Efficient Data Redistribution Algorithms From Irregular to Block Cyclic Data Distribution

Shengguo Li, Hao Jiang, Dezun Dong, Chun Huang, Jie Liu, Xia Liao, and Xuguang Chen

Abstract—In this paper, we propose some efficient data redistribution algorithms for redistributing matrices from 1D or 2D irregular format to block cyclic data distribution (BCDD) format, which can be much faster than the BLACS routine `PXGEMR2D`. These algorithms can be used to combine direct methods with iterative methods. The proposed algorithms divide the communication into two phases: one for processes in the same column and the other for processes in the same row, and the whole data redistribution task is divided into several independent sub-communications. The communication time can be reduced a lot compared with BLACS. Performance results show that our algorithms can be $2\times$–$5\times$ faster than the BLACS routine `PXGEMR2D` when using 4096 processes and the experiments are performed on Tianhe-2A supercomputer.

Index Terms—BLACS, data redistribution, irregular data, scalapack, BCDD, exascale computing

1 Introduction

Data distribution has been an important research topic, largely motivated by the High Performance Fortran (HPF) [1] programming style in which scientific applications are decomposed into phases, see [2], [3], [4], and [5] for more details. As the program proceeds from one computational phase to another, the data access patterns and the number of processes may change, and thus data redistributions may be needed when moving between phases. The HPF standard provides directives for such dynamic data redistribution. Redistribution of data is also critical for the simultaneous exploitation of task and data parallelism in applications [6]. As high performance computing (HPC) moves towards exascale, data redistributions are becoming important again in order to take advantage of the tremendous computational power of compute nodes and heterogeneous processors.

- *Shengguo Li, Hao Jiang, Dezun Dong, Chun Huang, Xia Liao, and Xuguang Chen are with the College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China. E-mail: {nudtlsg, haojiang, dong, chunhuang, liaoxia, chenxuguang}@nudt.edu.cn.*
- *Jie Liu is with the Science and Technology on Parallel and Distributed Processing Laboratory, Laboratory of Software Engineering for Complex Systems, College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China. E-mail: liujie@nudt.edu.cn.*

Manuscript received 22 June 2021; revised 26 Mar. 2022; accepted 3 Apr. 2022. Date of publication 12 Apr. 2022; date of current version 11 July 2022.
This work was supported in part by the National Key R&D Program of China under Grant 2021YFB0300101, in part by the NSFC under Grants 61902411, 62032023, 12002382, 11275269, and 42104078, in part by the 173 Program of China under Grant 2020-JCJQ-ZD-029, and in part by the Open Research Fund from the State Key Laboratory of High Performance Computing of China (HPCL) under Grant 202101-01. The work of Dezun Dong was supported by the Excellent Youth Foundation of Hunan Province under Grant 2021JJ10050. (Corresponding author: Xuguang Chen.)
Recommended for acceptance by P. Bangalore.
Digital Object Identifier no. 10.1109/TPDS.2022.3166484

Data redistribution from a given initial layout to another layout consists in the following four major steps [3]: *index computation*, *communication scheduling*, *message packing and unpacking*, and *data transfer*. Generally, data distribution can be classified into *regular* and *irregular*. The regular array distribution has three types in general: *block*, *cyclic* and *block-cyclic*(c), where c is the block size. The *block-cyclic*(c) is the most general form, which can reduce to the other two forms by choosing different values of c. Fig. 1 shows the three cases when distributing a one-dimensional array with 12 elements to three processes. When $c = 1$, block-cyclic(c) reduces to the cyclic case, and when $c = 4$, it reduces to the block case. The ScaLAPACK library uses the block-cyclic data distribution (BCDD). On the other hand, irregular distributions do not adopt any specific rules or methods to represent the desired data decomposition. Instead, the irregular array distributions are represented with the help of user-defined commands or procedures. For example, HPF2 (High Performance Fortran version 2) supports a generalized block distribution (named GEN_BLOCK) format [7], [8], where unevenly sized chunks of an array are assigned to a set of processors.

In this work, we focus on the irregular data redistribution case, especially that from one dimensional irregular format to two dimensional BCDD format. This work is motivated by our recent work [9], in which a hybrid framework is proposed for the eigenvalue problems by combining direct methods with iterative methods. For the direct and iterative eigenvalue methods, we refer to books [10], [11]. The computation process of this framework is shown in Fig. 2. In the direct method packages such as SLATE [12] and ELPA [13] (the ELPA2 approach), the input dense matrix is stored in 2D BCDD form and transformed to its symmetric banded form through sequences of orthogonal transformations. Then, we apply iterative methods to the banded matrix in [9] instead of using the direct methods which are based on the tridiagonalization approach [14], [15], [16]. For the iterative methods such as in PETSc [17], SLEPc [18] and FEAST [19], the computed eigenvectors are usually stored in 1D block data

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

Global index	1	2	3	4	5	6	7	8	9	10	11	12
Block	P_0				P_1				P_2			
Cyclic	P_0	P_1	P_2	P_0	P_1	P_2	P_0	P_1	P_2	P_0	P_1	P_2
Block-cyclic(3)	P_0			P_1			P_2			P_0		

Fig. 1. Examples of regular data distribution: Block, Cyclic and Block-cyclic.

distribution (BDD) form. To combine direct methods with iterative methods, an efficient data redistribution algorithm is required to redistribute eigenvectors of the banded matrix computed by iterative methods to 2D BCDD form, as shown at the bottom of Fig. 2. Due to the load imbalance problem, different processes may compute different numbers of eigenvectors. Therefore, the eigenvectors are stored irregularly.

Furthermore, due to the increasing complexity of hardware architectures and communication topologies, regular data distributions may be inappropriate for modern applications in terms of both efficiency and scalability. On the petascale and forthcoming exascale computers, hybrid methods are usually used to solve large-scale complicated physical problems [20]. Some packages have already incorporated some hybrid algorithms. For example, SLEPc-3.14.2 has an interface to ScaLAPACK, and it uses the BLACS routines to redistribute matrices. The eigenvalue package ELSI [21] implements some data redistribution algorithms to make the data structures of different phases consistent.

In this paper, we focus on the following cases of data redistribution: 1) redistribute data from 1D irregular to 2D BCDD form; 2) redistribute data from 1D irregular to 1D regular; 3) redistribute data from 2D regular to another 2D regular. The BLACS library [22], [23] provides a set of routines to perform array redistributions. We mainly compare our algorithms with those in BLACS. Performance results show that our algorithms are especially suitable for the general redistributions from block-cyclic(s) to block-cyclic(t) when there is no particular relation between s and t. For this general case, our algorithm can be more than two times faster than the BLACS routine `PXGEMR2D`[1]. For the case that $s = k \cdot t$ or $t = k \cdot s$ and k is an integer, some efficient algorithms have been proposed in [3], [5], [24]. These algorithms have some constraints on the dimension of the array, as will be further discussed in the next subsection. In this work, we focus on the general case.

In summary, the main contributions of this work are

1) A two-step data redistribution algorithm is proposed for redistributing matrices from 1D irregular or regular to 2D BCDD form. The central idea is to divide the communication into two phases: one for processes in the same column and the other for processes in the same row. The communication complexity analysis in section 3 shows that our approach can reduce the number of communications when comparing with the one-step data redistribution algorithm in BLACS in the 1D regular case.
2) For the data redistribution from 1D irregular to 1D regular, a dynamic scheduling based on the maximum cardinality matching (MCM) approach is exploited, which can be much faster than the caterpillar algorithm.

1. The letter `X` stands for the data type: real single (S), double (D), complex single (C), or complex double (Z).

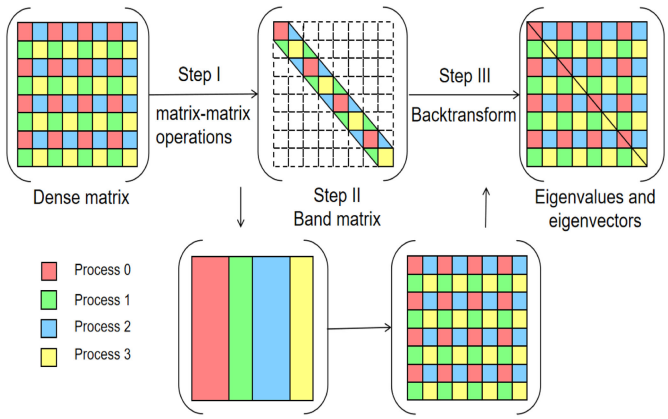


Fig. 2. The process of a hybrid eigenvalue solver proposed in [9]. The eigenvectors of a symmetric banded matrix should be redistributed from (irregular) 1D BDD to 2D BCDD form.

3) For the general redistribution from block-cyclic(s) to block-cyclic(t), our two-step algorithm can be usually more than two times faster than the BLACS routine `PXGEMR2D`.

1.1 Background and Related Work

In this subsection, we review some closely related works, and some are also building blocks of this work.

Ramaswamy and Banerjee [25] use a line segment approach called PITFALLS to represent the block-cyclic(r) distribution. The array elements that map to a processor are represented as a set of stride line segments. For every pair of processors, the array elements whose indices are in the intersection of the respective line segments are exchanged. This approach can handle irregular or regular data distributions as well as multidimensional arrays. However, PITFALLS does not consider communication scheduling. Our work also uses the line segment approach to perform the index computation, but we further consider the communication scheduling.

Walker and Otto were the first to consider scheduling the communications induced by data redistribution [5], and provided a synchronous scheduling to change the array distribution from block-cyclic(r) on a P-processor linear grid to block-cyclic(kr) on a grid of the same size. Desprez, Dongarra, *et al.* [2] extended this approach to a more general case, that is, moving from a block-cyclic(r) distribution on a P-processor grid to a block-cyclic(s) distribution on a Q-processor grid. By constructing the $|V| \times |E|$ incidence matrix A of graph $G = (V, E)$, they turned the scheduling problem to the bipartite matching problem, which is solved by a linear programming algorithm [26, Chap. 30]. They focused on the one-dimensional redistribution problem. We also implement this approach for the 1D irregular to 1D regular data redistribution problem in section 2.2. We use the Micali-Vazirani (MV) algorithm [27] to solve the maximum cardinality matching (MCM) problem instead of using a linear programming algorithm, and their performances should be similar, see [3].

The MV algorithm is an augmented path algorithm with the best theoretical runtime of $O(\sqrt{|V|} \cdot |E|)$ for solving the MCM problem for non-bipartite graphs, where $|V|$ and $|E|$ are the number of vertices and edges, respectively. The MV

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

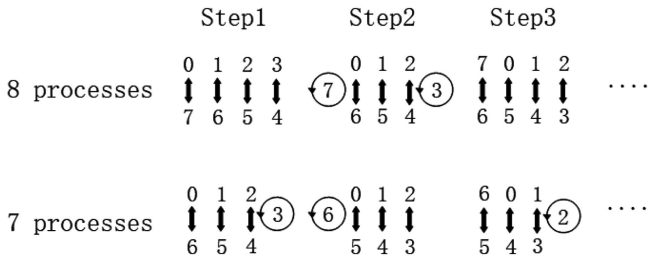


Fig. 3. The caterpillar communication steps with an even (8) and an odd (7) number of processes.

algorithm operates in phases, and each phase finds a maximal set of vertex disjoint shortest length augmenting paths. Each phase constructs a level graph using breadth-first search from unmatched vertices to find alternating paths. Once the level graph expands, the algorithm identifies bridges and performs double depth first search on them. The algorithm ends when the entire graph is searched and there does not exist any augmenting path [28].

As mentioned in [3], the bipartite matching scheme is not suitable for the all-to-all communication case. Based on a generalized circulant matrix formalism, Park, Prasanna and Raghavendra [3] derived an efficient algorithm for data redistribution from block-cyclic(r) on P processors to block-cyclic(Kr) on Q processors. Their algorithm generates a scheduling that minimizes the number of communication steps and eliminates node contention in each communication step, and it also works for the all-to-all communication case. The scheduling computation cost is O$(\max(P, Q))$ flops and it takes less than one percent of the data transfer time. One drawback of this approach is that it requires the array size to be a multiple of $LCM(P, KQ)$, the least common multiple of P and KQ, to avoid padding with dummy data [3].

As described by Prylli and Tourancheau in [29], the data redistribution implemented in BLACS is based on the caterpillar algorithm. The total data exchange is implemented as a succession of steps. At each step, several pairs of communications are performed including send and receive operations. Then, the caterpillar is shifted so that new exchange pairs are formed. At a given step d, each process p_i for $0 \leq i < n_p$ exchanges data with its partner process $p_{(n_p-i-d) \bmod n_p}$. It requires n_p communication steps. Fig. 3 shows the first three communication steps of the caterpillar algorithm when there are eight and seven processes, respectively.

Assume that there are $n_p = p \times q$ processes and they are organized as a 2D mesh. The algorithm in BLACS treats all the n_p processes as a 1D array and applies the caterpillar algorithm to them directly, and therefore it requires n_p communication steps and n_p synchronizations. Our algorithm divides the processes into p process rows and q process columns, and synchronization happens only among processes in the same row or column. The synchronization cost of our algorithms is reduced. Fig. 4 shows the main idea of our 1D to 2D redistribution algorithm: the first communication occurs among processes in the same column and the second communication occurs among processes in the same row. During the row and column communications, we also use the caterpillar algorithm for scheduling but much fewer processes are involved. More details are shown in section 2.1. Our algorithm is a two-step communication method and some extra workspace is required. In this work, we only talk about a 2D process grid but our approach can be generalized to a 3D process grid similarly.

A general redistribution algorithm is proposed for task-based runtime systems in [30], which is said to be the first general redistribution algorithm developed for tile-based applications. Task-based runtime systems have the potential to be able to more easily cope with less-regular data distribution, providing a more balanced computational load. Our current work focuses on the classical 2D BCDD form, and tries to combine ScaLAPACK with iterative eigenvalue solvers. Therefore, the tiled data structure is not the main concern of this work.

The following sections of this paper are organized as follows. Section 2 introduces the three newly proposed algorithms. The communication cost of our algorithms is analyzed in section 3, and it is compared with the BLACS routine `PXGEMR2D`. All the performance results are summarized in section 4. Conclusions are drawn in section 5.

2 Two-Step Data Redistribution

In this section, we introduce the new algorithms proposed in this paper, which are 1) Algorithm from 1D irregular to 2D BCDD; 2) Algorithm from 1D irregular to 1D regular form, where a dynamic scheduling based on the maximum cardinality matching algorithm [27] is exploited; 3) Algorithm from one 2D BCDD to another 2D BCDD, which can be 2× faster than the BLACS routines.

2.1 Irregular 1D to 2D BCDD

We assume there are n_p processes and each process has m_i local vectors of length n for $i = 1, \ldots, n_p$. The processes are organized as a $p \times q$ process grid, so that all the processes in the same process row belong to a communicator

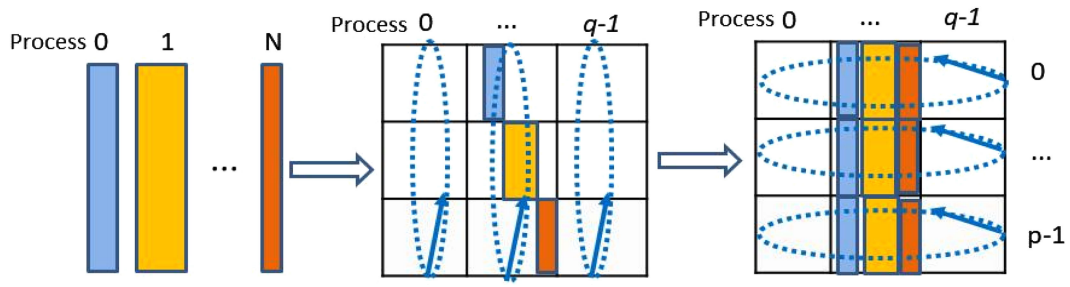


Fig. 4. Data redistribution from 1D irregular to 2D BCDD in two phases. The N processes are transformed to a $p \times q$ 2D grid, then phase 1 performs communications in the process columns and phase 2 performs communications in the process rows. Note that each square block in the middle and right subfigures represents one MPI process.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

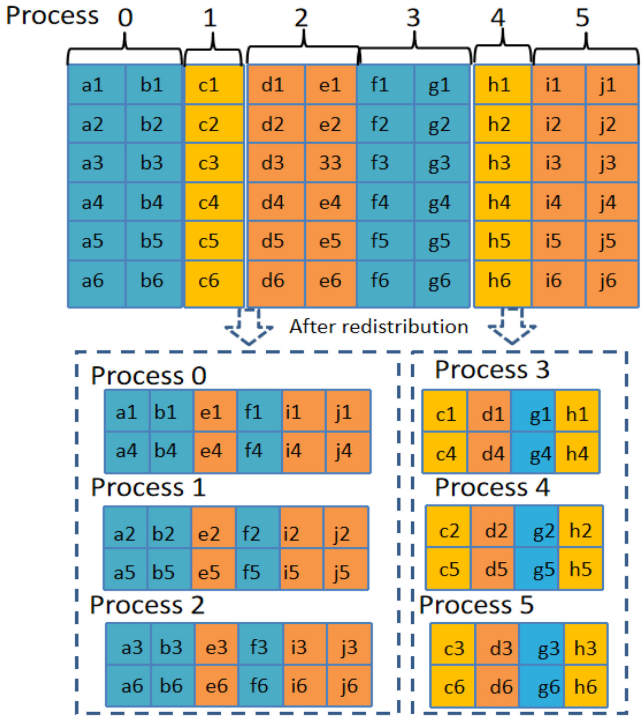


Fig. 5. The matrices before and after data redistribution.

mpi_comm_rows and all the processes in the same process column belong to a communicator *mpi_comm_cols*, in a similar way as the implementation of ELPA [13]. The goal of this subsection is to redistribute all the vectors stored in these n_p processes into a 2D BCDD form with row block size m_b and column block size n_b. Note that the storage of a 2D BCDD matrix is defined by four main parameters: the row block size m_b, the column block size n_b, the row dimension p of the process grid and the column dimension q of the process grid,

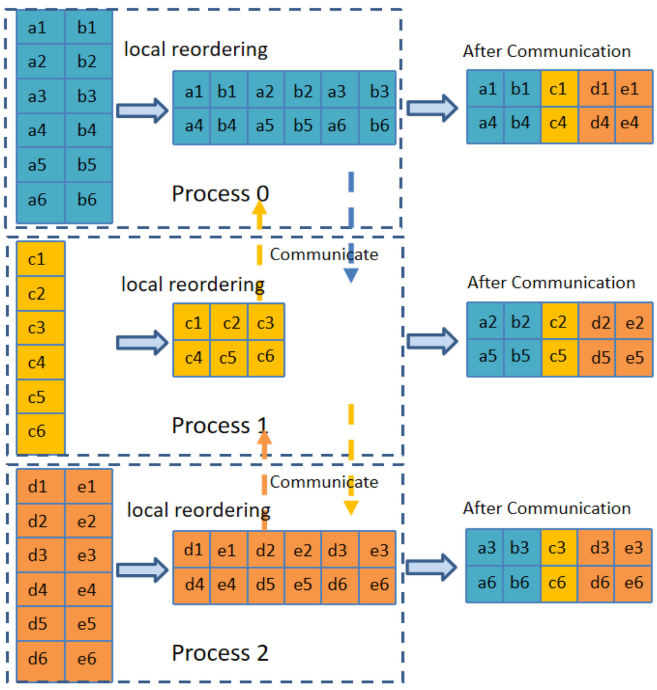


Fig. 6. Phase 1: Communication in the first process column.

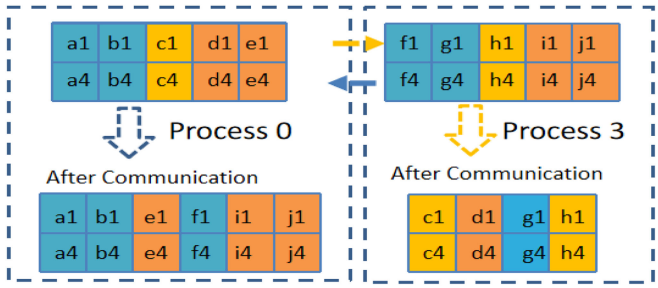


Fig. 7. Phase 2: Communication in the first process row.

plus some other parameters to determine when a submatrix is used, and where it is located in the global matrix.

The details of the proposed data redistribution procedure from irregular 1D to 2D BCDD are included in Algorithm 1, which consists of two phases, see Fig. 4. Phase 1 and 2 correspond to the middle and right subfigures of Fig. 4, respectively. The communications of Phase 1 occur in the same communicator *mpi_comm_cols*, and the communications of Phase 2 occur in the same communicator *mpi_comm_rows*. Note that the colored blocks in the middle of Fig. 4 are the same as the original ones on the left, which are stored locally in each process. The colored blocks in different rows of the right subfigure of Fig. 4 consist of some blocks of the original ones on the left, and the details are shown in Fig. 7.

We assume that there are six processes which are first arranged as a 1×6 grid and then arranged into a 3×2 grid in column major order, and the original and the redistributed matrices are shown at the top and bottom of Fig. 5, respectively. We use the three processes in the first column of Fig. 5 to show the communications of Phase 1 in Fig. 6. We assume that processes $0, 1$ and 2 are in the first process column. To illustrate the process of Phase 1, we assume that the matrix row dimension n satisfies $n = 6n_b$ and different processes have different numbers of local vectors: $m_0 = 2n_b, m_1 = n_b$, and $m_2 = 2n_b$. Each process first reorders its local vectors based on its position in the process grid and stores the data that should be sent to the same process continuously, just as shown in the dashed boxes at the left of Fig. 6. Then, all the processes in the same column perform all-to-all communication by calling MPI routine `MPI_Alltoallv` or the caterpillar algorithm. Finally, all processes (0,1,2) obtain all the data that should be communicated with all the processes (including itself) in the same column.

The communications of Phase 2 are similar and each process groups all local columns together and sends the corresponding columns to the target processes in the same process row, as shown in Fig. 7, where we assume that processes 0 and 3 are in the first process row. Note that after Phase 1 the local rows do not need to be permuted or exchanged with other processes any more, and we only need to permute the local columns or exchange the local columns with other processes. The process of Phase 2 performs a 1D data redistribution in the row direction and can be implemented by using the caterpillar algorithm [4], just as implemented in ScaLAPACK. Another 1D data redistribution algorithm is introduced in the next subsection, which is based on a dynamic scheduling by using the MCM approach [27], and it can be more efficient when the number of processes is large. By reversing the procedure of Algorithm 1, we can similarly

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

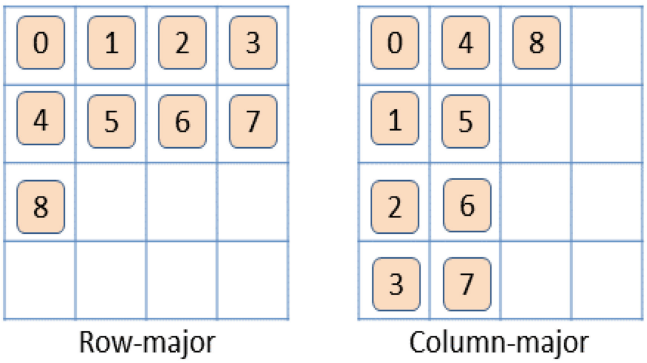


Fig. 8. The original 9 processes are mapped to another 4×4 process grid in row-major or column-major order.

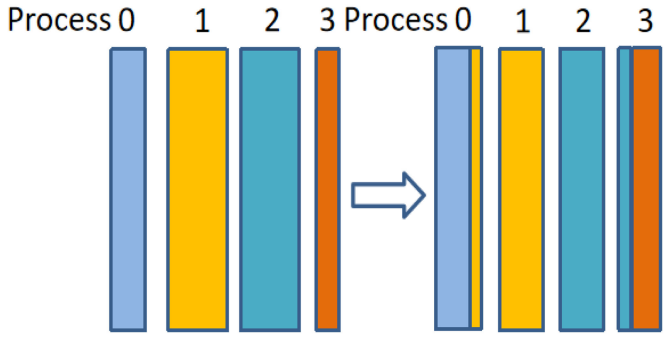


Fig. 9. The vectors among these four processes are redistributed to 1D BDD format.

transform a matrix in 2D BCDD to any irregular 1D BDD form.

Note that the order of these two phases in Algorithm 1 can be exchanged, i.e., first perform row communications and then perform column communications.

Furthermore, we discuss how to redistribute data from one process set to another different process set. The original process set may have some overlapped processes with the target process set. Note that Algorithm 1 assumes that the data redistribution occurs within the same process set. For simplicity, we assume in this paper that these two process sets are completely different, without any overlapped processes.

Algorithm 1. Data Redistribution Algorithm From Irregular 1D BDD to 2D BCDD

Input: The number of processes N_p, organized as a $p \times q$ process grid, each process has a local $n \times m_i$ matrix A_i, the column and row communicators: mpi_comm_cols and mpi_comm_rows, a vector $\mathcal{M}$ stores all m_i, and the total number of vectors $n_{ev} = \sum_{i=1}^{n_p} m_i$.

Output: An $n \times n_{ev}$ matrix B stored in BCDD with row block size m_b and column block size n_b in all $p \times q$ processes.

Phase 1: Communication in the process columns

- Each process reorders A_i into 1D BCDD form in the column direction with row block size m_b, and each segment is stored continuously (see the dashed boxed in Figure 6);
- Each process sends the corresponding segment to the target process in the same column (via MPI routine `MPI_Alltoallv` or the caterpillar algorithm);

Phase 2: Communication in the process rows

- Each process calculates the global indexes of local vectors based on vector $\mathcal{M}$;
- Each process groups all local columns together that are to be sent to the same target process, based on the column block size n_b;
- The processes in the same process row perform all pairwise communications, like the caterpillar algorithm [4];
- Based on the global indexes of received vectors, each process calculates their local indexes by using n_b and stores them into their appropriate positions of B.

To be more precise, we assume that there are n_p original processes which are organized as a $1 \times n_p$ process grid and each process has m_i local vectors of length n, and our goal is to redistribute these vectors to the target $n_q = p \cdot q$ processes in 2D BCDD form with block size n_b, and the target processes are organized as a $p \times q$ process grid. The first step is to map the original n_p processes to the target $p \times q$ process grid. They can be continuously ordered row by row or column by column, as shown in Fig. 8, where $n_p = 9$ original processes are mapped to the target 4×4 process grid in the row-major or column-major ordering. Then, the original processes send their data to their mapped processes, which requires only one *send-receive* communication. When $n_p > n_q$, the mapping is similar but many original processes may be mapped to the same target process. That means some target processes may receive more data than other processes. For the case that there are some overlapped processes, the overlapped processes should be mapped to themselves and thus do not need any communication during the preparation process since they already have the corresponding data. The next step is to execute Algorithm 1 among the target processes, and then the data redistribution between two different process sets is finished.

For the case shown in the right (column-wise) subfigure of Fig. 8, it seems better to perform row communication first and then column communication in Algorithm 1 since fewer processes will be idle. Similarly, for the left (row-wise) subfigure of Fig. 8, it seems better to perform column communication first.

2.2 Irregular 1D to 1D BCDD

In this subsection, we consider redistributing a matrix in 1D irregular block form to 1D BCDD form with a given column block size n_b, which appears in Phase 2 of Algorithm 1. The row block m_b is assumed to be equal to the dimension of matrix n. Fig. 9 shows the case when there are $n_p = 4$ processes. We mention that the caterpillar algorithm [29] can be used to solve this problem. However, the goal of this subsection is to show that the caterpillar algorithm may not be optimal and the maximum cardinality matching approach can be more efficient, especially when the number of processes is large.

The bipartite graph matching based scheduling has been used in [2]. When the original process set is different from the target process set, bipartite graph matching can be used to construct a dynamic scheduling. When the data redistribution occurs within the same process set as Phase 2 of Algorithm 1, we use the maximum cardinality matching (MCM) algorithm since the communication is performed

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

pairwise, both send and receive are performed between pairs of processes. We did not compare the performance of different maximum matching algorithms, and we simply used a code which was downloaded from Github[2] to do the numerical experiments. It implements the Micali-Vazirani (MV) algorithm [27] for maximum cardinality matching.

To determine the columns to be sent, received or stored locally, we use line segments (LS) just as done in [31]. The algorithm in [31] is called the FALLS (Family of line segments) method, and is good for dealing with irregular data for index computation. By combining FALLS with maximum cardinality matching, the distribution of irregular 1D array to regular BCDD can be very efficient. The steps are

1) Each process computes which segments to be sent to which processes based on the start and end indexes of each segment;
2) Each process groups the segments to be sent to the same process together;
3) Each process calls the same maximum cardinality matching algorithm to compute the communication scheduling based on the send-receive relationship, and thus all processes have a copy of the communication scheduling. Thus, no congestion will happen.

For a $1 \times n_p$ 1D process grid, the caterpillar algorithm requires n_p communication steps. The aim of using a maximum cardinality matching approach is to reduce the number of communication steps based on the send-receive relationship. This is especially useful when the ordering of vectors are continuous since each process only needs to communicate with several nearby processes, and therefore the number of communication steps can be much less than n_p, and the communication latency can be reduced a lot. The results are shown in Example 3 in section 4.

2.3 2D BCDD to 2D BCDD

The ScaLAPACK routine `PXGEMR2D` can redistribute a matrix from one 2D BCDD to another BCDD with different block sizes, m_b and n_b. `PXGEMR2D` works on all processes and performs the pairwise communications. We show that `PXGEMR2D` can also be replaced by our two-step communication approach, and our approach can be much more efficient for the general case. More precisely, we assume that a matrix A is stored in 2D BCDD form among $p \times q$ processes with row block size m_b and column block size n_b, and we redistribute it to another BCDD form with a new row block size $\hat{m}_b$ and a new column block size $\hat{n}_b$.

Similarly to Algorithm 1, our improved 2D to 2D data redistribution algorithm consists in two steps. The first step is to perform communications along each process row, and the second step is to perform communications along each process column. In each step, the type of every communication is from 1D regular to another 1D regular, which can be performed by calling `PXGEMR2D` on a much smaller process grid, $1 \times p$ or $q \times 1$. All the row or column communications can be executed simultaneously. The whole procedure is shown in Algorithm 2, which can be implemented by calling some BLACS routines. Note that `PXGEMR2D` originally works on a $p \times q$ process grid with many more processes. However, `PXGEMR2D` in Algorithm 2 works on only p or q processes and the number of involved processes is much smaller than the original `PXGEMR2D` in ScaLAPACK.

From the implementation point of view, we need to construct a BLACS context for each process row and each process column, and then all the communications are performed within each context. The implementation is simple, and only needs to call several BLACS routines such as `BLACS_GRIDMAP`. `PXGEMR2D` includes some good techniques for some regular redistribution patterns such as from block-cyclic(s) to block-cyclic(ks) or reverse, where k is an integer. Therefore, it is good to use `PXGEMR2D` to implement our two-step communication algorithm instead of writing new codes similar to the irregular case in section 2.1. The performance results are shown in Example 4 in section 4.

Algorithm 2. Two-Step Data Redistribution From 2D BCDD to Another 2D BCDD

Input: An input $p \times q$ process grid, matrix A is stored in BCDD with row block size m_b and column block size n_b, and new row block size $\hat{m}_b$ and column block size $\hat{n}_b$.

Output: Matrix B is stored in BCDD with row block size $\hat{m}_b$ and column block size $\hat{n}_b$.

Phase 1: Construct sub-communicators

- Construct BLACS contexts (`ContextC`(i)) for q process columns and BLACS contexts (`ContextR`(i)) for p process rows by calling `BLACS_GRIDMAP`;
- For each `ContextC`(i) and `ContextR`(i), construct row and column BLACS descriptors based on n_b and m_b, respectively;
- For each `ContextC`(i) and `ContextR`(i), construct new row and column BLACS descriptors based on $\hat{n}_b$ and $\hat{m}_b$, respectively;

Phase 2: Column communication

- For each `ContextC`(i), processes in the ith column call `PXGEMR2D` to redistribute A from row block size m_b to $\hat{m}_b$, and the intermediate matrix is stored in $\widehat{A}$;

Phase 3: Row communication

- For each `ContextR`(i), processes in the ith row call `PXGEMR2D` to redistribute $\widehat{A}$ from column block size n_b to $\hat{n}_b$, and the final matrix is stored in B.

3 Complexity Analysis

In this section, we analyze the communication cost of our algorithms, and mainly compare them with those in BLACS. The time for a data redistribution algorithm can be generally modeled as follows:

$$T = T_{comp} + T_{comm}, \tag{1}$$

where T_{comp} is the time of performing communication scheduling and packing and unpacking data, and T_{comm} is the time for an algorithm to send and receive data among processes. Since we also use the line segment approach to determine communication processes, which is similar to that used in BLACS, and it is shown in [29] that the communication time will dominate as the size of communication data increases, we analyze only the communication time.

2. https://github.com/jorants/MV-matching-C.git

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

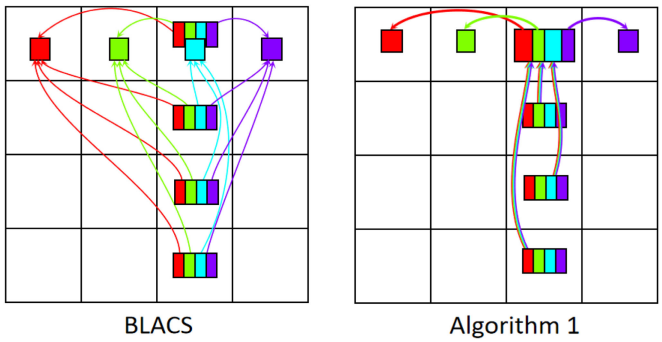


Fig. 10. The communication process of BLACS and Algorithm 1.

In our machine model, a single point-to-point communication is modeled as $\alpha + \beta L$, where α is the start-up time (or communication latency), β is the time for transferring one word without latency (or reciprocal of bandwidth) and L is the message size in words. T_{comm} is approximated by a linear function of the number L of items communicated and the number of times k a message is communicated, i.e.,

$$T_{comm} = k\alpha + \beta L.$$

BLACS uses only one communicator, and requires $O(p \cdot q)$ pair-wise all-to-all communications when using the caterpillar algorithm. Algorithm 1 and Algorithm 2 proposed in this work require only $O(p+q)$ rounds of simultaneous point-to-point communications when the caterpillar algorithm is used. Therefore, our algorithms can reduce the latency cost a lot in the worst case, i.e., $(p+q) \ll p \cdot q$. In recent decades, the improvement of communication latency on high performance computers is slower than that of communication bandwidth. Therefore, the newly proposed algorithms should work well on supercomputers with relatively high network bandwidth, and they are relatively not sensitive to network latency since the number of communications is reduced from $O(p \cdot q)$ to $O(p+q)$. The communication latency cost can be reduced a lot.

One drawback of our algorithms is that they consist in two steps, and require some more temporary memory and communicate more data than the BLACS routine. Fig. 10 shows the differences between `PXGEMR2D` and Algorithm 1. The BLACS routine `PXGEMR2D` sends the data directly from the source process to the target process, and performs in one step. Algorithm 1 first collects all the data from processes in the same column and then redistributes the data to the target processes. Since the messages should be transferred to one intermediate process and then transferred to the target process, it is easy to see that Algorithm 1 would communicate more data than BLACS but less than two times more data. The message sizes of Algorithm 1 can be p or q times larger than the message size of `PXGEMR2D`, since the data are first collected from processes in the same process row or column and then sent to the target process. Performance results in the next section show that our algorithms are usually faster than the BLACS routine `PXGEMR2D` for matrices with small dimensions when using many processes.

TABLE 1
The Test Platform and Environment of One Node

Items	Values
2*CPU	Intel Xeon CPU E5-2692 v2@2.2GHz
Memory size	64GB (DDR3)
Operating System	Linux 3.10.0
Compiler	Intel ifort
MPI	mpich-3.2.1
Optimization	-O3 -mavx

4 Performance Results

All the experimental results are obtained on the Tianhe-2A supercomputer [32], [33], located in Guangzhou, China. Each compute node is equipped with two 12-cores Intel Xeon E5-2692 v2 CPUs and our experiments use only CPU cores. The details of the test platform and environment of compute nodes are shown in Table 1. For all these numerical experiments, we used only plain MPI, and run 24 MPI processes per node. Our codes will be released on Github[3].

Example 1. In this example, we test Algorithm 1 and redistribute 1D irregular data to 2D BCDD form. We use three random matrices with dimensions $n = 10000, 30000$, and 60000, respectively. We only show the results for real double precision matrices, and the results for double complex and single precisions should be similar. The vectors are originally stored locally in $n_p = 256, 1024$ and 4096 processes and their column dimensions are generated randomly, and thus matrix A is stored in an irregular 1D block form. We used the Matlab intrinsic function `randsample` to generate the column dimension vector $\mathcal{M} = \{m_i\}_{i=1}^{n_p}$. Some properties of the random vector are given in Table 2, which shows that the data storage is highly irregular. We redistribute A to a 2D BCDD form with block sizes $m_b = n_b = 64$.

The results are shown in Fig. 11. From the results, we can see that the times spent by Algorithm 1 are very small, always less than one second. Note that all the times in this example are obtained by running each algorithm for five times and computing the average times.

Note that `PDGEMR2D` can redistribute A to any 2D BCDD if A is originally stored in 1D BCDD form. Therefore, we cannot compare Algorithm 1 with `PDGEMR2D` in the irregular case. To compare with BLACS, we need to assume that A is originally stored in 1D BCDD with block sizes $m_b^{(0)} = n$ and $n_b^{(0)} = \lceil n/n_p \rceil$ ($\lceil x \rceil$ means the smallest integer not less than x), and then we can use `PDGEMR2D` to redistribute A to 2D BCDD with $m_b = n_b = 64$. For simplicity we assume $n = 4096 * k$, where $k = 2, 4$, and 8. We then compare Algorithm 1 with `PDGEMR2D` which is included in ScaLAPACK with version 2.1.0, and the results are shown in Fig. 12, which shows the speedup of Algorithm 1 over `PDGEMR2D`. We have further tested the case when $m_b = 32$ and $n_b = 64$, and plots are similar to Fig. 12, so the results are not reported.

We use a real double matrix with dimension $n = 16384$ to show the time required by the four major steps:

3. https://github.com/shengguolsg/DataRedist

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
The Statistic Properties of the Numbers of Vectors Which are Stored in Different Processes Locally

n	n_p	Mean	Std_Var	Max	Min
10000	256	39.06	36.48	180	1
	1024	9.77	9.00	62	1
	4096	2.44	1.85	15	1
30000	256	117.19	120.65	779	2
	1024	29.30	29.40	207	1
	4096	7.32	6.81	51	1
60000	256	234.38	219.06	1020	1
	1024	58.59	56.59	373	1
	4096	14.65	14.32	158	1

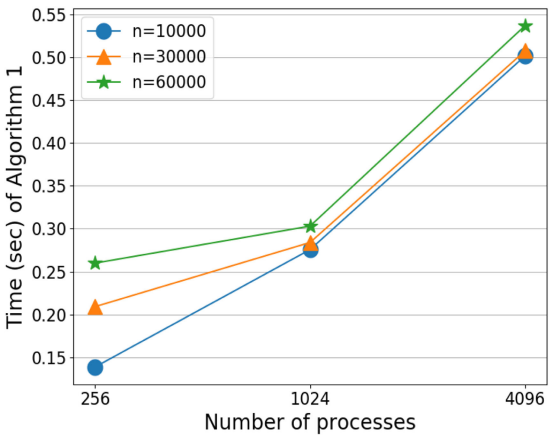


Fig. 11. Data redistribution times of Algorithm 1, which distributes arrays from irregular 1D BDD to 2D BCDD. Times are in seconds.

index computation, communication scheduling, message packing and unpacking, and data transfer. Since both PDGEMR2D and Algorithm 1 use the caterpillar algorithm for scheduling, the cost of communication scheduling is negligible. The costs of the other three major steps are summarized in Table 3. We see that data transfer takes most of the time. For PDGEMR2D, the cost of data transfer increases more quickly than that of Algorithm 1. When using 4096 processes, the cost of data transfer of PDGEMR2D is 6.95 times larger than that of Algorithm 1.

From the results in Fig. 12, we can see that the speedup of Algorithm 1 decreases as the dimension of the matrix increases. It is because Algorithm 1 requires more temporary memory storage, and requires more data to be communicated than BLACS, though the number of communications of Algorithm 1 is reduced. When the number of processes is large, using 4096 processes, Algorithm 1 can be $2\times$–$5\times$ faster than PDGEMR2D.

Example 2. In this example, we use some matrices from real applications to compare Algorithm 1 with PXGEMR2D. We use an eigenvalue problem obtained from a density functional theory (DFT) simulation, in particular a matrix labeled NaCl produced by the FLEUR code [34]. The NaCl matrix is first reduced to a banded matrix by using ELPA [13], and the eigenvectors of this banded matrix are then computed by using the spectrum slicing algorithm

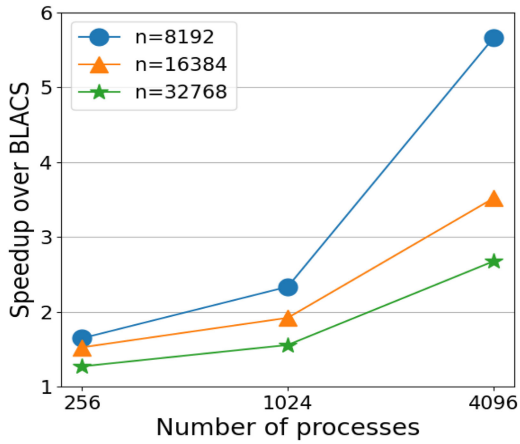


Fig. 12. Speedup of Algorithm 1 over PDGEMR2D for random matrices.

TABLE 3
Time (sec) of Index Computation, Message Packing and Unpacking, and Data Transfer

Method	Step	n_p=256	n_p=1024	n_p=4096
Algorithm 1	Index	9.8e-3	1.3e-2	8.5e-2
	Packing	1.2e-1	1.3e-1	1.6e-1
	Transfer	7.8e-2	1.4e-1	2.3e-1
PDGEMR2D	Index	1.4e-4	3.2e-4	1.4e-3
	Packing	1.4e-2	4.2e-3	1.9e-3
	Transfer	2.4e-1	5.0e-1	1.6

in FEAST [35]. The eigenvectors are locally stored in each process, and they need to be transformed to 2D BCDD format to finish the whole computation.

The matrix NaCl is complex and its size is 9273. We use different processes (n_p=64, 256, 1024, 4096) to compute nev=2000, 5000 or 9273 eigenpairs. The eigenvectors are locally stored in different processes. Note that some processes may have none of eigenvectors when nev is smaller than n_p. Assume the eigenvector matrix is denoted by X. We can use Algorithm 1 to transform matrix X to its 2D BCDD format. Note that PZGEMR2D does not work for irregular data distributions, and we do not include the time of data redistribution from irregular to regular 1D BDD. Algorithm 1 redistributes 1D irregular data to 2D BCDD directly, and all times are included, which places Algorithm 1 at a disadvantage when compared to PZGEMR2D. But our algorithm can still be faster than PZGEMR2D, and the speedup can be more than 10. The speedups of Algorithm 1 over PZGEMR2D are shown in Fig. 13. The data redistribution times in seconds are shown in Fig. 14.

We note that PZGEMR2D takes the longest time when $nev = 2000$ and $n_p = 4096$. In this case, each of the first 2000 processes initially has only one vector, and the other processes do not have any vector, and after redistribution only the processes in the first 32 columns have data, when the processes are organized as a 64×64 grid. This may be a very extreme case for PZGEMR2D, but Algorithm 1 can still solve it efficiently.

Example 3. In this example, the algorithm that redistributes a matrix A from 1D irregular to 1D BCDD form is tested.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

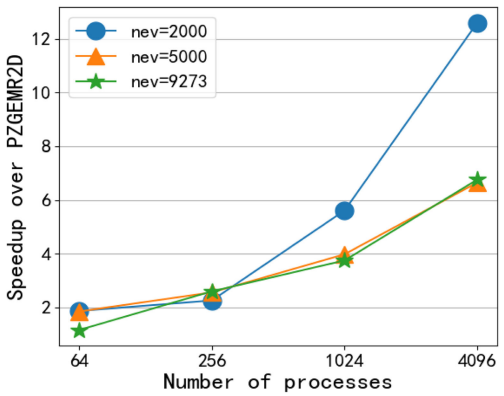


Fig. 13. Speedup of Algorithm 1 over PZGEMR2D for matrix NaCl.

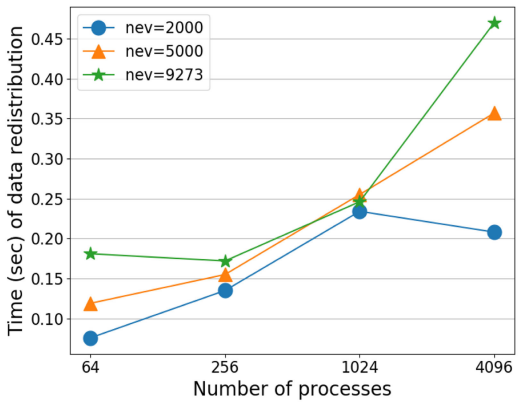


Fig. 14. Time (sec) of data redistribution from irregular 1D BDD to 2D BCDD for matrix NaCl.

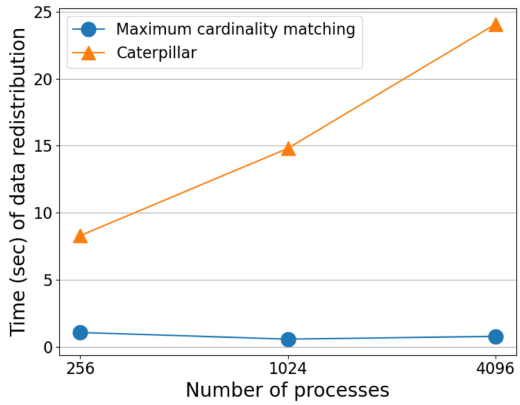


Fig. 15. Data redistribution times with different scheduling: Caterpillar (used in ScaLAPACK) and Maximum cardinality matching.

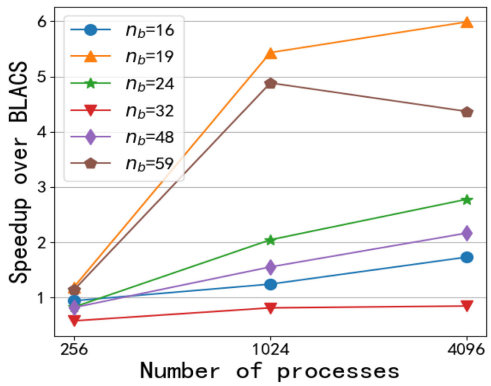


Fig. 16. Speedup of Algorithm 2 over PDGEMR2D in BLACS.

The goal is to show that the dynamic scheduling based on maximum cardinality matching (MCM) can be much faster than the caterpillar algorithm especially when the number of processes is large. We use a random matrix with dimension $n = 30000$ which is stored column-wise in $n_p = 256, 1024$ and 4096 processes and the column dimensions are generated randomly. We redistribute it to 1D BDD form with column block size $n_b = \lceil n/n_p \rceil$. The execution times are shown in Fig. 15 when using the caterpillar algorithm and the MCM approach. From the results, we can see that the MCM approach can be more than $30\times$ faster than using the caterpillar algorithm. This is because the processes need to communicate with several neighbors and the number of communications is significantly reduced by using the MCM-based communication scheduling. The time cost of MCM is very small, because the number of edges in the graph is not large in our case. When applying MCM to 2D data redistribution, the number of edges will become much larger, and its cost would become an important issue.

The MCM-based 1D data redistribution algorithm can also be incorporated in Phase 2 of Algorithm 1. Since the number of processes in each process row or column is small, the gain of using MCM-based approach is not much, and it further requires some extra floating-point operations when constructing the MCM-based communication scheduling. The MCM-based approach also depends on the implementation of the MCM algorithm and the performance of computer processor units (CPU). For supercomputers composed of weak CPUs, MCM would take up most of the time, which may reduce the overall performance of the whole algorithm.

Example 4. In this example, we compare Algorithm 2 with PDGEMR2D when redistributing a dense matrix in one 2D BCDD form to another 2D BCDD form. We use a random dense complex matrix A with dimension $n = 30000$ in double precision. We assume the row and column block sizes of BCDD are equal, $m_b = n_b$, and matrix A is redistributed from BCDD with block size n_b to another BCDD with $\hat{n}_b$. We use three different numbers of processes, $n_p = 256, 1024$ and 4096. The speedups of Algorithm 2 over PDGEMR2D are shown in Fig. 16. We note that the time variations are small, less than 10% of the average times. Therefore, the speedups are computed from the average times of five runs.

We fix $\hat{n}_b$ to be 64 and choose six different n_b, and every two values of n_b represents a different case:

- $n_b = 19, 59$ denote the case that $\hat{n}_b$ and n_b have only one common factor, one (relatively prime);
- $n_b = 24, 48$ denote the case that $\hat{n}_b$ and n_b have some common factors besides one;
- $n_b = 16, 32$ denote the case that $\hat{n}_b$ is a multiple of n_b.

For the first two cases, Algorithm 2 is always faster than PDGEMR2D when using many processes, and the speedup can be up to $6\times$ when using 4096 processes. Algorithm 2 can be slower than PDGEMR2D when $\hat{n}_b = k \cdot n_b$ and k is an integer. However, their performances

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

are similar when using more than 4096 processes and $n_b = 32$. When $n_b = 16$, Algorithm 2 can be also faster than PDGEMR2D. Note that there are some optimized algorithms for the case $\hat{n}_b = k \cdot n_b$ such as [3], [5]. Since these algorithms have some constraints and their implementations are difficult for the general case, we don't incorporate these techniques in our implementation. Performance results show that PDGEMR2D is usually better when $\hat{n}_b = k \cdot n_b$ and Algorithm 2 is a better choice for the other values of n_b and $\hat{n}_b$.

5 Conclusion

In this paper, some efficient data redistribution algorithms are proposed for matrices from irregular or regular form to 1D and 2D block cyclic data distribution (BCDD) form, which can be much faster than the BLACS routine PXGEMR2D, even for the case of one 2D BCDD to another 2D BCDD redistribution. Note that PXGEMR2D does not work for irregular data and we only compare it with our algorithms for the cases that PXGEMR2D works. We further compare the dynamic communication scheduling based on maximum cardinality matching with the caterpillar approach [29].

The central idea of our algorithms is to divide the communication into two steps: one for processes in the same column and the other for processes in the same row. The whole data redistribution task is divided into several independent subcommunications. Our approaches are expected to have less latency cost. Performance results show that our algorithms can be $2\times$–$5\times$ faster than the BLACS routine PXGEMR2D when using 4096 processes on Tianhe-2A supercomputer.

Acknowledgments

The authors would like to thank the referees for their valuable comments which greatly improve the quality of this paper. The authors would like to thank Min Xie from NUDT for many valuable discussions, and Jose E. Roman from Universitat Politècnica de València, Spain, for helpful suggestions and helping us polish the English.

References

[1] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr, and M. Zosel, *The High Performance Fortran Handbook*. Cambridge, MA, USA: MIT Press, 1994.

[2] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert, "Scheduling block-cyclic array redistribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 2, pp. 192–205, Feb. 1998.

[3] N. Park, V. K. Prasanna, and C. S. Raghavendra, "Efficient algorithms for block-cyclic array redistribution between processor sets," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 12, pp. 1217–1240, Dec. 1999.

[4] L. Prylli and B. Tourancheau, "Fast runtime block cyclic data redistribution on multiprocessors," *J. Parallel Distrib. Comput.*, vol. 45, no. 1, pp. 63–72, 1997.

[5] D. W. Walker and S. W. Otto, "Redistribution of block-cyclic data distributions using MPI," *Concurrency: Pract. Experience*, vol. 8, no. 9, pp. 707–728, 1996.

[6] J. Subhlok, J. M. Stichnoth, D. R. O'hallaron, and T. Gross, "Exploiting task and data parallelism on a multicomputer," in *Proc. 4th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 1993, pp. 13–22.

[7] H. Wang, M. Guo, S. K. Prasad, Y. Pan, and W. Chen, "An efficient algorithm for irregular redistributions in parallelizing compilers," in *Proc. Int. Symp. Parallel Distrib. Process. Appl.*, 2003, pp. 76–87.

[8] H. Wang, M. Guo, and D. Wei, "A divide-and-conquer algorithm for irregular redistribution in parallelizing compilers," *J. Supercomputing*, vol. 29, no. 2, pp. 157–170, 2004.

[9] S. Li, X. Wu, J. E. Roman, Z. Yuan, R. Wang, and L. Cheng, A parallel direct eigensolver for sequences of hermitian eigenvalue problems with no tridiagonalization, 2020. [Online]. Available: https://arxiv.org/abs/2012.00506

[10] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems A Practical Guide*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[11] J. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: SIAM, 1997.

[12] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a modern distributed and accelerated linear algebra library," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 1–18.

[13] A. Marek *et al.*, "The ELPA library: Scalable parallel eigenvalue solutions for electronic structure theory and computational science," *J. Phys.: Condens. Matter*, vol. 26, pp. 1–15, 2014.

[14] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed., Baltimore, MD, USA: The Johns Hopkins Univ. Press, 1996.

[15] X. Liao, S. Li, Y. Lu, and J. E. Roman, "A parallel structured divide-and-conquer algorithm for symmetric tridiagonal eigenvalue problems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 367–378, Feb. 2021.

[16] S. Li, M. Gu, L. Cheng, X. Chi, and M. Sun, "An accelerated divide-and-conquer algorithm for the bidiagonal SVD problem," *SIAM J. Matrix Anal. Appl.*, vol. 35, no. 3, pp. 1038–1057, 2014.

[17] S. Balay *et al.*, "PETSc/TAO users manual," Argonne National Laboratory, Lemont, IL, USA, Rep. No. ANL-21/39 - Revision 3.17, 2022.

[18] V. Hernandez, J. E. Roman, and V. Vidal, "SLEPc: A scalable and flexible Toolkit for the solution of eigenvalue problems," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 351–362, 2005.

[19] E. Polizzi, "Density-matrix-based algorithm for solving eigenvalue problems," *Phys. Rev. B*, vol. 79, no. 11, 2009, Art. no. 115112.

[20] J. Dongarra *et al.*, "Applied mathematics research for exascale computing," Lawrence Livermore National Lab.(LLNL), Livermore, CA, USA, Tech. Rep. LLNL-TR-651000, 2014.

[21] V. W.-Z. Yu *et al.*, "ELSI–an open infrastructure for electronic structure solvers," *Comput. Phys. Commun.*, vol. 256, 2020, Art. no. 107459.

[22] J. Choi *et al.*, "ScaLAPACK: A portable linear algebra library for distributed memory computers-design issues and performance," *Comput. Phys. Commun.*, vol. 97, pp. 1–15, 1996.

[23] J. J. Dongarra, R. A. van de Geijn, and R. C. Whaley, "Two dimensional basic linear algebra communication subprograms," in *Proc. 6th SIAM Conf. Parallel Process. Sci. Comput.*, 1993, pp. 347–352.

[24] C.-H. Hsu, Y.-C. Chung, and C.-R. Dow, "Efficient methods for multi-dimensional array redistribution," *J. Supercomputing*, vol. 17, no. 1, pp. 23–46, 2000.

[25] S. Ramasulamy and P. Banerjee, "Automatic generation of efficient array redistribution routines for distributed memory multicomputers," in *Proc. Front. 5th Symp. Front. Massively Parallel Comput.*, 1995, pp. 342–349.

[26] R. L. Graham, *Handbook of Combinatorics*, vol. I, Amsterdam, The Netherlands: Elsevier, 1995.

[27] S. Micali and V. V. Vazirani, "An $O(\sqrt{|v|} \cdot |E|)$ algoithm for finding maximum matching in general graphs," in *Proc. 21st Annu. Symp. Found. Comput. Sci.*, 1980, pp. 17–27.

[28] M. Huang and C. Stein, "Extending search phases in the micali-vazirani algorithm," in *Proc. 16th Int. Symp. Exp. Algorithms*, 2017, pp. 10:1–10:19.

[29] L. Prylli and B. Tourancheau, "Efficient block cyclic data redistribution," in *Proc. Eur. Conf. Parallel Process.*, 1996, pp. 155–164.

[30] Q. Cao, G. Bosilca, W. Wu, D. Zhong, A. Bouteiller, and J. Dongarra, "Flexible data redistribution in a task-based runtime system," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2020, pp. 221–225.

[31] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimizations for efficient array redistribution on distributed memory multicomputers," *J. Parallel Distrib. Comput.*, vol. 38, no. 2, pp. 217–228, 1996.

[32] X. Liao, L. Xiao, C. Yang, and Y. Lu, "Milkyway-2 supercomputer: System and application," *Front. Comput. Sci.*, vol. 8, no. 3, pp. 345–356, 2014.

[33] Y. Liu *et al.*, "623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 39–54, 2016.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.

[34] S. Blügel, G. Bihlmayer, and D. Wortmann, The FLEUR code, 2017. [Online]. Available: http://www.flapw.de
[35] J. Kestyn, V. Kalantzis, E. Polizzi, and Y. Saad, "PFEAST: A high performance sparse eigenvalue solver using distributed-memory linear solvers," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2016, pp. 178–189.



Shengguo Li received the BS, MS and PhD degrees from National University and Defense Technology, Changsha, China, in computational mathematics. He is currently an assistant professor with the College of Computer, National University of Defense Technology. His research interests include numerical linear algebra, high performance computing, and machine learning.



Hao Jiang received the MSc and PhD degrees in computational mathematics from the National University of Defense Technology (NUDT), Changsha, China. His research interests include high performance computing, high precision computation and machine learning.



Dezun Dong received the BS, MS, and PhD degrees from the National University of Defense Technology (NUDT), Changsha, China, in 2002, 2004, and 2010, respectively. He is a professor with the College of Computer, National University of Defense Technology, where he leads the research group of high-performance network and architecture (HiNA) and serves as the deputy director designer of Tianhe supercomputer. His research interests focus on high-performance network and architecture for supercomputer, datacenter and deep learning systems. He has published more than 70 peer-reviewed papers in reputed international journals and conferences.



Chun Huang received the MSc and PhD degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China. She is a full professor in computer science with the National University of Defense Technology, where she serves as the director designer of Tianhe supercomputer compilation system. Her research interests include high-performance computing, system software, parallel compilers, parallel programming, performance optimization, and high-performance math libraries.



Jie Liu received the MSc and PhD degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China. He is a professor with the College of Computer, National University of Defense Technology. His research interests include high performance computing, CFD and machine learning.



Xia Liao received the PhD degree from the College of Computer Science, National University of Defense Technology (NUDT), Changsha, China. Her research interests include high performance computing, Big Data analysis and processing and parallel computing.



Xuguang Chen received the BS degree from the College of Science, National University of Defense Technology (NUDT), Changsha, China. He is an assistant professor with the College of Computer, National University of Defense Technology. His research interests include high performance computing and structural finite element simulation.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on July 17,2022 at 21:28:25 UTC from IEEE Xplore. Restrictions apply.