
$S_A := \{\boldsymbol{w}_k^{(1)}, \ldots, \boldsymbol{w}_k^{(m)}, \boldsymbol{x}_k^{(1)}, \ldots, \boldsymbol{x}_k^{(m)}, \boldsymbol{p}_k^{(1)}, \ldots, \boldsymbol{p}_k^{(m)}\}^T$
$\{\boldsymbol{W}_k^{(1)}, \ldots, \boldsymbol{W}_k^{(m)}, \boldsymbol{X}_k^{(1)}, \ldots, \boldsymbol{X}_k^{(m)}, \boldsymbol{P}_k^{(1)}, \ldots, \boldsymbol{P}_k^{(m)}\}$
$S_B := \{\boldsymbol{w}_k^{(1)}, \ldots, \boldsymbol{w}_k^{(m)}, \boldsymbol{x}_k^{(1)}, \ldots, \boldsymbol{x}_k^{(m)}, \boldsymbol{p}_k^{(1)}, \ldots, \boldsymbol{p}_k^{(m)}\}^T$
$\{\boldsymbol{w}_k^{(1)}, \ldots, \boldsymbol{w}_k^{(m)}, \boldsymbol{x}_k^{(1)}, \ldots, \boldsymbol{x}_k^{(m)}, \boldsymbol{p}_k^{(1)}, \ldots, \boldsymbol{p}_k^{(m)}\}$
Solve the generalized eigenvalue problem $S_A \boldsymbol{v} = \mu S_B \boldsymbol{v}$ to obtain the m smallest eigenvalues $\mu^{(i)}$ and the corresponding eigenvectors $\boldsymbol{v}^{(i)} = (\alpha^{(i)}, \beta^{(i)}, \gamma^{(i)})^T$ ($i = 1, \ldots, m$),
$\boldsymbol{x}_{k+1}^{(i)} := \sum_{j=1}^{m} \{\alpha^{(j)} \boldsymbol{w}_k^{(j)} + \beta^{(j)} \boldsymbol{x}_k^{(j)} + \gamma^{(j)} \boldsymbol{p}_k^{(j)}\}$, $i = 1, \ldots, m$
$\boldsymbol{p}_{k+1}^{(i)} := \sum_{j=1}^{m} \{\alpha^{(j)} \boldsymbol{w}_k^{(j)} + \gamma^{(j)} \boldsymbol{p}_k^{(j)}\}$, $i = 1, \ldots, m$
$\boldsymbol{X}_{k+1}^{(i)} := \sum_{j=1}^{m} \{\alpha^{(j)} \boldsymbol{W}_k^{(j)} + \beta^{(j)} \boldsymbol{X}_k^{(j)} + \gamma^{(j)} \boldsymbol{P}_k^{(j)}\}$, $i = 1, \ldots, m$
$\boldsymbol{P}_{k+1}^{(i)} := \sum_{j=1}^{m} \{\alpha^{(j)} \boldsymbol{W}_k^{(j)} + \gamma^{(j)} \boldsymbol{P}_k^{(j)}\}$, $i = 1, \ldots, m$
$\mu_k^{(i)} := (\boldsymbol{x}_{k+1}^{(i)}, \boldsymbol{X}_{k+1}^{(i)}) / (\boldsymbol{x}_{k+1}^{(i)}, \boldsymbol{x}_{k+1}^{(i)})$, $i = 1, \ldots, m$
$\boldsymbol{w}_{k+1}^{(i)} := {T^{(i)}}^{-1} (\boldsymbol{X}_{k+1}^{(i)} - \mu_k^{(i)} \boldsymbol{x}_{k+1}^{(i)})$, $i = 1, \ldots, m$
enddo

Fig. 3. LOBPCG method for solving the m smallest eigenvalues and corresponding eigenvectors. $T^{(i)}$ is a preconditioner for the i-th smallest eigenvalues. This algorithm requires m matrix-vector multiplication operations and m preconditioned ones per iteration.

3.2 Neumann Expansion Preconditioner of LOBPCG Method for Solving Multiple Eigenvalues

When we calculate multiple eigenvalues (and corresponding eigenvectors) using the LOBPCG method, we can individually apply a preconditioning operation to each vector corresponding to the eigenvectors. We set the matrix M_i using the Neumann expansion preconditioner for the i-th smallest eigenvalue λ_i of the Hamiltonian as

$$M_i = I - \frac{2}{\lambda_{\max} - \lambda_i}(H - \lambda_i I).$$

Since we obtain approximate eigenvalues after each iteration of the LOBPCG method, we consider the residual errors of these approximations to define an appropriate λ_i. The matrix M_i has $(i-1)$ eigenvalues whose absolute values are greater than or equal to 1. In this case, the Neumann expansion using M_i can not converge. The eigenvectors corresponding to the eigenvalues agree with those corresponding to the eigenvalues λ_1, λ_2, ..., λ_{i-1} of the Hamiltonian, and then, they are calculated during the LOBPCG iteration simultaneously. Accordingly, we orthogonalize the vectors $x_k^{(i)}$, $w_k^{(i)}$, and $p_k^{(i)}$ ($i = 1, 2, \ldots, m$) in the order that takes away the components of the vectors $x_k^{(1)}$, $x_k^{(2)}$, ..., $x_k^{(i-1)}$ from $w_k^{(i)}$ given

by the Neumann expansion preconditioner using M_i. That is, we orthogonal the vectors utilizing the algorithm including the following operation

$$w_k^{(j)} := w_k^{(j)} - \sum_{i=1}^{j-1} (w_k^{(j)}, x_k^{(i)}) x_k^{(i)}. \tag{5}$$

The formula (5) can approximately remove the components of the eigenvectors corresponding to the eigenvalues, whose absolute values are greater than or equal to 1, from the preconditioned vectors. Therefore we expect that the Neumann expansion using M_i becomes an appropriate preconditioner for solving for multiple eigenvalues.

4 Performance Result

4.1 Computational Performance and Convergence Property

We examined the computational performance and convergence properties of the LOBPCG method. We solved the 2-D 4×5-site Hubbard model with 5 up-spin electrons and 5 down-spin ones. The dimension of the Hamiltonian derived from the model is about 240 million. The number of non-zero off-diagonal elements is about 1.6 billion. We solved for one, five and 10 eigenvalues (and corresponding eigenvectors) of the Hamiltonian on 768 cores (64 MPI processes $\times$ 12 OpenMP threads) of the SGI ICE X supercomputer (see Table 1) in Japan Atomic Energy Agency (JAEA). Table 2 shows the results for a weak interaction case ($U/t = 1$) and a strong one ($U/t = 10$). Table 3 shows the elapsed times of some representative operations.

The results for $U/t = 1$ indicate that point Jacobi (PJ) and zero-shift point Jacobi (ZSPJ) preconditioners hardly improve the convergence compared to without using a preconditioner at all. When we solve for many eigenvalues, the PJ and ZSPJ preconditioners have little effect on the speed of the calculation. On the other hand, the Neumann expansion preconditioner can decrease the number of iterations required for convergence. Moreover, the larger the Neumann expansion series s, the fewer iterations required. When we solve for only

Table 1. Details of SGI ICE X

Processor	Intel Xeon E5-2680v3 (2.5 GHz, 30 MB L2 cache)
FLOPS per processor	480 GFLOPS
Number of cores per CPU	12
Number of processors per node	2
Memory of node	64 GB
Memory bandwidth	68 GB/s
Network	Infini Band 6.8 GB/s
Compiler	Intel compiler

Table 2. Elapsed time and number of iterations for convergence of LOBPCG method using zero-shift point Jacobi (ZSPJ), Neumann expansion (NE), or communication avoiding Neumann expansion (CANE) preconditioner. Here, s is the number of the Neumann expansion series.

(a) One eigenvalue (The ground state)									
	Number of iterations (top) & Elapsed time (sec) (bottom)								
	No precon.	PJ	ZSPJ	NE			CANE		
				$s=1$	$s=2$	$s=3$	$s=1$	$s=2$	$s=3$
$U/t=1$	133	133	132	69	59	46	69	59	46
	9.16	9.19	9.13	8.79	11.06	10.78	7.40	9.18	8.94
$U/t=10$	184	132	124	95	81	65	94	81	64
	13.03	9.08	8.61	12.07	14.65	15.62	10.05	12.62	12.79
(b) 5 eigenvalues									
	Number of iterations (top) & Elapsed time (sec) (bottom)								
	No precon.	PJ	ZSPJ	NE			CANE		
				$s=1$	$s=2$	$s=3$	$s=1$	$s=2$	$s=3$
$U/t=1$	199	190	168	81	77	59	86	72	54
	171.75	164.57	145.28	89.44	103.31	93.69	87.89	91.21	77.16
$U/t=10$	293	217	240	159	156	108	155	142	105
	250.77	186.35	204.97	172.90	208.80	171.37	156.41	179.12	149.59
(c) 10 eigenvalues									
	Number of iterations (top) & Elapsed time (sec) (bottom)								
	No precon.	PJ	ZSPJ	NE			CANE		
				$s=1$	$s=2$	$s=3$	$s=1$	$s=2$	$s=3$
$U/t=1$	551	777	624	319	257	184	340	302	198
	1221.55	1672.69	1369.91	911.56	897.79	759.40	863.57	936.35	680.48
$U/t=10$	398	298	313	232	184	161	201	177	137
	996.19	740.98	763.42	720.22	704.14	705.03	579.98	607.46	515.60

the smallest eigenvalue, the total elapsed time increases as s increases. The reason is that the elapsed time of the Hamiltonian-vector multiplication operation is dominant over the whole calculation for solving the only smallest eigenvalue (see Table 3). When we solve multiple eigenvalues, the TSQR operation becomes dominant. Therefore when the series number s becomes large, it is possible to achieve speedup of the computation.

Next, we discuss the results for $U/t = 10$. The results indicate that the PJ preconditioner improves the convergence properties. On the other hand, ZSPJ for small m improves convergence, however, its convergence properties when solving for multiple eigenvalues are almost the same as those for the PJ preconditioner. When we solve for multiple eigenvalues using the Neumann expansion preconditioner, the solution is obtained faster than using the PJ or ZSPJ preconditioners. Moreover, as the Neumann expansion series s increases, the Neumann expansion

Table 3. Elapsed time for operations per iteration. This table shows the results using the zero-shift point Jacobi (ZSPJ), Neumann expansion (NE), and communication avoiding Neumann expansion (CANE). Here, the Neumann expansion series s is equal to 1. For $m = 1$, instead of executing TSQR, we calculate S_B ,moreover, ZSPJ preconditioner is calculated together with x, p, X, P.

	Elapse time per iteration (sec)								
	$m = 1$			$m = 5$			$m = 10$		
	ZSPJ	NE	CANE	ZSPJ	NE	CANE	ZSPJ	NE	CANE
Hw (& H^2w)	0.061	0.117	0.100	0.276	0.545	0.448	0.568	1.088	0.909
TSQR	—	—	—	0.407	0.408	0.407	1.498	1.503	1.502
S_A (& S_B)	0.007	0.007	0.007	0.073	0.073	0.073	0.255	0.257	0.254
x, p, X, P	0.008	0.007	0.007	0.107	0.122	0.121	0.301	0.331	0.332
Preconditioner	—	0.003	0.003	0.018	0.015	0.014	0.035	0.030	0.028

Table 4. Speedup ratio for the elapsed time per iteration using the Neumann expansion preconditioner and communication avoiding strategy.

	Speedup ratio								
	$m = 1$			$m = 5$			$m = 10$		
	$s = 1$	$s = 2$	$s = 3$	$s = 1$	$s = 2$	$s = 3$	$s = 1$	$s = 2$	$s = 3$
$U/t = 1$	1.19	1.20	1.21	1.08	1.06	1.11	1.13	1.13	1.20
$U/t = 10$	1.19	1.16	1.20	1.08	1.06	1.11	1.08	1.12	1.16

preconditioner improves the convergence properties and the total elapsed time decreases, especially when m is large.

Finally, we talk about the effect of the communication avoiding strategy. Table 4 shows the speedup ratio for the elapsed time using the Neumann expansion preconditioner per iteration and the communication avoiding strategy. In all cases the communication avoiding strategy realizes speedup. When we solve for only the smallest eigenvalue (and its corresponding eigenvector), the speedup ratio is almost the same as that for the matrix-vector multiplication, because the multiplication cost is dominant. On the other hand, when we solve for multiple eigenvalues, the calculation cost except the multiplication becomes dominant. Therefore the speedup ratio is a little smaller than that for only the multiplication. Furthermore, when the Neumann expansion series s is equal to 3, we confirm that the ratio improves. In this case, since four multiplications (Hw, H^2w, H^3w and H^4w) are executed per iteration, the ratio of the multiplication cost increases. Moreover, we can execute four multiplication operations by two communication avoiding multiplications. Therefore, the ratio for $s = 3$ is better than that for $s = 1$.

4.2 Parallel Performance

In order to examine the parallel performance of the LOBPCG method using the Neumann expansion preconditioner, we solved for the 10 smallest eigenvalues and corresponding eigenvectors of the Hamiltonian derived from the 4×5-site Hubbard model for $U/t = 1$ with 6 up-spin and 6 down-spin electrons. We used the LOBPCG method with ZSPJ, NE, and CANE preconditioners using hybrid parallelization on SGI ICEX in JAEA and the K computer in RIKEN (see Table 5). The results are shown in Table 6. The results indicate that all preconditioners achieve excellent parallel efficiency. The communication avoiding strategy on SGI ICEX decreases the elapsed time per iteration by about 15%. On the other hand, the communication avoiding strategy on the K computer did not realize speedup when using a small number of cores. The ratio of the network bandwidth to FLOPS per node of the K computer is larger than that of SGI ICEX, so it is possible that the cost of the extra calculations (CAL 4 & CAL 6) is larger than that of the all-to-all communication operation. However since the cost of the all-to-all communication operation increases as the number of the cores increases, the strategy realizes speedup on 4096 cores. Therefore, the strategy has a potential of speedup for parallel computing using a sufficiently large number of cores, even if the ratio of the network bandwidth to FLOPS is large.

Although the LOBPCG method using NE has four times more Hamiltonian-vector multiplications per iteration than the method with ZSPJ, the former takes about twice the elapsed time of the latter. The reason is that the calculation operations except the multiplication is dominant in this case. Therefore, we conclude that in order to solve for multiple eigenvalues of the Hamiltonian derived from the Hubbard model using the LOBPCG method in a short computation time, it is crucial to reduce the number of the iterations for the convergence even if the calculation cost of the preconditioner is large.

Table 5. Details of K computer

Processor	SPARCTM 64 VIIIfx (2 GHz, 6 MB L2 cache)
FLOPS per processor	128 GFLOPS
Number of cores per CPU	8
Number of processors per node	1
Memory of node	16 GB
Memory bandwidth	64 GB/s
Network	Torus network (Tofu) 5 GB/s
Compiler	Fujitsu compiler

Table 6. Parallel performance of LOBPCG method on SGI ICEX and K computer. This table shows the number of iterations, the total elapsed time, and the elapsed time per iteration of LOBPCG method using zero-shift point Jacobi (ZSPJ), Neumann expansion (NE), or communication avoiding Neumann expansion (CANE) preconditioner. Here, the Neumann expansion series s is 3.

(a) SGI ICEX			
	Number of iterations (top)		
	Elapsed time (sec) (middle)		
	Elapsed time per iteration (sec) (bottom)		
	ZSPJ	NE	CANE
64 MPI × 12 OpenMP	591	226	225
	9501.694	5886.533	4921.302
	16.077	26.047	21.872
128 MPI × 12 OpenMP	605	246	229
	4611.478	3662.846	2909.048
	7.622	14.890	12.703
256 MPI × 12 OpenMP	601	244	226
	2259.070	2043.231	1603.456
	3.759	8.374	7.095
(b) K computer			
	Number of iterations (top)		
	Elapsed time (sec) (middle)		
	Elapsed time per iteration (sec) (bottom)		
	ZSPJ	NE	CANE
128 MPI × 8 OpenMP	503	209	230
	5775.971	3752.884	4596.063
	11.483	17.956	19.983
256 MPI × 8 OpenMP	551	224	303
	3231.566	2085.268	2974.883
	5.865	9.309	9.818
512 MPI × 8 OpenMP	862	243	250
	2548.534	1327.093	1130.652
	2.957	5.461	4.523

5 Conclusions

In this paper we applied the Neumann expansion preconditioner to the LOBPCG method to solve for multiple eigenvalues and corresponding eigenvectors of the Hamiltonian derived from the Hubbard model. We examined the convergence properties and parallel performance of the algorithms. Since the norm of the matrix used in the Neumann expansion should be less than 1, we transform

it using approximate eigenvalues calculated by the LOBPCG iteration and the upper bounds of the eigenvalues by the Gershgorin circle theorem. Moreover, we orthogonalize the iteration vectors in the order that removes the components of the eigenvectors corresponding to the eigenvalues, whose absolute values are greater than or equal to 1, from the preconditioned vectors.

The Neumann expansion preconditioner with the communication avoiding strategy can achieve speedup even for problems which are hardly improved by the conventional preconditioners. Furthermore, a numerical experiment indicated that the LOBPCG method using this preconditioner has excellent parallel efficiency on thousands cores, and the communication avoiding strategy based on the property of the Hubbard model realizes speedup for parallel computers if a sufficiently large number of cores are used. Therefore, we confirm that the preconditioner based on the Neumann expansion is suitable for solving the eigenvalue problem derived from the Hubbard model using the LOBPCG method.

Acknowledgments. Computations in this study were performed on the SGI ICE X at the JAEA and the K computer at RIKEN Advanced Institute for Computational Science (project ID:ra000005). This research was partially supported by JSPS KAKENHI Grant Number 15K00178.

References

1. Rasetti, M. (ed.): The Hubbard Model: Recent Results. World Scientific, Singapore (1991)
2. Montorsi, A. (ed.): The Hubbard Model. World Scientific, Singapore (1992)
3. Cullum, J.K., Willoughby, R.A.: Lanczos Algorithms for Large Symmetric Eigenvalue Computations, vol. 1: Theory. SIAM, Philadelphia (2002)
4. Knyazev, A.V.: Preconditioned eigensolvers - an oxymoron? Electron. Trans. Numer. Anal. **7**, 104–123 (1998)
5. Knyazev, A.V.: Toward the optimal eigensolver: locally optimal block preconditioned conjugate gradient method. SIAM J. Sci. Comput. **23**, 517–541 (2001)
6. Saad, Y.: Numerical Methods for Large Eigenvalue Problems: Revised Edition. SIAM (2011)
7. Yamada, S., Imamura, T., Machida, M.: 16.447 TFlops and 159-Billion-dimensional exact-diagonalization for trapped Fermion-Hubbard Model on the Earth Simulator. In: Proceedings of SC 2005 (2005)
8. Yamada, S., Imamura, T., Machida, M.: Communication avoiding Neumann expansion preconditioner for LOBPCG method: convergence property of exact diagonalization method for Hubbard model. In: Proceedings of ParCo 2017 (2017, accepted)
9. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia (1994)
10. Langou, J.: AllReduce algorithms: application to Householder QR factorization. In: Proceedings of the 2007 International Conference on Preconditioning Techniques for Large Sparse Matrix Problems in Scientific and Industrial Applications, pp. 103–106 (2007)
11. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-avoiding paralleland sequential QR factorizations, Technical report, Electrical Engineering and Computer Sciences, University of California Berkeley (2008)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Application of a Preconditioned Chebyshev Basis Communication-Avoiding Conjugate Gradient Method to a Multiphase Thermal-Hydraulic CFD Code

Yasuhiro Idomura[1(✉)], Takuya Ina[1], Akie Mayumi[1], Susumu Yamada[1], and Toshiyuki Imamura[2]

[1] Japan Atomic Energy Agency, Kashiwa, Chiba 227-0871, Japan
idomura.yasuhiro@jaea.go.jp
[2] RIKEN, Kobe, Hyogo 650-0047, Japan

Abstract. A preconditioned Chebyshev basis communication-avoiding conjugate gradient method (P-CBCG) is applied to the pressure Poisson equation in a multiphase thermal-hydraulic CFD code JUPITER, and its computational performance and convergence properties are compared against a preconditioned conjugate gradient (P-CG) method and a preconditioned communication-avoiding conjugate gradient (P-CACG) method on the Oakforest-PACS, which consists of 8,208 KNLs. The P-CBCG method reduces the number of collective communications with keeping the robustness of convergence properties. Compared with the P-CACG method, an order of magnitude larger communication-avoiding steps are enabled by the improved robustness. It is shown that the P-CBCG method is 1.38× and 1.17× faster than the P-CG and P-CACG methods at 2,000 processors, respectively.

1 Introduction

Krylov subspace methods are widely used for solving linear systems given by extreme scale sparse matrices, and thus, their scalability is one of critical issues towards exascale computing. In nuclear engineering, exascale computing is needed for Computational Fluid Dynamics (CFD) simulations of turbulent flows such as multiphase thermal-hydraulic simulations of nuclear reactors and fusion plasma simulations. In these CFD simulations, implicit solvers based on Krylov subspace methods occupy dominant computational costs, and the scalability of such CFD simulations largely depends on the performance of Krylov solvers.

The current Peta-scale machines are characterized by extreme concurrency reaching at ∼100 k computing nodes. In addition to this feature, on future exascale machines, which may be based on many-core processors or accelerators, significant acceleration of computation is expected. In Ref. [1], we optimized stencil computation kernels from CFD simulations on the latest many-core

© The Author(s) 2018
R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 257–273, 2018.
https://doi.org/10.1007/978-3-319-69953-0_15

processors and GPUs, and significant performance gains were achieved. However, the accelerated computation revealed severe bottlenecks of communication.

Krylov solvers involve local halo data communications for stencil computations or sparse matrix vector operations SpMVs, and global data reduction communications for inner product operations in orthogonalization procedures for basis vectors. Although communication overlap techniques [2] may reduce the former latency, it can not be applied to the latter. In order to resolve this issue at mathematics or algorithm levels, in Refs. [3,4], we have introduced communication-avoiding (CA) Krylov methods to a fusion plasma turbulence code GT5D [5] and a multiphase thermal-hydraulic CFD code JUPITER [6].

The implicit solver in the GT5D is well-conditioned, and the communication-avoiding general minimum residual (CA-GMRES) method [7] was stable for large CA-steps $s > 10$. On the other hand, the Poisson solver in the JUPITER is ill-conditioned, and the convergence of the left-preconditioned communication-avoiding conjugate gradient (P-CACG) method [7] was limited for $s \leq 3$. Even with $s = 3$, the strong scaling of the JUPITER on the K-computer [8] was dramatically improved by reducing the number of global data reduction communications to $1/s$. However, for practical use, it is difficult to operate CA Krylov solvers at the upper limit of CA-steps, because the Poisson operator is time dependent and its condition number may increase in time. Therefore, we need to use more robust CA Krylov methods at CA-steps well below the upper limit, beyond which they become numerically unstable. In order to resolve this issue, in this work, we introduce the preconditioned Chebyshev basis communication-avoiding conjugate gradient (P-CBCG) method to the JUPITER, and examine its robustness and computational performance on the Oakforest-PACS, which consists of 8,208 KNLs.

The reminder of this paper is organized as follows. Related works are reviewed in Sect. 2. In Sect. 3, we explain CA Krylov subspace methods used in this work. In Sect. 4, we discuss numerical properties and kernel performances of CA Krylov solvers. In Sect. 5, we present the convergence property of CA Krylov methods and the computational performances of CA Krylov solvers on the JAEA ICEX and the Oakforest-PACS. Finally, a summary is given in Sect. 6.

2 Related Works

The CACG method is based on the so-called s-step CG method, in which the data dependency between SpMV and inner product operations in the standard CG method is removed. Van Rosendale [9] first developed a s-step version of the CG method. Chronopoulos and Gear [10] called their own variant of the CG method as the s-step CG method. However, the above works did not change SpMV operations for generating the s-step basis. Toledo optimized the computation of the s-step basis in the s-step CG method [11], in which the number of words transferred between levels of the memory hierarchy is reduced. The CACG method by Hoemmen [7] reduced communications between levels of the memory hierarchy and between processors by a matrix power kernel (MPK) [12].

Carson [13] showed the performance of the CACG method on the Hopper supercomputer using a simple Poisson model problem.

CA-preconditioning is based on sparse approximate inverses with the same sparsity pattern as the matrix A, or block Jacobi (BJ) and polynomial preconditioners [9,11,14]. For instance, in BJ preconditioning, each processor independently solves its local problem. However, when the local preconditioner has data dependency over the whole local problem as in ILU factorization, it is difficult to construct a MPK without additional communications, because each local SpMV requires preconditioned input vector elements from neighboring processors. To avoid the additional communications, Yamazaki et al. [15] proposed an underlap approach, in which each subdomain is divided into an inner part and the remaining surface part, and preconditioning for the latter is approximated by point Jacobi preconditioning. However, in our previous work [3], it was shown that for ill-conditioned problems given by the JUPITER, the underlap approach leads to significant convergence degradation, and a hybrid CA approach, in which SpMVs and BJ preconditioning are unchanged and CA is applied only to inner product operations, was proposed.

In most of performance studies [4,13,15], CA Krylov methods were applied to well-conditioned problems, where CA-steps are extended for $s > 10$. However, in Ref. [3], it was shown that for ill-conditioned problems given by the JUPITER, the P-CACG method is numerically stable only within a few CA-steps even with the original BJ preconditioning. This issue is attributed to the monomial basis vectors, which are aligned to the eigenvector with the maximum eigenvalue as s increases, and the other eigen-components become relatively smaller and are hidden by the round-off errors. This violates the linear independency of the monomial basis vectors, and makes them ill-conditioned, when each basis vectors are not orthogonalized after creating it. To resolve this issue, Hoemmen [7] proposed to use the Newton basis vectors and the Chebyshev basis vectors. Suda et al. [16] proposed the P-CBCG method, which was tested with point Jacobi preconditioning on the K-computer [17]. In this work, we apply the P-CBCG method with BJ preconditioning to the JUPITER, compare its convergence property and numerical stability against the P-CACG method, and demonstrate its computational performance on the Oakforest-PACS.

3 Krylov Solvers in JUPITER Code

3.1 Code Overview

In the JUPITER code [6], thermal-hydraulics of the molten material in nuclear reactors is described by the equations of continuity, Navier-Stokes, and energy, assuming Newtonian and incompressible viscous fluids. The dynamics of gas, liquid, and solid phases of multiple components consisting of fuel pellets, fuel cladding, the channel box, the absorber, reactor internal components, and the atmosphere are described by an advection equation of the volume of fluid (VOF) function. The main computational cost (∼90%) comes from computation of the pressure Poisson equation, because the Poisson operator given by the density

has extreme contrast $\sim 10^7$ between gas and solid phases, and is ill-conditioned. The Poisson equation is discretized by the second order accurate centered finite difference scheme (7 stencils) in the Cartesian grid system (x, y, z). The linear system of the pressure Poisson equation, which is a symmetric block diagonal sparse matrix, is solved using Krylov subspace methods explained in the following subsections. These Krylov solvers use the compressed diagonal storage (CDS) format, which enables highly efficient direct memory access for the block diagonal sparse matrix than the compressed sparse row (CSR) format, which is commonly used in many matrix libraries, and are parallelized using a MPI+OpenMP hybrid parallelization model, in which MPI is used for coarse 3D domain decomposition in (x, y, z) and fine 1D domain decomposition in z is applied to each domain via OpenMP. BJ preconditioning is applied to each fine subdomain so that it is computed in thread parallel.

3.2 Preconditioned Conjugate Gradient (P-CG) Method

In the original version of the JUPITER, the pressure Poisson equation was computed using the P-CG method [18] with BJ preconditioning, in which ILU factorization [18] is applied to each block. In the P-CG method in Algorithm 1, a single iteration consists of SpMV, BJ preconditioning, two inner product operations, and three vector operations (AXPYs). Here, the SpMV requires a local halo data communication per iteration, and the inner product operations need two global data reduction communications (All_reduce) per iteration. One All_reduce at line 4 transfers two elements including the norm of residual vector, while the other All_reduce at line 8 sends one element.

Algorithm 1. Preconditioned Conjugate Gradient (P-CG) method

Input: $Ax = b$, Initial guess x_1
Output: Approximate solution x_i
1: $r_1 := b - Ax_1, z_1 = M^{-1}r_1, p_1 := z_1$
2: **for** $j = 1, 2, ...$ until convergence **do**
3: Compute $w := Ap_j$
4: $\alpha_j := \langle r_j, z_j \rangle / \langle w, p_j \rangle$
5: $x_{j+1} := x_j + \alpha_j p_j$
6: $r_{j+1} := r_j - \alpha_j w$
7: $z_{j+1} := M^{-1} r_{j+1}$
8: $\beta_j := \langle r_{j+1}, z_{j+1} \rangle / \langle r_j, z_j \rangle$
9: $p_{j+1} := z_{j+1} + \beta_j p_j$
10: **end for**

3.3 Preconditioned Communication-Avoiding Conjugate Gradient (P-CACG) Method

The P-CACG method in Algorithm 2 [7] is based on a three term recurrence variant of CG (CG3) method [18]. The CG3 method is decomposed into the outer loop and the inner s-step loop, and the algorithm is modified so that the latter is processed without any communication. At the k-th outer loop, firstly, the

Algorithm 2. Preconditioned Communication Avoiding CG (P-CACG) method

Input: $Ax = b$, Initial guess x_1
Output: Approximate solution x_i

1: $z_0 := 0, z_1 := b - Ax_1$
2: $q_0 := 0, q_1 := M^{-1}z_1$
3: **for** $k = 0, 1, 2, ...$ until convergence **do**
4: $v_{sk+1} := z_{sk+1}$
5: Compute $\underline{V}_k$ $(v_{sk+1}, M^{-1}Av_{sk+1}, ..., (M^{-1}A)^s v_{sk+1})$
6: Compute $\underline{W}_k$ $(M^{-1}\underline{W}_k = \underline{V}_k)$
7: $G_{k,k-1} := \underline{V}_k^* Z_{k-1}, G_{kk} := \underline{V}_k^* \underline{W}_k$
8: $G_k = \begin{pmatrix} D_{k-1} & G_{k,k-1}^* \\ G_{k,k-1} & G_{kk} \end{pmatrix}$
9: **for** $j = 1$ to s **do**
10: Compute d_{sk+j} that satisfies $Aq_{sk+j} = [Z_{k-1}, \underline{W}_k]d_{sk+j}$ and $M^{-1}Aq_{sk+j} = [Q_{k-1}, \underline{V}_k]d_{sk+j}$
11: Compute g_{sk+j} that satisfies $z_{sk+j} = [Z_{k-1}, \underline{W}_k]g_{sk+j}$ and $q_{sk+j} = [Q_{k-1}, \underline{V}_k]g_{sk+j}$
12: $\mu_{sk+j} := g_{sk+j}^* G_k g_{sk+j}$
13: $\nu_{sk+j} := g_{sk+j}^* G_k d_{sk+j}$
14: $\gamma_{sk+j} := \mu_{sk+j}/\nu_{sk+j}$
15: **if** $sk + j = 1$ **then**
16: $\rho_{sk+j} := 1$
17: **else**
18: $\rho_{sk+j} := (1 - \frac{\gamma_{sk+j}}{\gamma_{sk+j-1}} \cdot \frac{\mu_{sk+j}}{\mu_{sk+j-1}} \cdot \frac{1}{\rho_{sk+j-1}})^{-1}$
19: **end if**
20: $u_{sk+j} := [Q_{k-1}, \underline{V}_k]d_{sk+j}$
21: $y_{sk+j} := [Z_{k-1}, \underline{W}_k]d_{sk+j}$
22: $x_{sk+j+1} := \rho_{sk+j}(x_{sk+j} + \gamma_{sk+j}q_{sk+j}) + (1 - \rho_{sk+j})x_{sk+j-1}$
23: $q_{sk+j+1} := \rho_{sk+j}(q_{sk+j} + \gamma_{sk+j}u_{sk+j}) + (1 - \rho_{sk+j})q_{sk+j-1}$
24: $z_{sk+j+1} := \rho_{sk+j}(z_{sk+j} + \gamma_{sk+j}y_{sk+j}) + (1 - \rho_{sk+j})z_{sk+j-1}$
25: **end for**
26: **end for**

s-step monomial basis vectors $\underline{V}_k$ (line 5) and the corresponding preconditioned basis vectors $\underline{W}_k$ (line 6) are generated at once. Secondly, the Gram matrix G_k (line 8) is computed for the inner product operations, which are replaced as $\mu = \langle z, q \rangle = g_{sk+j}^* G_k g_{sk+j}$ (line 12) and $\nu = \langle Aq, q \rangle = g_{sk+j}^* G_k d_{sk+j}$ (line 13). Here, d_{sk+j} (line 10) and g_{sk+j} (line 11) are defined so that they satisfy $Aq_{sk+j} = [Z_{k-1}, \underline{W}_k]d_{sk+j}$ and $z_{sk+j} = [Z_{k-1}, \underline{W}_k]g_{sk+j}$, respectively. Here, x^* denotes its transpose. At the j-th inner loop, these coefficients are computed to obtain the inner products, and then, the solution vector x_{sk+j} (line 22) and two sets of the residual vectors, the unpreconditioned residual vector z_{sk+j} (line 24) and the preconditioned residual vector q_{sk+j} (line 23), are updated by the three term recurrence formulae. In exact arithmetic, s iterations of the P-CG3 method and one outer loop iteration of the P-CACG method are equivalent.

In Ref. [3], we compared convergence properties of the pressure Poisson solver between the original BJ preconditioning and CA preconditioning based on the

underlap approach [15], and significant convergence degradation was observed with the latter preconditioning. In addition, if one uses a MPK with CA preconditioning, s-step halo data is transferred at once. The number of halo data communication directions are significantly increased from 6 (bidirectional in x, y, z) to 26 (including three dimensional diagonal directions), and redundant computations are needed for the halo data. In order to avoid these issues, in this work, we use the BJ preconditioning with a hybrid CA approach [3]. In the P-CACG method, dominant computational costs come from the s-step SpMVs (line 5) and the following BJ preconditioning (line 6) in the outer loop, GEMM operations for constructing the Gram matrix (line 7) in the outer loop, and three vector operations for the three term recurrence formulae (lines 22–24) in the inner loop. Here, the size of GEMM operations depends on s, and thus, their arithmetic intensity is increased with s. If one applies cache blocking optimization, coefficients of the three term recurrence formulae can be reused for s-steps, and the arithmetic intensity of three vector operations is also improved by extending s. The SpMV requires one local halo data communication per inner iteration as in the P-CG method, while the Gram matrix computation needs only one All_reduce for $s(s+1)$ elements of $G_{k,k-1}$ and $(s+2)(s+1)/2$ upper-triangular elements of $G_{k,k}$ per outer iteration. In addition, we compute the norm of residual vector $r_{sk} = b - Ax_{sk+1}$ for the convergence check, which require one All_reduce per outer iteration. Therefore, the P-CACG method requires two All_reduces per outer iteration.

3.4 Preconditioned Chebyshev Basis Communication-Avoiding Conjugate Gradient (P-CBCG) Method

The P-CBCG method [16] is shown in Algorithm 3. Unlike the P-CACG method which is based on the CG3 method, the P-CBCG method computes two term recurrences as in the P-CG method. The inner product operations are performed using the so-called look-unrolling technique [9] instead of a Gram matrix approach in the P-CACG method. A multi-step CG method constructed using the above approach is computed using s-step Chebyshev basis vectors. Here, the preconditioned Chebyshev basis vectors (line 10) are computed using Algorithm 4. In this algorithm, the basis vectors are generated using $T_j(AM^{-1})$, which is the j-th Chebyshev polynomials scaled and shifted within $[\lambda_{\min}, \lambda_{\max}]$ and thus, satisfies $|T_j(AM^{-1})| < 1$, where $\lambda_{\min}$ and $\lambda_{\max}$ are the minimum and maximum eigenvalues of AM^{-1}. In the monomial basis vectors, the generated vectors are aligned to the eigenvector with $\lambda_{\max}$ as s increases, and the other eigencomponents become relatively smaller and are hidden by the round-off errors. This violates the linear independency of the monomial basis vectors, and makes them ill-conditioned, when each basis vector is not orthogonalized after creating it. On the other hand, the minimax property of Chebyshev polynomials helps to keep the basis vectors well-conditioned without orthogonalizing each basis vector. By using this method, one can construct a Krylov subspace, which is mathematically equivalent to that given by the monomial basis vectors, with much less impact of the round-off errors, and s can be extended compared with

Algorithm 3. Preconditioned Chebyshev Basis communication avoiding CG (P-CBCG) method

Input: $Ax = b$, Initial guess x_0
Output: Approximate solution x_i
1: $r_0 := b - Ax_0$
2: Compute S_0 $(T_0(AM^{-1})r_0, T_1(AM^{-1})r_0, ..., T_{s-1}(AM^{-1})r_0\ )$
3: $Q_0 = S_0$
4: **for** $k = 0, 1, 2, ...$ until convergence **do**
5: Compute $Q_k^* A Q_k$
6: Compute $Q_k^* r_{sk}$
7: $a_k := (Q_k^* A Q_k)^{-1} Q_k^* r_{sk}$
8: $x_{s(k+1)} := x_{sk} + Q_k a_k$
9: $r_{s(k+1)} := r_{sk} - A Q_k a_k$
10: Compute S_{k+1} $(T_0(AM^{-1})r_{s(k+1)}, T_1(AM^{-1})r_{s(k+1)}, ..., T_{s-1}(AM^{-1})r_{s(k+1)}\ )$
11: Compute $Q_k^* A S_{k+1}$
12: $B_k := (Q_k^* A Q_k)^{-1} Q_k^* A S_{k+1}$
13: $Q_{k+1} := S_{k+1} - Q_k B_k$
14: $AQ_{k+1} := AS_{k+1} + AQ_k B_k$
15: **end for**

CA Krylov methods based on the monomial basis vectors. In this work, $\lambda_{\max}$ is computed by a power method, while $\lambda_{\min}$ is approximated as zero.

In the P-CBCG method, dominant computational costs come from the preconditioned Chebyshev basis vector generation involving the SpMVs and the BJ preconditioning (line 10) and the remaining matrix computations. The SpMVs at line 10 require s local halo data communications, while the matrix computations at lines 5, 11 need global data reduction communications. Therefore, the P-CBCG method requires two All_reduces per s-steps. One All_reduce at lines 5, 6 transfers $s(s+1)/2$ upper-triangular elements of $Q_k^* A Q_k$, s elements of $Q_k^* r_{sk}$, and one element for the norm of residual vector, while the other All_reduce sends s^2 elements of $Q_k^* A S_{k+1}$.

Algorithm 4. Preconditioned Chebyshev basis

Input: r_{sk}, Approximate minimum/maximum eigenvalues of AM^{-1}, $\lambda_{\min}, \lambda_{\max}$
Output: S_k $(\tilde{z}_0, \tilde{z}_1, ..., \tilde{z}_{s-1})$, AS_k $(A\tilde{z}_0, A\tilde{z}_1, ..., A\tilde{z}_{s-1})$
1: $\eta := 2/(\lambda_{\max} - \lambda_{\min})$
2: $\zeta := (\lambda_{\max} + \lambda_{\min})/(\lambda_{\max} - \lambda_{\min})$
3: $z_0 := r_{sk}$
4: $\tilde{z}_0 := M^{-1} z_0$
5: $z_1 := \eta A \tilde{z}_0 - \zeta z_0$
6: $\tilde{z}_1 := M^{-1} z_1$
7: **for** $j = 2, 3, ..., s$ **do**
8: $z_j := 2\eta A \tilde{z}_{j-1} - 2\zeta z_{j-1} - z_{j-2}$
9: $\tilde{z}_j := M^{-1} z_j$
10: **end for**

4 Kernel Performance Analysis

4.1 Computing Platforms

In this work, we estimate computing performances of the P-CG, P-CACG, and P-CBCG solvers on computing platforms in Table 1. The JAEA ICEX is based on the Xeon E5-2680v3 processor (Haswell) and the dual plane $4 \times$ FDR Infiniband with a hyper cube topology, and the Oakforest-PACS (KNL) consists of the Xeon Phi 7250 processor (Knights Landing) and the Omni Path with a fat tree topology. The compilers used are the Intel Fortran compiler 16.0.1 with the Intel MPI library 5.0 (-O3 -mcmodel= large -qopenmp -fpp -align array64byte -no-prec-div -fma -xHost) and the Intel Fortran compiler 17.0.4 with the Intel MPI library 2017 (-O3 -mcmodel= large -qopenmp -fpp -align array64byte -no-prec-div -fma -axMIC-AVX512) on ICEX and KNL, respectively. In this work, cross-platform comparisons are performed using the same number of processors. Since ICEX is based on a NUMA architecture with two processors per node, we assign two MPI processes per node. As for OpenMP parallelization, we use 12 and 68 threads on ICEX and KNL, respectively. On KNL, we choose 68 threads without hyper threading to avoid performance degradation in MPI communications [4]. Although KNL has hierarchical memory architecture consisting of MCDRAM (16 GByte, B$\sim$480 GByte/s) and DDR4 (96 GByte), we suppress the problem size below 16 GB per node and use only MCDRAM in a flat mode.

In this section, we analyze a single processor performance for the three solvers using a small problem size of $N = 104 \times 104 \times 265$, which corresponds to a typical problem size on a single processor. The achieved performance is compared against the modified roofline model [19], in which a theoretical processing time of each kernel is estimated by the sum of costs for floating point operations and memory access, $t_{RL} = f/F + b/B$. Here, f and b are the numbers of floating point operations and memory access of the kernel. F and B are the

Table 1. Specifications of the JAEA ICEX and the Oakforest-PACS (KNL).

	ICEX	KNL
Number of nodes	2,510	8,208
Total performance [PFlops]	2.41	25.00
Number of cores per node	12×2	68
Peak performance F [GFlops/processor]	480	3046
STREAM bandwidth B [GByte/s/processor]	58	480(MCDRAM)
B/F	0.12	0.16
Cache [MB/cores]	30/12	1/2
Memory per node [GByte]	64	16
Interconnect bandwidth [GByte/s]	13.6	12.5

peak performance and the STREAM memory bandwidth of the processor. The performance ratios of F and B between ICEX and KNL are 6.3× and 8.6×, respectively.

Table 2. Kernel performance analysis (Floating point operation f [Flop/grid], Memory access b [Byte/grid], Arithmetic intensity f/b, Roofline time $t_{RL} = f/F + b/B$ [ns/grid], Peak performance F [Flops], STREAM bandwidth B [Byte/s], Elapse time t [ns/grid], Sustained performance P [GFlops], Roofline ratio $R_{RL} = t_{RL}/t$, and ICEX/KNL ratio R_{ICEX}) in the kernel benchmarks using a single processor of ICEX and KNL.

Algorithm	Kernel	f	b	f/b	ICEX				KNL				
					t_{RL}	t	P	R_{RL}	t_{RL}	t	P	R_{RL}	R_{ICEX}
P-CG	SpMV	15.00	80.00	0.19	1.40	1.94	7.75	0.72	0.17	0.28	52.78	0.60	6.81
	BJ	20.00	128.00	0.16	2.24	2.53	7.90	0.88	0.27	0.46	43.26	0.59	5.48
	Vector	4.00	40.00	0.10	0.69	0.72	5.55	0.96	0.08	0.10	39.74	0.84	7.16
	Total	39.00	248.00	0.16	4.33	5.19	7.52	0.84	0.53	0.85	46.03	0.62	6.12
P-CACG ($s = 3$)	SpMV	13.00	80.00	0.16	1.40	1.83	7.11	0.76	0.17	0.24	54.59	0.72	7.68
	BJ	14.00	120.00	0.12	2.09	2.02	6.94	1.03	0.25	0.44	31.87	0.58	4.59
	Gram	18.67	29.33	0.64	0.54	0.54	34.70	1.01	0.07	0.13	142.86	0.51	4.12
	3-term	41.67	80.00	0.52	1.46	1.42	29.26	1.02	0.18	0.49	84.20	0.36	2.88
	Total	87.33	309.33	0.28	5.49	5.81	15.04	0.94	0.67	1.30	67.03	0.52	4.46
P-CBCG ($s = 12$)	CB	30.58	228.67	0.13	3.98	4.91	6.22	0.81	0.49	0.89	34.46	0.55	5.54
	Matrix	93.17	83.33	1.12	1.62	1.79	51.98	0.91	0.20	0.63	147.14	0.32	2.83
	Total	123.75	312.00	0.40	5.61	6.71	18.45	0.84	0.69	1.52	81.38	0.45	4.41

4.2 P-CG Solver

Computational kernels of the P-CG method consist mainly of SpMV (lines 3, 4), BJ (lines 7, 8), and Vector (AXPYs, lines 5, 6, 9). Here, SpMV and BJ involve the following inner product operations in the same loop. The numbers of floating point operations f and memory access b and the resulting arithmetic intensity f/b of each kernel are summarized in Table 2. Since the pressure Poisson equation is solved using the second order accurate centered finite difference scheme, SpMV and BJ have relatively low arithmetic intensity $f/b < 0.2$. In addition, the remaining AXPYs are memory-intensive kernels with $f/b = 0.1$. Therefore, the high memory bandwidth on KNL has a great impact on the acceleration of the P-CG solver, and the performance ratio between ICEX and KNL exceeds $R_{ICEX} > 6$. Although AXPYs in Vector achieve ideal sustained performances with the performance ratio against the roofline model $R_{RL} \sim 0.9$ both on ICEX and KNL, stencil computations in SpMV and BJ show performance degradation from $R_{RL} \sim 0.8$ on ICEX to $R_{RL} \sim 0.6$ on KNL.

4.3 P-CACG Solver

Computational kernels of the P-CACG method consist mainly of SpMV (lines 5, 6), BJ (lines 5, 6), Gram (lines 7, 8), and 3-term (lines 20–24). The arithmetic intensity of the P-CACG method changes depending on s, because the

arithmetic intensity of Gram and 3-term are proportional to s. Gram scales as $f = 2(s+1)(2s+1)/s$ and $b = 8(3s+2)/s$. 3-term scales as $f = (8s^2+12s+2)/s$ and $b = 48(s+2)/s$, where the s-dependency comes from cache blocking optimization [3]. In Table 2, the kernel performance is summarized at $s = 3$, which is the upper limit of CA-steps in the benchmark problem in Sect. 5. SpMV and BJ in the P-CACG method have lower f and b than the P-CG method, because they do not involve inner product operations. Compared with SpMV and BJ, Gram and 3-term have higher arithmetic intensity $f/b > 0.5$, and thus, an impact of additional computation in the P-CACG method on the total computational cost ($\sim$1.12$\times$ on ICEX) is much lower than the increase of f ($\sim$2.24$\times$) from the P-CG method. These compute-intensive kernels achieve ideal performances with $R_{RL} \sim 1$ on ICEX. However, they show significant performance degradation with $R_{RL} \sim 0.4$ on KNL, and thus, the performance ratio is limited to $R_{ICEX} \sim 4.46$.

4.4 P-CBCG Solver

Computational kernels of the P-CACG method consist mainly of the Chebyshev basis computation CB (line 10), and the remaining matrix computations Matrix. The arithmetic intensity of the P-CBCG method depends on s as in the P-CACG method. CB scales as $f = 2(9s+4)/s$ and $b = 8(4s+35)/s$, and Matrix scales as $f = (7s+2)(s+1)/s$ and $b = 40(2s+1)/s$. In Table 2, the kernel performance is summarized for $s = 12$, which is used in the benchmark problem in Sect. 5. Although f of the P-CBCG method becomes 3.17$\times$ larger than the P-CG method, the increase of computational cost is suppressed to 1.3$\times$ on ICEX, because of the improved arithmetic intensity. However, on KNL, the performance ratio between the P-CG and P-CBCG methods is expanded to 1.79$\times$, because the compute-intensive Matrix kernel shows performance degradation from $R_{RL} \sim 0.9$ on ICEX to $R_{RL} \sim 0.3$ on KNL. As a result, the performance ratio between ICEX and KNL is limited to $R_{ICEX} \sim 4.41$.

5 Numerical Experiment

5.1 Convergence Property

In the present numerical experiment, we compute nonlinear evolutions of molten debris in a single fuel assembly component of nuclear reactor (see Fig. 1). The problem size is chosen as $N = 800 \times 500 \times 3,540 \sim 1.4 \times 10^9$, which was used also in the former works [3,6]. The problem treats multi-phase flows consisting of gas and multi-component liquid and solid of fuel pellets, fuel cladding, the channel box, the absorber, and the other reactor internal components. The convergence property and the computational performance are investigated for fully developed multi-phase flows, which give the largest iteration number. Because of the large problem size and the extreme density contrast of multi-phase flows, the pressure Poisson equation is ill-conditioned, and the P-CG solver is converged

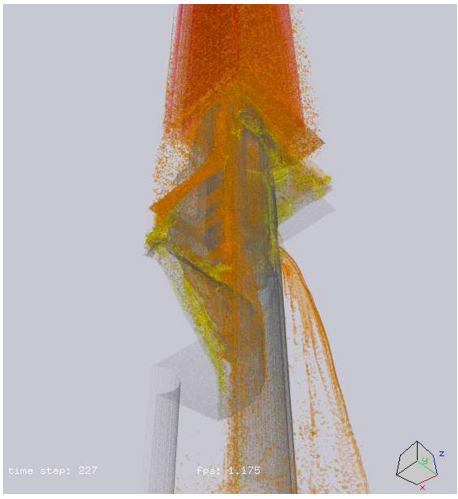


Fig. 1. Visualization of nonlinearly evolved multiphase flows of molten debris in reactor internal components computed by the JUPITER with $N = 800 \times 500 \times 3,540$.

with $\sim 6,000$ iterations (see Fig. 2). Here, the convergence condition is given by the relative residual error of $|b - Ax|/|b| < 10^{-8}$.

The convergence properties of the P-CG, P-CACG, P-CBCG, and P-MBCG solvers are summarized in Fig. 2. Here, the P-MBCG method is a variant of the P-CBCG method, in which the Chebyshev basis vectors at lines 2, 10 are replaced by the monomial basis vectors $S_k(r_{sk}, (AM^{-1})r_{sk}, (AM^{-1})^2 r_{sk}, ..., (AM^{-1})^{s-1} r_{sk})$. Although the P-MBCG method is mathematically similar to the P-CACG method, the former uses the two term recurrence formulae, while the latter is based on the CG3 method or the three term recurrence formulae. In Ref. [20], it was shown that Krylov subspace methods based on three term recurrences give significantly less accurate residuals than those with two term recurrences. In this work, we examine this point for CA Krylov subspace methods. As shown in Ref. [3], the convergence of the P-CACG solver is limited to $s = 3$, while in the P-MBCG solver, the convergence is somewhat extended to $s = 5$. On the other hand, in the P-CBCG method, the convergence property is dramatically extended to $s = 40$. These observations show that the main cause of the convergence degradation is not the three term recurrence formulae, but the ill-conditioned monomial basis vectors. Another important property is in the P-CBCG solver, the convergence property becomes worse gradually above the upper limit of CA-steps, while the P-CACG and P-MBCG solvers breaks down immediately above the upper limit. This property is important for practical use in extreme-scale CFD simulations.

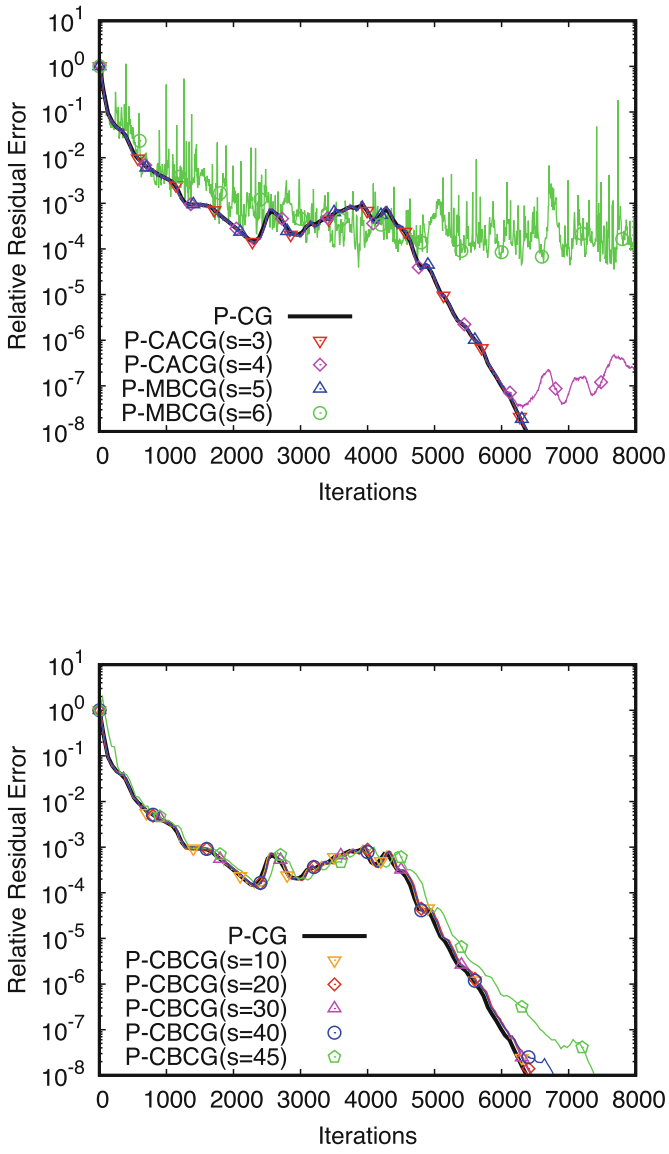


Fig. 2. Comparisons of convergence properties among the P-CG, P-CACG, P-CBCG, and P-MBCG (a variant of the P-CBCG method using the monomial basis vectors) solvers. The relative residual error $|b - Ax|/|b|$ is plotted for the JUPITER with $N = 800 \times 500 \times 3,540$. The computation is performed using 800 processors on ICEX.

5.2 Strong Scaling Test

In the P-CACG solver, we use $s = 3$, which is the upper limit of CA-steps from the viewpoint of numerical stability. On the other hand, the choice of s in the

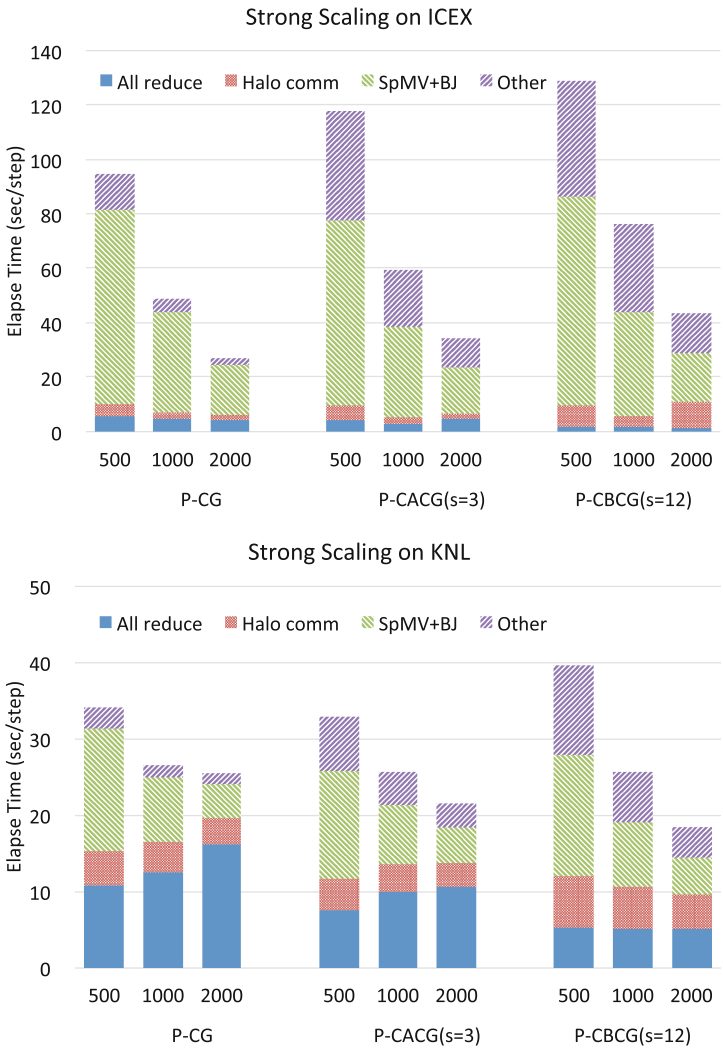


Fig. 3. Strong scaling of the P-CG, P-CACG($s = 3$), and P-CBCG($s = 12$) solvers using 500, 1,000, and 2,000 processors (MPI processes) on ICEX and KNL. The cost distribution in a single time step is shown.

P-CBCG solver is rather flexible, and the optimum s depends on the following factors. Firstly, the number of All_reduce is reduced to $1/s$ compared to the P-CG method. Here, the communication data size scales as $\sim s^2$. Secondly, the numbers of floating point operations and memory access per iteration in Matrix kernel scale as $f \sim s$ and $b \sim const.$, respectively, and the arithmetic intensity of Matrix scales as $f/b \sim s$. Thirdly, cache efficiency of CB is affected by the number of basis vectors. Therefore, the computational performance of each kernel varies depending

on s. Finally, the communication performance is also affected when the data size is changed from a latency bound regime to a bandwidth bound regime. Although a simple performance model was presented in Refs. [13,17], we need more detailed performance models to predict the above complex behaviors. In this work, we chose $s = 12$ from s-scan numerical experiments.

The strong scaling of the P-CG, P-CACG, and P-CBCG solvers are summarized in Fig. 3. In the strong scaling test, we use 500, 1,000, and 2,000 processors on ICEX and KNL, respectively. On ICEX, all Krylov solvers show good strong scaling, because the computation part is dominant in all cases and the communication part is suppressed below $\sim$10 s. Therefore, the P-CACG and P-CBCG solvers are slower than the P-CG solver, because of additional computation in CA Krylov methods. On the other hand, on KNL, the computation part is significantly accelerated ($3.5\times \sim 5.1\times$) and the communication part is comparable or slower ($0.3\times \sim 1.1\times$) compared to ICEX. Here, the cause of slower communication performance on KNL is still under investigation. As a result, the remaining communication part, in particular, All_reduce becomes a severe bottleneck. On KNL, the cost of All_reduce in the P-CG solver increases with the number of processors. This tendency is observed even in the P-CACG solver. However, in the P-CBCG solver, the cost increase of All_reduce is suppressed, and at 2,000 processors, it is reduced to $\sim 1/3$ and $\sim 1/2$ compared to the P-CG and P-CACG solvers, respectively. Because of this CA feature, the best performance on KNL is obtained by the P-CBCG solver, and the P-CBCG solver is $1.38\times$ and $1.17\times$ faster than the P-CG and P-CACG solvers at 2,000 processors, respectively.

It is noted that in Ref. [3], the P-CACG solver on the K-computer showed ideal cost reduction of All_reduce by $1/s$. However, in the present numerical experiment, the cost reduction of All_reduce from the P-CG solver is limited to $\sim 2/3$ and $\sim 1/3$ in the P-CACG and P-CBCG solvers, respectively. These performance ratios are far above the ideal one $1/s$. In order to understand this issue, more detailed performance analysis for All_reduce is needed. Another issue is that the cost of halo data communications increases in the P-CBCG solver, while the number of SpMVs is almost the same as the other solvers. It is confirmed that this cost becomes comparable to that in the P-CACG solver, when the number of CA-steps is reduced to $s = 3$. Therefore, the performance degradation of halo data communications seems to depend on the memory usage, which increases with s. These issues will be addressed in the future work.

6 Summary

In this work, we applied the P-CBCG method to the pressure Poisson equation in the JUPITER. We analyzed numerical properties of the P-CACG and P-CBCG methods in detail, and compared it against the P-CG method, which was used in the original code. The P-CACG and P-CBCG methods reduce data reduction communications to $1/s$, but additional computation is needed for CA procedures. The P-CACG ($s = 3$) and P-CBCG ($s = 12$) methods have $\sim 2\times$ and $\sim 3\times$ larger f, while the increase in b is only $\sim 1.25\times$. Because of the improved arithmetic intensity f/b, the resulting computational costs of the P-CACG and P-CBCG solvers

on a single processor were respectively suppressed to ∼1.1× and ∼1.3× on ICEX, while they were expanded to ∼1.5× and ∼1.8× on KNL.

We tested convergence properties of CA Krylov subspace methods based on the monomial basis vectors (P-CACG, P-MBCG) and the Chebyshev basis vectors (P-CBCG). In the comparison between the P-CACG and P-MBCG methods, which are based on three term recurrences and two term recurrences, the latter showed slightly improved convergence. However, the convergence of both solvers were limited for $s \ll 10$. On the other hand, the convergence of the P-CBCG method was extended to $s \sim 40$, and the robustness of CA Krylov solvers was dramatically improved.

Strong scaling tests of the P-CG, P-CACG ($s = 3$), and P-CBCG ($s = 12$) solvers were performed using 500, 1,000, and 2,000 processors on ICEX and KNL. On ICEX, the computational costs were dominated by the computation part, and all three solvers showed good strong scaling. As the communication part is minor, the P-CG solver was fastest on ICEX. On the other hand, on KNL, the computation part was significantly accelerated, and the remaining communication part, in particular, All_reduce became a severe bottleneck. By reducing the cost of All_reduce, the best performance was achieved by the P-CBCG solver, and the P-CBCG solver is 1.38× and 1.17× faster than the P-CG and P-CACG solvers at 2,000 processors, respectively. As the P-CBCG method satisfies both high computational performance and excellent robustness, it is promising algorithm for extreme scale simulations on future exascale machines with limited network and memory bandwidths.

Acknowledgement. The authors would like to thank Dr. S. Yamashita for providing the JUPITER for the present benchmark, and Dr. T. Kawamura for the visualization image. This work is supported by the MEXT (Grant for Post-K priority issue No.6: Development of Innovative Clean Energy). Computations were performed on the Oakforest-PACS (Univ. Tokyo/Univ. Tsukuba) and the ICEX (JAEA).

References

1. Asahi, Y., et al.: Optimization of fusion Kernels on accelerators with indirect or strided memory access patterns. IEEE Trans. Parallel Distrib. Syst. **28**(7), 1974–1988 (2017)
2. Idomura, Y., et al.: Communication-overlap techniques for improved strong scaling of Gyrokinetic Eulerian code beyond 100k cores on the K-computer. Int. J. High Perform. Comput. Appl. **28**(1), 73–86 (2014)
3. Mayumi, A., et al.: Left-preconditioned communication-avoiding conjugate gradient methods for multiphase CFD simulations on the K computer. In: Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA 2016, Piscataway, NJ, USA, pp. 17–24. IEEE Press (2016)
4. Idomura, Y., Ina, T., Mayumi, A., Yamada, S., Matsumoto, K., Asahi, Y., Imamura, T.: Application of a communication-avoiding generalized minimal residual method to a gyrokinetic five dimensional Eulerian code on many core platforms. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA 2017, New York, NY, USA, pp. 7:1–7:8. ACM (2017)

5. Idomura, Y., et al.: Study of ion turbulent transport and profile formations using global gyrokinetic full-f Vlasov simulation. Nucl. Fusion **49**, 065029 (2009)
6. Yamashita, S., Ina, T., Idomura, Y., Yoshida, H.: A numerical simulation method for molten material behavior in nuclear reactors. Nucl. Eng. Des. **322**(Suppl. C), 301–312 (2017)
7. Hoemmen, M.: Communication-avoiding Krylov subspace methods. Ph.D. thesis, University of California, Berkeley (2010)
8. Fujitsu Global: K computer. http://www.fujitsu.com/global/about/businesspolicy/tech/k/
9. Van Rosendale, J.: Minimizing inner product data dependencies in conjugate gradient iteration. NASA contractor report (1983)
10. Chronopoulos, A., Gear, C.: s-step iterative methods for symmetric linear systems. J. Comput. Appl. Math. **25**(2), 153–168 (1989)
11. Toledo, S.A.: Quantitative performance modeling of scientific computations and creating locality in numerical algorithms. Ph.D. thesis, Massachusetts Institute of Technology (1995)
12. Demmel, J., Hoemmen, M., Mohiyuddin, M., Yelick, K.: Avoiding communication in sparse matrix computations. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–12, April 2008
13. Carson, E.C.: Communication-avoiding Krylov subspace methods in theory and practice. Ph.D. thesis, University of California, Berkeley (2015)
14. Chronopoulos, A., Gear, C.W.: Implementation of preconditioned s-step conjugate gradient methods on a multiprocessor system with memory hierarchy. Technical report, Department of Computer Science, Illinois University, Urbana, USA (1987)
15. Yamazaki, I., Anzt, H., Tomov, S., Hoemmen, M., Dongarra, J.: Improving the performance of CA-GMRES on multicores with multiple GPUs. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 382–391, May 2014
16. Suda, R., Cong, L., Watanabe, D., Kumagai, Y., Fujii, A., Tanaka, T.: Communication-avoiding CG method: new direction of Krylov subspace methods towards exa-scale computing. RIMS Kôkyûroku **1995**, 102–111 (2016)
17. Kumagai, Y., Fujii, A., Tanaka, T., Hirota, Y., Fukaya, T., Imamura, T., Suda, R.: Performance analysis of the Chebyshev basis conjugate gradient method on the K computer. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 74–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_8
18. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2003)
19. Shimokawabe, T., et al.: An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11, November 2010
20. Gutknecht, M.H., Strakos, Z.: Accuracy of two three-term and three two-term recurrences for Krylov space solvers. SIAM J. Matrix Anal. Appl. **22**(1), 213–229 (2000)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Optimization of Hierarchical Matrix Computation on GPU

Satoshi Ohshima[1(✉)], Ichitaro Yamazaki[2], Akihiro Ida[3], and Rio Yokota[4]

[1] Kyushu University, Fukuoka, Japan
ohshima@cc.kyushu-u.ac.jp
[2] University of Tennessee, Knoxville, USA
iyamazak@icl.utk.edu
[3] The University of Tokyo, Tokyo, Japan
ida@cc.u-tokyo.ac.jp
[4] Tokyo Institute of Technology, Tokyo, Japan
rioyokota@gsic.titech.ac.jp

Abstract. The demand for dense matrix computation in large scale and complex simulations is increasing; however, the memory capacity of current computer system is insufficient for such simulations. Hierarchical matrix method ($\mathcal{H}$-matrices) is attracting attention as a computational method that can reduce the memory requirements of dense matrix computations. However, the computation of $\mathcal{H}$-matrices is more complex than that of dense and sparse matrices; thus, accelerating the $\mathcal{H}$-matrices is required. We focus on $\mathcal{H}$-matrix - vector multiplication (HMVM) on a single NVIDIA Tesla P100 GPU. We implement five GPU kernels and compare execution times among various processors (the Broadwell-EP, Skylake-SP, and Knights Landing) by OpenMP. The results show that, although an HMVM kernel can compute many small GEMV kernels, merging such kernels to a single GPU kernel was the most effective implementation. Moreover, the performance of BATCHED BLAS in the MAGMA library was comparable to that of the manually tuned GPU kernel.

1 Introduction

The scale of computer simulations continues to increase as hardware capability advances from post-Peta to Exascale. At such scales, the asymptotic complexity of both computation and memory is a serious bottleneck if they are not (near) linear. In addition, the deep memory hierarchy and heterogeneity of such systems are a challenge for existing algorithms. A fundamental change in the underlying algorithms for scientific computing is required to facilitate exascale simulations, i.e., (near) linear scaling algorithms with high data locality and asynchronicity are required.

In scientific computing, the most common algorithmic components are linear algebra routines, *e.g.*, matrix - vector multiplication, matrix-matrix multiplication, factorization, and eigenvalue problems. The performance of these components has been used as a proxy to measure the performance of large scale systems.

© The Author(s) 2018
R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 274–292, 2018.
https://doi.org/10.1007/978-3-319-69953-0_16

Note that the general usefulness of the high performance LINPACK benchmark for supercomputers has long been disputed, and recent advancements of dense linear algebra methods with near linear complexity could be the final nail in the coffin.

Dense matrices requires $\mathcal{O}(N^2)$ storage and have a multiplication/factorization cost of $\mathcal{O}(N^3)$. Hierarchical low-rank approximation methods, such as $\mathcal{H}$-matrices [1], hierarchical semi-separable matrices [2], hierarchical off-diagonal low-rank matrices [3], and hierarchical interpolative factorization methods [4], reduce this storage requirement to $\mathcal{O}(N \log N)$ and the multiplication/factorization cost to $\mathcal{O}(N \log^q N)$, where, q denotes a positive number. With such methods, there is no point performing large scale dense linear algebra operations directly. Note that, we refer to all hierarchical low-rank approximation methods as $\mathcal{H}$-matrices in this paper for simplicity.

$\mathcal{H}$-matrices subdivide a dense matrix recursively, i.e., off-diagonal block division terminates at a coarse level, whereas diagonal blocks are divided until a constant block size obtained regardless of the problem size. Here, off-diagonal blocks are compressed using low-rank approximation, which is critical to achieving $\mathcal{O}(N \log N)$ storage and $\mathcal{O}(N \log^q N)$ arithmetic complexity. Recently, $\mathcal{H}$-matrices have attracted increasing attention; however, such efforts have a mathematical and algebraic focus. As a result, few parallel implementations of the $\mathcal{H}$-matrix code have been proposed.

In this paper, we focus on a parallel implementation. Specifically, we target matrix - vector multiplications on GPUs. Of the many scientific applications that involve solving large dense matrices, we selected electric field analysis based on boundary integral formulation. Our results demonstrate that orders of magnitude speedup can be obtained by merging many matrix - vector computations into a single GPU kernel and proper implementation of batched BLAS operations in the MAGMA library [5–7].

The remainder of this paper is organized as follows. An overview of the $\mathcal{H}$-matrices and its basic computation are presented in Sect. 2. In Sect. 3, we focus on $\mathcal{H}$-matrix - vector multiplication (HMVM) and propose various single GPU implementations. Performance evaluation results are presented and discussed in Sect. 4, and conclusions and suggestions for future work are given in Sect. 5.

2 Hierarchical Matrix Method ($\mathcal{H}$-matrices)

$\mathcal{H}$-matrices are an approximation technique that can be applied to the dense matrices in boundary integral equations and kernel summation. The $\mathcal{O}(N^2)$ storage requirement $\mathcal{O}(N^3)$ factorization cost of $\mathcal{H}$-matrices can be reduced to $\mathcal{O}(N \log^q N)$. Therefore, $\mathcal{H}$-matrices allow calculations at scales that are otherwise impossible. In the following, we describe the formulation of $\mathcal{H}$-matrices using boundary integral problems as an example.

2.1 Formulation of $\mathcal{H}$-matrices for Boundary Integral Problems

Let H be a Hilbert space of functions in a $(d-1)$-dimensional domain $\Omega \subset \mathbb{R}^d$ and H' be the dual space of H. For $u \in H$, $f \in H'$, and a kernel function of a

convolution operator g: $\mathbb{R}^d \times \Omega \to \mathbb{R}$, we consider following the integral equation:

$$\int_{\Omega} g(x,y)u(y)\mathrm{d}y = f. \tag{1}$$

To calculate (1) numerically, we divide domain Ω into elements $\Omega^h = \{\omega_j : j \in J\}$, where J is an index set. When we use weighted residual methods, the function u is approximated from a d-dimensional subspace $H^h \subset H$. Given a basis $(\varphi_i)_{\in\mathcal{I}}$ of H^h for an index set $\mathcal{I} := \{1, ..., N\}$, the approximant $u^h \in H^h$ to u can be expressed using a coefficient vector $\phi = (\phi_i)_{i\in\mathcal{I}}$ that satisfies $u^h = \sum_{i\in\mathcal{I}} \phi_i\varphi_i$. Note that the supports of the basis $\Omega^h_{\varphi_i} := \mathrm{supp}\ \varphi_i$ are assembled from the sets ω_j. Equation (1) is reduced to the following system of linear equations.

$$A\phi = B. \tag{2}$$

Here, assume that we have two subsets (i.e., clusters) $s, t \in \mathcal{I}$, where the corresponding domains are defined as follows:

$$\Omega^h_s := \bigcup_{i\in s} \mathrm{supp}\,\varphi_i,\ \ \Omega^h_t := \bigcup_{i\in t} \mathrm{supp}\,\varphi_i. \tag{3}$$

A cluster pair (s, t) is 'admissible', if the Euclidian distance between Ω^h_s and Ω^h_t is sufficiently large compared to their diameters:

$$\min\{\mathrm{diam}(\Omega^h_s), \mathrm{diam}(\Omega^h_t)\} \le \eta\ \mathrm{dist}(\Omega^h_s, \Omega^h_t), \tag{4}$$

where η is a positive constant number depending on the kernel function g and the division Ω^h. For the domain corresponding to the admissible cluster pairs $x \in \Omega^h_s$, $y \in \Omega^h_t$, we assume that the kernel function can be approximated at certain accuracy using a degenerate kernel such as

$$g(x,y) \cong \sum_{\nu=1}^{k} g_1^{\nu}(x)g_2^{\nu}(y), \tag{5}$$

where k is a positive number. Such kernel functions are employed in various scientific applications, e.g., electric field analysis, mechanical analysis, and earthquake cycle simulations. The kernel functions in such applications can be written as follows:

$$g(x,y) \in \mathrm{span}(\{|x-y|^{-p}, p > 0\}). \tag{6}$$

When we consider static electric field analysis as a practical example, the kernel function is given by

$$g(x,y) = \frac{1}{4\pi\epsilon}|x-y|^{-1}. \tag{7}$$

Here, ϵ denotes the electric permittivity. Figure 1 shows the calculation result when a surface charge method is used to calculate the electrical charge on the surface of the conductors. We divided the surface of the conductor into triangular elements and used step functions as the base function φ_i of the BEM.

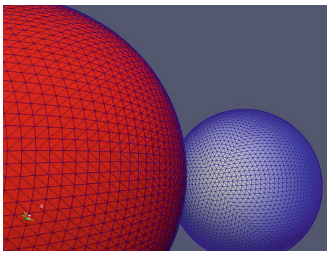


Fig. 1. Calculated surface charge density and triangular elements dividing the conductor surface.

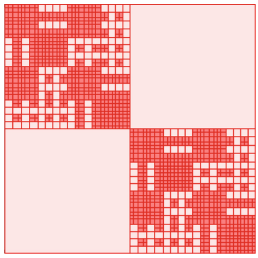


Fig. 2. Partition structure of $\mathcal{H}$-matrix $\tilde{A}^K_{\mathcal{H}}$ for the two-sphere model in Fig. 1. Dark and light red blocks represent dense sub-matrices and low-rank sub-matrices, respectively. (Color figure online)

An $\mathcal{H}$-matrix $\tilde{A}^K_{\mathcal{H}}$, the approximation of A in (2), is characterized by a partition $\mathcal{H}$ of $N \times N$ with blocks $h = s_h \times t_h \in \mathcal{H}$ and block-wise rank K (Fig. 2). Note that most off-diagonal blocks in $\tilde{A}^K_{\mathcal{H}}$ have a low-rank, and the diagonal blocks remain dense. A low-rank matrix $\tilde{A}^K_{\mathcal{H}}|^h$, which approximates a sub-matrix $A_{\mathcal{H}}|^h$ of the original matrix corresponding to block h, is expressed as

$$\tilde{A}^K_{\mathcal{H}}|^h := \sum_{\nu=1}^{k_h} v^\nu (w^\nu)^T, \tag{8}$$

where $v^\nu \in \mathbb{R}^{s_h}$, $w^\nu \in \mathbb{R}^{t_h}$, and $k_h \leq K$. Typically, the upper limit K of the ranks of sub-matrices is set such that $||A - \tilde{A}^K_{\mathcal{H}}||_{\mathcal{F}} \leq \epsilon$ for a given tolerance ϵ.

For $x, b \in \mathbb{R}^{\mathcal{I}}$, we consider the following equation:

$$\tilde{A}^K_{\mathcal{H}} x = b. \tag{9}$$

To solve (9), we use a Krylov subspace method, such as the BiCGSTAB method. The HACApK library [8] and ppOpen-APPL/BEM [9,10] implement these computations in parallel and distributed computer environments using the MPI and OpenMP.

2.2 BiCGSTAB Method for the Hierarchical Matrix

We select BiCGSTAB method to solve (2) because the coefficient matrices are not positive definite. Similar to the BiCGSTAB method for a dense matrix, most of the execution time of the BiCGSTAB method for an $\mathcal{H}$-matrix is spent in HMVM. Low-rank sub-matrix - vector multiplication involves two dense matrix - vector multiplications; therefore, HMVM results in many dense matrix - vector multiplications (Fig. 3). Figure 4 shows the pseudo code of the HMVM kernel in ppOpen-APPL/BEM which is optimized for multi-core CPUs. The original code was implemented in Fortran; however, to develop a GPU version of HMVM, we have developed a C version that is nearly the same as the algorithm in the original code. Hereafter, we refer to this kernel as the *OMP kernel*.

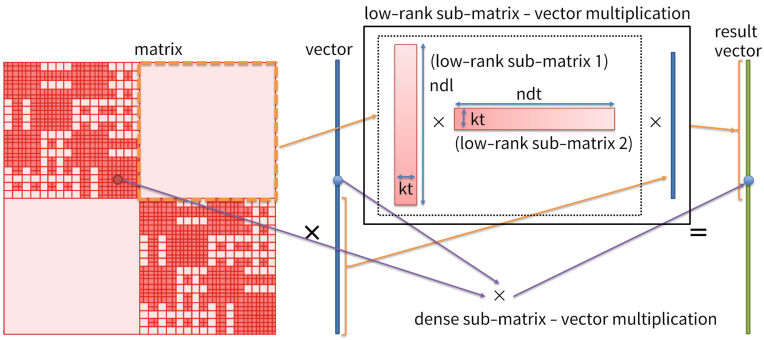


Fig. 3. HMVM calculation

```
#pragma omp for
for(ip=0; ip<number_of_leaves; ip++){
  if(leaf[ip]==low-rank_sub-matrix){
    tmpvec2 <= sub-matrix_2 * vector;
    tmpvec  <= sub-matrix_1 * tmpvec2;
  }
  if(leaf[ip]==small_dense_sub-matrix)
    tmpvec <= sub-matrix * vector;
  }
  for(...;...;...){
#pragma omp atomic
    result <= result + tmpvec;
  }
}
```

low-rank sub-matrix - vector multiplication

small dense sub-matrix - vector multiplication

Fig. 4. Pseudo code of the HMVM kernel (*OMP kernel*); the range of the loops in each sub-matrix - vector multiplication depends on the target leaves.

HMVM comprises many low-rank sub-matrix - vector multiplications for off-diagonal blocks and dense sub-matrix - vector multiplications for diagonal blocks. These matrix - vector calculations correspond to the leaves of a tree structure; thus, we refer to both low-rank sub-matrix - vector multiplication and dense sub-matrix - vector multiplication as leaves. This parallel implementation requires atomic addition because multiple leaves may have partial values of the same index of the result vector. Although it can be eliminated using atomic operations in each matrix - vector multiplications, the *OMP kernel* merges partial results after sub-matrix - vector multiplication because atomic operations in sub-matrix - vector multiplication incur additional computation cost and cannot obtain better performance than previous implementations. Note that the length of the parallel loop is sufficient for current parallel processors because our target matrices have greater than thousands of leaves.

3 $\mathcal{H}$-matrix Computation on GPU

The BiCGSTAB method employs basic matrix and vector operations. Here, HMVM is a dominant component in terms of the time to solution. Therefore, we

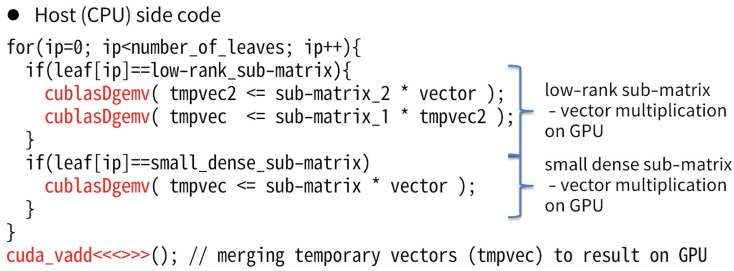


Fig. 5. Pseudo code of the HMVM kernel with CUBLAS (*CUBLAS kernel*); red text indicates functions executed on the GPU. (Color figure online)

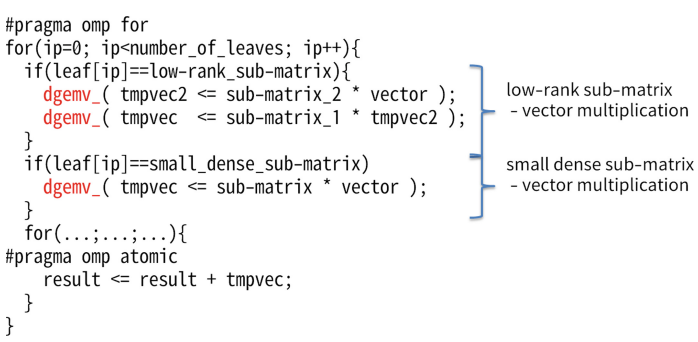


Fig. 6. Pseudo code of the HMVM kernel with MKL (*MKL kernel*); red text indicates MKL functions. (Color figure online)

consider a GPU implementation of HMVM on an NVIDIA Tesla P100 (Pascal architecture) GPU [11].

3.1 BLAS GEMV

As discussed in Sect. 2, HMVM consists of many dense sub-matrix - vector multiplications. A dense matrix - vector multiplication can be replaced by the well-known general matrix vector product calculation (GEMV) in BLAS, and this calculation is provided by some BLAS libraries for NVIDIA GPUs, e.g., the CUBLAS [12] and MAGMA libraries. Therefore, using these BLAS libraries, we can implement HMVM relatively easily. Here, we use CUBLAS for GPUs. In addition, to compare performance, we also implement HMVM using the Math Kernel Library (MKL) for CPUs. Hereafter, we refer to these kernels as the *CUBLAS* and *MKL kernels*. Figures 5 and 6 show the pseudo code of an HMVM kernel using the *CUBLAS* and *MKL kernels*, respectively.

3.2 Simple GEMV Kernels

BLAS libraries are useful; however, they cannot always achieve optimal performance. Generally, such libraries perform optimally for large matrix calculations.

- Host (CPU) side code

```
for(ip=0; ip<number_of_leaves; ip++){
  if(leaf[ip]==low-rank_sub-matrix){
    myDgemv1<<<g,b>>>(...); // tmpvec2 <= sub-matrix_2 * vector
    myDgemv2<<<g,b>>>(...); // tmpvec  <= sub-matrix_1 * tmpvec2
  }
  if(leaf[ip]==small_dense_sub-matrix)
    myDgemv2<<<g,b>>>(...); // tmpvec  <= sub-matrix * vector
  }
}
cuda_vadd<<<>>>(); // merging temporary vectors (tmpvec) to result on GPU
```

low-rank sub-matrix - vector multiplication on GPU

small dense sub-matrix - vector multiplication on GPU

- Device (GPU) side code

```
__global__ void myDgemv1(...){
  int gid = blockIdx.x,  glen = gridDim.x;
  int tid = threadIdx.x, tlen = blockDim.x;
  double tmp;
  for(il=gid; il<rows; il+=glen){
    for(it=tid; it<cols; it+=tlen){
      tmp += matrix[m+it] * vector[v+it];
    }
    reduction_and_write_tmp_
    to_vector_on_globalmemory
  }
}
```

```
__global__ void myDgemv2(...){
  int gid = blockIdx.x,  glen = gridDim.x;
  int tid = threadIdx.x, tlen = blockDim.x;
  double tmp;
  for(il=gid; il<rows; il+=glen){
    for(it=tid; it<cols; it+=tlen){
      tmp += matrix[m+it] * vector[v+it];
    }
    reduction_and_atomicAdd_tmp_
    to_vector_on_globalmemory
  }
}
```

Fig. 7. Pseudo code of the HMVM kernel with CUDA (*SIMPLE kernel*); the entire GPU kernel calculates a single GEMV, and each thread block calculates one GEMV row.

In contrast, HMVM involves many small GEMV calculations. With GPUs, if the CUBLAS GEMV function is used in HMVM, performance will be low because of the lack of parallelism. Moreover, launching GPU kernels requires significant time. In addition, the *CUBLAS kernel* launches a GEMV kernel for each leaf; thus, the incurred overhead will increase execution time.

To evaluate and reduce this overhead, we implemented two HMVM kernels using CUDA.

The first is a GEMV kernel that performs a single GEMV calculation using the entire GPU, and each thread block calculates one GEMV row. Threads in the thread block multiply the matrix and vector elements and calculate the total value using a reduction operation. The reduction algorithm is based on an optimized example code in the CUDA toolkit, which we refer to as the *SIMPLE kernel*. Figure 7 shows the pseudo code of an HMVM kernel using the *SIMPLE kernel*. The execution form (i.e., the number of thread block and threads per block) is an optimization parameter.

Note that many of the GEMV calculations in the HMVM are small; thus, it is difficult for the *SIMPLE kernel* to obtain sufficient performance. To improve performance, some parts of the GPU should calculate a single GEMV in parallel. Thus, we developed an advanced kernel in which a single GEMV kernel is calculated by one thread block, and each line in a single GEMV is calculated by a single warp. Moreover, to eliminate data transfer between the CPU and GPU, two GEMV calculations in low-rank sub-matrix - vector multiplication are merged to a single GPU kernel, and shared memory is used rather

- Host (CPU) side code

```
for(ip=0; ip<number_of_leaves; ip++){
  if(leaf[ip]==low-rank_sub-matrix){
    myDgemvA<<<1,b,,s[ip]>>>(...); // tmpvec2 <= sub-matrix_2 * vector
  }                                // tmpvec  <= sub-matrix_1 * tmpvec2
  if(leaf[ip]==small_dense_sub-matrix)
    myDgemvB<<<1,b,,s[ip]>>>(...); // tmpvec  <= sub-matrix * vector
  }
}
cudaThreadSynchronize();
cuda_vadd<<<>>>(); // merging temporary vectors (tmpvec) to result on GPU
```

low-rank sub-matrix - vector multiplication on GPU

small dense sub-matrix - vector multiplication on GPU

- Device (GPU) side code

```
__global__ void myDgemvA(...){
  int wid  = threadIdx.x/32; // WARP ID
  int wlen = blockDim.x/32;
  int xid  = threadIdx.x%32, xlen = 32;
  double tmp;
  __shared__ double tmpvec[];
  for(il=wid; il<kt; il+=wlen){
    for(it=xid; it<ndt; it+=xlen){
      tmp += matrix2[m+it] * vector[v+it];
    }
    reduction_and_write_tmp_
    to_tmpvec_on_sharedmemory
  }
  __syncthreads();
  for(it=wid; it<ndl; it+=wlen){
    for(il=xid; il<kt; il+=xlen){
      tmp += matrix1[m+il] * tmpvec[v+il];
    }
    reduction_and_atomicAdd_tmp_
    to_vector_on_globalmemory
  }
}
```

```
__global__ void myDgemvB(...){
  int wid  = threadIdx.x/32; // WARP ID
  int wlen = blockDim.x/32;
  int xid  = threadIdx.x%32, xlen = 32;
  double tmp;
  for(il=wid; il<ndl; il+=wlen){
    for(it=xid; it<ndt; it+=xlen){
      tmp += matrix[m+it] * vector[v+it];
    }
    reduction_and_atomicAdd_tmp_
    to_vector_on_globalmemory
  }
}
```

Fig. 8. Pseudo code of the HMVM kernel with CUDA (*ASYNC kernel*); one thread block calculates one GEMV, each warp in the thread blocks calculates a single line, two GEMV calculations of low-rank sub-matrix - vector multiplication are merged into a single GPU kernel, and multiple GPU kernels are launched asynchronously.

than global memory. Note that we refer to this kernel as the *ASYNC kernel*. Figure 8 shows the pseudo code of an HMVM kernel with the *ASYNC kernel*. Here, the execution form is also an optimization parameter, similar to the *SIMPLE kernel*; however, the number of thread blocks is always one and multiple GPU kernels are launched concurrently using CUDA stream. Moreover, atomic function is used to merge the partial results because the atomic addition operation of the P100 is fast enough and this implementation can make memory management easy.

3.3 All-in-One Kernel

It is well known that launching a GPU kernel requires much more time than launching a function executed on a CPU. In previous HMVM kernels, the number of launched GPU kernels has depended on the number of leaves; therefore, GPU kernels are launched many times, which may degrade performance. To address

- Host (CPU) side code

```
myHMVM<<<g,b>>>(...);
```

- Device (GPU) side code

```
__global__ void myHMVM(...){
  int gid  = blockIdx.x,      glen = gridDim.x;
  int wid  = threadIdx.x/32,  wlen = blockDim.x/32; // wid = WARP ID
  int xid  = threadIdx.x%32,  xlen = 32;
  double tmp;
  __shared__ double tmpvec[];
  for(ip=gid; ip<number_of_leaves; ip+=glen){
    if(leaf[ip]==low-rank_sub-matrix){
      for(il=wid; il<kt; il+=wlen){
        for(it=xid; it<ndt; it+=xlen){
          tmp += matrix2[m+it] * vector[v+it];
        }
        reduction_and_write_tmp_to_tmpvec_on_sharedmemory
      }
      __syncthreads();
      for(it=wid; it<ndl; it+=wlen){
        for(il=xid; il<kt; il+=xlen){
          tmp += matrix1[m+it] * tmpvec[v+it];
        }
        reduction_and_atomicAdd_tmp_to_vector_on_globalmemory
      }
    }
    if(leaf[ip]==small_dense_sub-matrix){
      for(il=wid; il<ndl; il<wlen){
        for(it=xid; it<ndt; it+=xlen){
          tmp += matrix[m+it] * vector[v+it];
        }
        reduction_and_write_tmp_to_vector_on_globalmemory
      }
    }
    __syncthreads();
  }
}
```

low-rank sub-matrix - vector multiplication on GPU

small dense sub-matrix - vector multiplication on GPU

Fig. 9. Pseudo code of the HMVM kernel with CUDA (*A1 kernel*); the entire HMVM calculation is executed by a single GPU kernel.

this issue, we have created a new GPU kernel that calculates all sub-matrix - vector multiplications using a single GPU kernel, which we refer to as the *A1 kernel*.

Figure 9 shows the pseudo code of an HMVM kernel with the *A1 kernel*. In this kernel, each leaf is calculated by a single warp, and the basic algorithm of each leaf is similar to that of the *ASYNC kernel*. Although the loop for the number of leaves is executed on the CPU in the *ASYNC kernel*, this loop is executed on the GPU in the *A1 kernel*. Similar to the *ASYNC kernel*, here, the execution form is an optimization parameter.

3.4 BATCHED BLAS

Similar to HMVM, many small BLAS calculations are required in various applications, such as machine learning, graph analysis, and multi-physics. To

```
void magmablas_dgemv_vbatched (
  magma_trans_t trans, magma_int_t* m, magma_int_t* n, double alpha,
  magmaDouble_ptr dA_array[], magma_int_t* ldda,
  magmaDouble_ptr dx_array[], magma_int_t* incx,
  magmaDouble_ptr dy_array[], magma_int_t* incy,
  magma_int_t batchCount, magma_queue_t queue);
```

Fig. 10. Example interface of BATCHED MAGMA BLAS (magmablas_dgemv_vbatched).

```
● Host (CPU) side code
for(ip=0; ip<number_of_leaves; ip++){
  if(leaf[ip]==low-rank_sub-matrix){
    dA_array[m++] = sub-matrix_2[ip];    dx_array[v++] = tmpvector;
    dA_array[m++] = sub-matrix_1[ip];    dx_array[v++] = tmpvector;
  }                                                                  prepare information about
  if(leaf[ip]==small_dense_sub-matrix)                               low-rank sub-matrix
    dA_array[m++] = sub-matrix[ip];      dx_array[v++] = tmpvector;    - vector multiplication
  }                                                                  prepare information about
}                                                                    small dense sub-matrix
magmablas_dgemv_vbatched_atomic(..., dA_array, ..., dx_array, ..., dy_array, ...); // calc on GPU
                                                                     - vector multiplication
```

Fig. 11. Pseudo code of the HMVM kernel with BATCHED MAGMA BLAS (*BATCHED kernel*).

accelerate many small BLAS calculations, batched BLAS has been proposed by several BLAS library developers. For example, MKL, MAGMA, and CUBLAS provide batched BLAS functions. Although gemm is the main target function of batched BLAS, MAGMA provides batched gemv functions for a GPU [13]. Figure 10 shows one of the interfaces of the batched gemv function in MAGMA.

Note that we implemented an HMVM kernel using the batched gemv function of MAGMA [14]. Figure 11 shows the pseudo code of our HMVM kernel with BATCHED MAGMA BLAS, which we refer to as the *BATCHED kernel*. In this kernel, the calculation information is constructed in the loop of leaves on the CPU, and the GPU calculates the entire HMVM calculation using the magmablas_dgemv_vbatched_atomic function. Note that the magmablas_dgemv_vbatched_atomic function is not the original BATCHED MAGMA function, i.e., it is a function that we modified to use atomic addition to produce the results.

4 Performance Evaluation

4.1 Execution Environment

In this section, we discuss the performance obtained on the Reedbush-H supercomputer system at the Information Technology Center, The University of Tokyo [15]. Here, we used the Intel compiler 16.0.4.258 and CUDA 8.0.44. We used the following main compiler options: `-qopenmp -O3 -xCORE-AVX2 -mkl=sequential` for the Intel compiler (icc and ifort) and `-O3 -gencode`

Table 1. Execution environment.

Processor	Xeon E5-2695 v4	Tesla P100
Architecture	Broadwell-EP (BDW)	Pascal
# cores	18	3584 (64 cores × 56 SMs)
Clock speed	2.1 GHz (upto 3.3 GHz)	1328 MHz (upto 1480 MHz)
Peak performance (DP)	604.8 GFLOPS	5.3 TFLOPS
Memory type & bandwidth (STREAM Triad)	DDR4 65 GB/s	HBM2 550 GB/s
Processor	Xeon Gold 6140	Xeon Phi 7150
Architecture	Skylake-SP (SKX)	Knights Landing (KNL)
# cores	18	68
Clock speed	2.3 GHz (upto 3.7 GHz)	1.4 GHz (upto 1.6 GHz)
Peak performance (DP)	1324.8 GFLOPS	3046.4 GFLOPS
Memory type (STREAM Triad) & bandwidth	DDR4 95 GB/s	MCDRAM 495 GB/s DDR4 85 GB/s

`arch=compute_60, code="sm_60,compute_60"` for CUDA (nvcc). The *MKL kernel* is called at the multi-threaded region; thus, sequential MKL is linked. Note that threaded MKL obtained near by the same performance in all cases. Here, we used MAGMA BLAS 2.2.

Moreover, to compare performance with other current processors, we measured the performance on a Skylake-SP CPU and a Knights Landing processor. The Skylake-SP processor is installed in the ITO supercomputer system (test operation) at Kyushu University [16], and Intel compiler 17.0.4 with `-qopenmp -O3 -xCORE-AVX512 -mkl=sequential` compiler options was used. The Knights Landing processor is installed in the Oakforest-PACS at JCAHPC [17] and Intel compiler 17.0.4 with `-qopenmp -O3 -xMIC-AVX512 -mkl=sequential` compiler options was used.

Table 1 shows the hardware specifications of all target hardware. Note that we focus on the performance of a single socket in this paper. The execution times of the Broadwell-EP (BDW) and Skylake-SP (SKX) were measured using all 18 CPU cores. The cluster mode of Knights Landing (KNL) was the quadrant mode, and the memory mode was flat (i.e., only MCDRAM was used). Note that the KNL execution times were measured using 64 threads with scatter affinity and hyper-threading degrades performance.

4.2 Target Data

The four matrices in Table 2 are the target matrices of this evaluation. These matrices were generated from electric field analysis problems. Here, the 10ts and 100ts matrices were generated from a problem with a single spherical object,

Table 2. Target matrices.

Matrix name	10ts	216h	human_1x1	100ts
Number of lines	10,400	21,600	19,664	101,250
Number of leaves	23,290	50,098	46,618	222,274
Number of approximate matrices pairs	8,430	17,002	16,202	89,534
Number of small dense matrices	14,860	33,096	30,416	132,740
Amount of $\mathcal{H}$-matrix (MByte)	136	295	298	2,050

and the 216h matrix was generated from a problem with two spherical objects. In addition, a human_1x1 matrix was generated from a problem with a single human-shaped object.

The sizes of the low-rank sub-matrices and small dense sub-matrix of each target matrix are shown in Fig. 12, where the two left graphs of each matrix show the size of the low-rank sub-matrices and the right shows the size of the small dense sub-matrix.

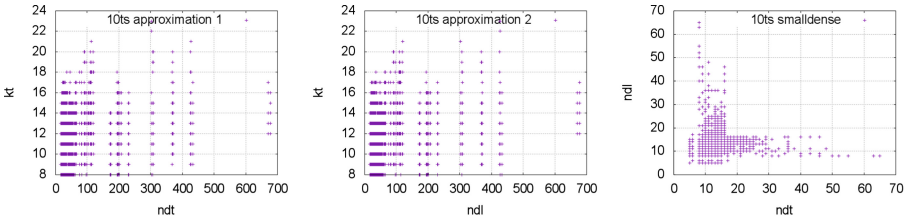
With the 10ts and 100ts matrices, the size of the approximate matrices *ndt* and *ndl* was less than approximately 200 (some were close to 700). Note that all ranks *kt* were very small (the largest was 23). With the small dense matrices, all matrix lengths were less than 100, and many were less than 30.

With the 216h and human_1x1 matrices, the aspect ratio of the small dense matrices was similar to that of the 10ts and 100ts matrices. With the approximate matrices, although *kt* was greater than that of the 10ts and 100ts matrices, the aspect ratio was similar. However, although nearly all *ndt* and *ndl* lengths were less than 1000, a few matrices had *ndt* and *ndl* lengths that were greater than 5000.

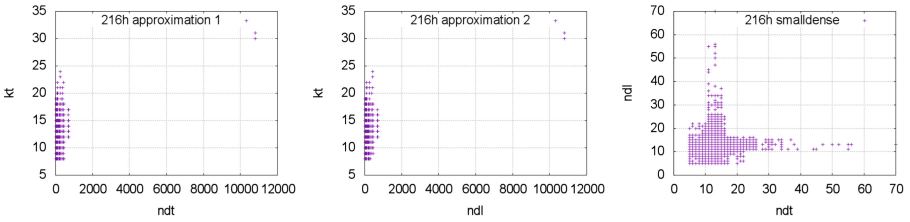
Note that the sizes of these matrices depend on the target matrix. Moreover, the size is controlled by the matrix assembling algorithm and HACApK parameters. The above sizes were generated using current usual HACApK parameter settings. It is expected that optimizing the matrix size will affect HMVM performance, and this will be the focus of future work.

4.3 Performance Evaluation

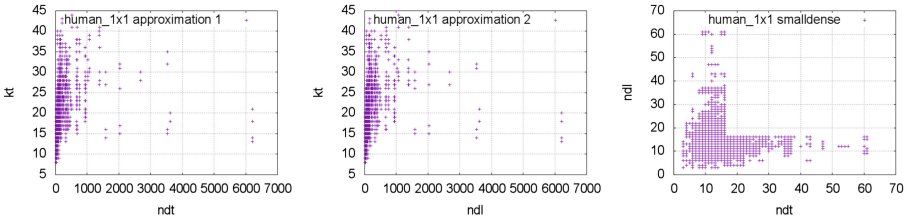
In this subsection, we discuss execution time and performance. Note that the dominant part of the BiCGSTAB method is HMVM; therefore we focus on the execution time of the HMVM. Moreover, the BiCGSTAB method does not modify the matrix data in its own kernel; thus, the each execution time does not include the time required to perform data transfer between the main memory and the GPU in the main iteration of the BiCGSTAB method. Figures 13 and 14 show the execution times for the target matrices. All times are the average execution time of 100 HMVM calculations in 50 BiCGSTAB iterations. As mentioned in the previous section, although the execution form (i.e., grid layout) of the *SIMPLE*, *ASYNC*, and *A1 kernels* are the optimization parameters, only



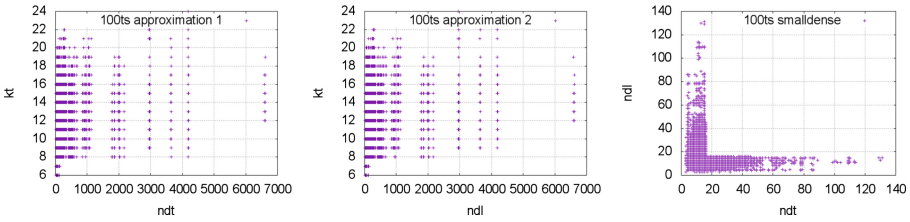
(a) Matrix size of a 10ts matrix



(b) Matrix size of a 216h matrix



(c) Matrix size of a human_1x1 matrix



(d) Matrix size of a 100ts matrix

Fig. 12. Matrix sizes.

the fastest cases are shown and the chosen forms are shown at Table 3. Note that the *ASYNC kernel* launches many GEMV kernels asynchronously with a single thread block. The "#leaves" grids of the *A1 kernel* indicate that the number of thread blocks is equal to the number of leaves, and the outermost GPU kernel loop is eliminated.

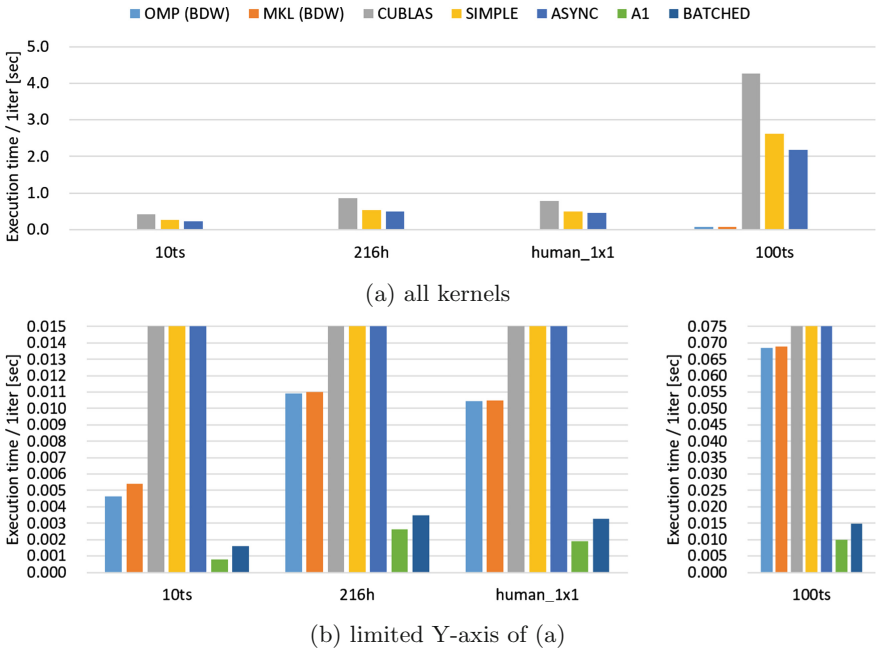
Figure 13(a) shows the execution times of all measurements on the Reedbush-H. As can be seen, the *CUBLAS*, *SIMPLE*, and *ASYNC kernels* were too slow

for a performance comparison with the fast kernels. Figure 13(b) shows graphs with a limited Y-axis from Fig. 13(a). Relative to the CPU execution time, the *OMP* and *MKL kernels* obtained nearly the same performance with all target matrices. Focusing on the GPU execution time, it is clear that the execution times of the *CUBLAS*, *SIMPLE*, and *ASYNC kernels* were much greater than that of the *A1* and *BATCHED kernels*. The major difference between these two groups is the number of launched GPU kernels. As mentioned in the previous section, launching GPU kernels requires more time than executing functions on the CPU and causes long execution times with the three slower kernels. Therefore, although the *ASYNC kernel* improves the performance compared to the *CUBLAS* and *SIMPLE kernels*, its performance is much slower than that of the *A1* and *BATCHED kernels*. On the other hand, the *A1* and *BATCHED kernels* obtained much higher performance than the other kernels. Note that the *A1 kernel* showed better performance than the *BATCHED kernel* because the batched functions in MAGMA BLAS include computations that are unnecessary for HMVM calculation or the execution form is unoptimized.

The execution time ratio of the *OMP kernel* (BDW) to the *A1 kernel* was 17.37% with the 10ts matrix, 24.22% with the 216h matrix, 18.18% with the human_1x1 matrix, and 14.45% with the 100ts matrix, and the execution time ratio of the *OMP kernel* (BDW) to the *BATCHED kernel* was 34.39% with the 10ts matrix, 32.07% with the 216h matrix, 31.43% with the human_1x1 matrix, and 21.67% with the 100ts matrix. Considering that the calculation performance ratio of the GPU to CPU was 11.4% and the memory performance was 10.5%, there might be room to improve the GPU implementation.

Figure 14 shows the execution times of the *A1 kernel*, *BATCHED kernel*, and CPU (i.e., the *OMP* and *MKL kernels*) on the Broadwell-EP (BDW), Skylake-SP (SKX), and Knights Landing (KNL). All times of the KNL are the average execution time of 100 HMVM calculations in 50 BiCGSTAB iterations, but that of the SKX are average execution time of greater than 10 iterations because of the resource limitation of the test operation.

Relative to the performance of SKX, both the *OMP* and *MKL kernels* required nearly 30% less execution time than the *OMP kernel* of the BDW. By considering the performance gap between the BDW and SKX in terms of specification, i.e., the SKX has 45% greater memory bandwidth and more than 200% greater calculation performance than the BDW, it was expected that the SKX would obtain higher performance than 30%. However, HMVM calculation involves various loop length, and it is not a suitable calculation for AVX512; therefore, the obtained performance is not unexpected. On the other hand, there are large differences between the *OMP kernel* and *MKL kernel* of the KNL. However, it is difficult to describe the reason why the performance of the *MKL kernel* was unstable because the MKL implementation is undisclosed. There might be room to improve the KNL implementation. By considering the performance gap between the BDW and KNL in terms of specification, i.e., the KNL has 7.6 times greater memory bandwidth and 5.0 times greater calculation performance than the BDW. The *OMP kernel* of the KNL obtained 34% to 57% better perfor-



(a) all kernels

(b) limited Y-axis of (a)

Fig. 13. Execution times of HMVM on Reedbush-H

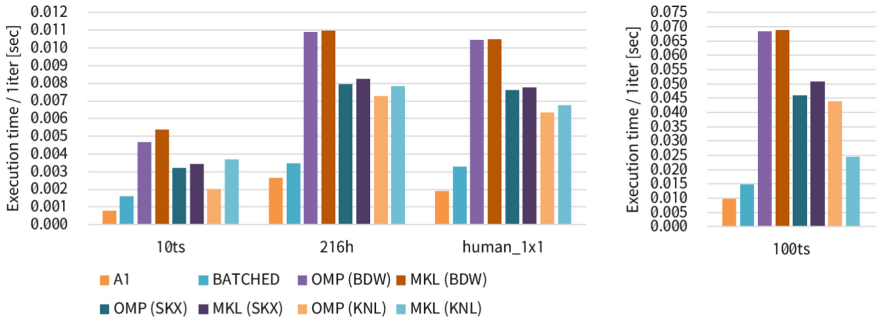


Fig. 14. HMVM execution times.

mance than the *OMP kernel* of the BDW. Similar to the SKX, the KNL has much higher peak performance than the BDW; thus the performance improvement of the KNL is insufficient relative to the performance gap between the BDW and KNL.

Figure 15 shows the entire execution time of the BiCGSTAB method in all target environments. Here, although the iteration count was not exactly the same, only the total computation times are compared. Nearly all vector and matrix calculations of the BiCGSTAB method were executed on the GPU with the *A1 kernel*. Similarly, nearly all vector and matrix calculations of the