Check for updates

ACM DIGITAL LIBRARY

acm open

Latest updates: https://dl.acm.org/doi/10.1145/3774934.3786446

RESEARCH-ARTICLE

A Diagonal Block Memory-Aware Polynomial Preconditioner for Linear and Eigenvalue Solvers

XIAOJIAN YANG, National University of Defense Technology China, Changsha, Hunan, China

YUHUI NI, National University of Defense Technology China, Changsha, Hunan, China

FAN YUAN, Xiangtan University, Xiangtan, Hunan, China

SHENGGUO LI, National University of Defense Technology China, Changsha, Hunan, China

DEZUN DONG, National University of Defense Technology China, Changsha, Hunan, China

CHUANFU XU, National University of Defense Technology China, Changsha, Hunan, China

View all

Open Access Support provided by:

National University of Defense Technology China

Xiangtan University

University of Chinese Academy of Sciences

PDF Download
3774934.3786446.pdf
22 February 2026
Total Citations: 0
Total Downloads: 149

Published: 28 January 2026

Citation in BibTeX format

PPoPP '26: 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming
January 31 - February 4, 2026
NSW, Sydney, Australia

Conference Sponsors:
SIGHPC
SIGPLAN

A Diagonal Block Memory-Aware Polynomial Preconditioner for Linear and Eigenvalue Solvers

Xiaojian Yang*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
yangxj@nudt.edu.cn

Yuhui Ni*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
n.yh@foxmail.com

Fan Yuan*
School of Mathematics and Computational Science, Xiangtan University
Xiangtan, China
fyuan@smail.xtu.edu.cn

Shengguo Li†
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
nudtlsg@nudt.edu.cn

Dezun Dong†
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
dong@nudt.edu.cn

Chuanfu Xu
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
xuchuanfu@nudt.edu.cn

Haipeng Jia
Institute of Computing Technology, University of Chinese Academy of Sciences
Beijing, China
jiahaipeng@ict.ac.cn

Jie Liu
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
liujie@nudt.edu.cn

Abstract

Krylov subspace methods are widely used in scientific computing to solve large sparse linear systems and eigenvalue problems. Their performance bottleneck is often dominated by high-order matrix-power kernels (MPK), especially in polynomial preconditioners that must scale to millions or billions of variables. We present Diagonal Block MPK (**DBMPK**), a lightweight and parallel-friendly optimization that partitions the input matrix into diagonal blocks and off-diagonal regions. This design enables efficient intra-block data reuse and eliminates inter-block dependencies. It improves cache locality, parallelism, and reduces preprocessing overheads, compared to existing techniques. Our evaluation on x86 and Arm HPC platforms shows that DBMPK improves MPK performance by 26.6%-38.4%. When applied to polynomial preconditioners for linear systems and eigenvalue problems, it achieves consistent end-to-end speedups of 18.6%-34.0%, including in weak scaling tests on 128 nodes, demonstrating strong scalability and practical impact.

CCS Concepts: • **Mathematics of computing** → **Solvers**; • **Computing methodologies** → **Shared memory algorithms**; **Massively parallel algorithms**.

Keywords: Matrix-Power Kernel, Diagonal Block, Polynomial Preconditioning

ACM Reference Format:
Xiaojian Yang, Yuhui Ni, Fan Yuan, Shengguo Li, Dezun Dong, Chuanfu Xu, Haipeng Jia, and Jie Liu. 2026. A Diagonal Block Memory-Aware Polynomial Preconditioner for Linear and Eigenvalue Solvers . In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3774934.3786446

1 Introduction

Krylov subspace methods [24, 58] are fundamental tools in scientific and engineering computing, widely used to solve large sparse linear systems of the form $Ax = b$ and related eigenvalue problems [55, 56]. These solvers are valued for their robustness, scalability, and ability to handle extremely large systems arising in physics simulations [25], structural analysis [8], and machine learning [59].

To accelerate convergence, Krylov methods often rely on *preconditioning*, which transforms the original system into a mathematically equivalent one that is easier to solve. One

*These authors contribute equally to this research.
†Corresponding author

This work is licensed under a Creative Commons Attribution 4.0 International License.

PPoPP '26, Sydney, NSW, Australia

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2310-0/2026/01
https://doi.org/10.1145/3774934.3786446

effective strategy is *polynomial preconditioning*, where the preconditioner is a matrix polynomial $M = p(A)$ that approximates A^{-1}. Compared to other preconditioners, polynomial preconditioners are memory-efficient, simple to construct, and highly parallelizable, making them well suited for large-scale and memory-constrained applications [23, 24, 55, 56]. More recently, polynomial preconditioning has also been applied effectively to linear eigenvalue problems [37, 38, 62, 68], where the goal is to enhance eigenvalues within a target spectral interval while suppressing all others.

However, the performance of polynomial preconditioners is often limited by the cost of computing the matrix-power kernel (MPK), defined as $w_d = A^d v$. This computation applies d consecutive sparse matrix-vector multiplications (SpMV), one of the most memory-bound operations in numerical linear algebra. Each SpMV requires loading the full matrix A and the intermediate vector w_{i-1}, resulting in high memory traffic and limited arithmetic intensity. As the polynomial degree d increases, the cumulative cost of these SpMV operations can dominate the total runtime. In some cases, the preconditioning step may take longer than solving the original system, thereby negating the benefits of using a preconditioner. Efficient MPK computation is therefore essential to fully realize the advantages of polynomial preconditioning on modern high-performance architectures.

Efforts have been made to reduce memory accesses in MPK computations. For example, [16] exploits sparse matrix data reuse for MPK, while [46] extended this idea to iterative solvers with a communication-avoiding (CA) formulation. Building on these techniques, the Level-Based MPK (LBMPK) method [2, 3] uses level-set scheduling with reordering and recursive partitioning to improve data reuse and parallelism. It computes all contributions from submatrix fragments of A to the intermediate vectors $w_1, w_2, \ldots, w_k$, aiming to maximize reuse of matrix fragments in cache. Although LBMPK offers high theoretical data reuse and strong performance potential, its efficiency is highly sensitive to cache size. As the polynomial degree d increases or the size of submatrix fragments exceeds the available cache, performance degrades sharply. While high-order LBMPK can be decomposed into combinations of lower-order LBMPKs, determining the optimal decomposition remains a challenge. An alternative approach, forward-backward MPK (FBMPK) [67], splits matrix A into its lower (L) and upper (U) triangular parts. It then performs two consecutive SpMV operations to exploit data reuse within L and U. Compared to LBMPK, FBMPK is simpler to implement, imposes fewer constraints on cache size, and delivers more consistent performance across different values of k. However, all these methods introduce strong data dependencies that hinder parallel scalability on multicore CPUs. Moreover, the preprocessing steps required to improve data locality and parallelism, especially for large matrices, can incur significant overhead, potentially offsetting the performance benefits of these optimizations.

We present Diagonal Block MPK (DBMPK[1]), a lightweight optimization strategy to improve the efficiency of MPK computations. DBMPK partitions the matrix into independent diagonal and off-diagonal blocks. Each computing unit (thread or process) is assigned to a diagonal block and performs FBMPK-style computations locally, thereby avoiding intra-block data contention, while the off-diagonal parts are handled using well-established SpMV optimization techniques. This design eliminates data dependencies among diagonal blocks, enabling high parallel scalability while retaining data reuse. Furthermore, the partitioning scheme is fully compatible with standard 1D distributed matrix partitioning strategies, allowing DBMPK to extend naturally to distributed environments.

We evaluate the proposed DBMPK method on two distinct architectures: the Intel x86 Xeon Gold-6348 and the Arm-based Kunpeng 920 [29]. Using 108 diverse matrices from SuiteSparse [15], DBMPK achieves average performance gains of 38.4% and 26.6% over the baseline MPK with serial SpMV calls on the two platforms when the polynomial degree is $d = 40$. Compared with the LBMPK method [2, 3], DBMPK requires a much smaller working set, which improves cache locality while maintaining stable and efficient performance across platforms. In particular, DBMPK delivers strong acceleration for matrices with dense diagonal blocks, whereas matrices with irregular structures can benefit from reordering techniques to improve element distribution. DBMPK also incurs minimal preprocessing overhead, equivalent to only 9.5 to 12.9 SpMVs on average.

We have integrated DBMPK into the YHAMG [64] library[2] and tested the enhanced library on a range of important workloads, including polynomial preconditioning for sparse linear systems [43], polynomial filters for eigenvalue problems, and polynomial smoothers for multigrid methods. Across these applications, DBMPK significantly outperforms equivalent methods in established algebraic libraries, including Trilinos [27], EVSL [37], and HYPRE [21], while preserving numerical stability and convergence properties. In particular, DBMPK delivers 18.6%–34.0% performance improvements in YHAMG. For 7-point stencil matrices on a 128-node Intel cluster, DBMPK achieves a 34% efficiency gain in weak scaling tests as we increase the number of parallel processes.

This paper makes the following contributions:

- **Scalable decomposition:** We introduce DBMPK, a diagonal/off-diagonal partitioning scheme that removes

[1]Code and data available at https://github.com/YXJ-123/DBMPK.

[2]YHAMG is an industry-graded high-performance algebraic multigrid package that incorporates advanced memory-aware smoothers and efficient restriction/interpolation operators [63, 65, 66], and consistently outperforms HYPRE [22] on modern architectures.

Algorithm 1: General Polynomials for Equation (1).

Input: Matrix A; Vector x; $a_{0,\ldots,d}$.
Output: Vector y.

```
1 y = a_0 x;  v = x
2 for i from 1 to d do
3 |   v = A * v;  y+ = a_i v
4 end
```

inter-block dependencies and scales efficiently on multicore and distributed systems.

- **Lightweight and portable:** DBMPK achieves stable performance with only minimal preprocessing and is robust to various cache sizes.
- **Practical and effective:** Integrated into the YHAMG library, DBMPK consistently outperforms state-of-the-art libraries while preserving numerical stability and convergence.

2 Background

2.1 Sparse Matrix Polynomial

Polynomial preconditioners improve the convergence of iterative solvers while avoiding the cost and complexity of factorisation-based methods [14, 55]. They approximate A^{-1} with a matrix polynomial, making them lightweight, easy to implement, and naturally suited for parallel platforms. Common variants include Chebyshev, least-squares, and Neumann series expansions [5, 20, 32, 56, Sec. 12.3]. For a linear system $Ax = b$, the general form is

$$Ap(A)y = b, \quad x = p(A)y, \tag{1}$$

where $Ap(A)$ usually has a smaller condition number than A, thereby accelerating convergence.

Applying a polynomial preconditioner requires evaluating $p(A)x$. In the monomial basis,

$$p(A) = \sum_{i=0}^{d} a_i A^i, \tag{2}$$

which reduces to a sequence of SpMV, also known as MPK. Since MPK dominates runtime, the efficiency of polynomial preconditioning critically depends on optimizing this kernel. However, the monomial basis can be numerically unstable for large d, so Chebyshev or Newton bases are typically adopted [6, 7, 12, 17].

2.2 Memory-aware MPKs

Figure 1(a) shows that the classical MPK requires d sequential calls to the matrix A for SpMV computations. Recent studies have observed that matrix fragments residing in cache can be reused across multiple SpMV operations, thereby improving performance, which we describe as follows.

2.2.1 Level-based MPK. One approach to optimize MPK is the Level-Based MPK (LBMPK) [2, 3]. As illustrated in Figure1(b), once certain rows of the matrix are computed, the resulting partial output vector is already sufficient to trigger additional SpMV updates for other rows. Because these rows remain in cache, they can be naturally reused. This process resembles a "wave" propagation, where some rows are reused multiple times across successive multiplications. For high-degree polynomials, LBMPK attempts to maximize cache utilization, making its performance heavily dependent on cache size. In theory, with a sufficiently large cache, each matrix element could be reused d times in a single pass, requiring the matrix to be read only once from memory. In practice, however, cache capacity is far too limited for large-scale problems. Moreover, LBMPK produces fragmented outputs, which makes it difficult to directly integrate with Chebyshev- or Newton-based polynomial preconditioners. A potential workaround is to store intermediate LBMPK results, but this inevitably introduces significant storage overhead.

2.2.2 Forward–backward MPK. The Forward–Backward MPK (FBMPK) method achieves data reuse by traversing the matrix in both forward and backward directions [67]. As shown in Figure 1(c), the sparse matrix is decomposed into the diagonal, strictly upper, and strictly lower triangular parts, each reused across two consecutive SpMV operations. Algorithm 2 summarizes the workflow. The lower triangular part is computed first, followed by the upper triangular part. The first traversal completes the current SpMV (line 3), while the second simultaneously performs half of the next SpMV and stores the result in a temporary vector $temp$ (line 4). Thus, FBMPK reduces the total number of matrix accesses to $(d + 1)/2$ for an MPK of degree d. FBMPK executes calculations and reuses data row by row, lowering cache size requirements. Although its reuse rate is lower than that of LBMPK, its performance is not constrained by polynomial degree and remains highly efficient even for large d [67]. Furthermore, FBMPK integrates naturally with polynomial preconditioners such as Chebyshev forms.

2.3 Data Dependency and Preprocessing

Both LBMPK and FBMPK can accelerate MPK through data reuse, however, they introduce data dependencies in large-scale parallel environments. As shown in Figure 2, we use the lower triangular matrix L in FBMPK to explain it. Suppose two threads are assigned to compute rows i and j, respectively. During the m-th and (m+1)-th SpMV calculations that produce x_m and $temp$, the i-th nonzero in the j-th row (blue dot) must retrieve the i-th entry from the result vector x_m. This forces thread 2 to wait for thread 1 to finish computing row i to generate $x_m[i]$. An analogous dependency arises in matrix U, which is therefore computed from bottom to up. The right side of Figure 2 shows the task dependency graph for rows i and j. It is evident that task rows i and j may also depend on other task rows and also affect the computation of other rows. This converts the memory-bound nature of

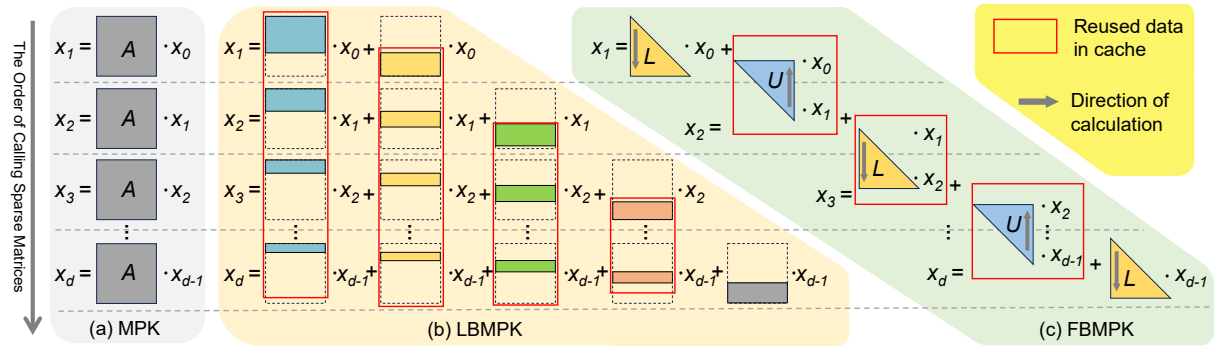


Figure 1. Several ways of calling the matrix for MPK. The number k represents the k-th SpMV operation. (a) is the classical MPK, which serially calls the matrix for SpMV. (b) is LBMPK, which recursively reuses matrix rows for multiple SpMV substeps. (c) is FBMPK, reusing matrix rows to compute two consecutive upper or lower triangular matrices.

Algorithm 2: Polynomials in FBMPK method.

Input: Matrix L, U; Vector D, x; and $a_{0,\ldots,d}$.
Output: Vector y.

$v_0 = x$; $y = a_0 v_0$; $temp = Lx$
for i *from* 1 *to* d **do**
  $v_1 = (U + D)v_0 + temp$
  $temp = (U + D)v_1$ ▷ Except for $i == d$
  $y+ = a_i v_1$
  $i++$, **if** $i > d$: **break**
  $v_0 = Lv_1 + temp$
  $temp = Lv_0$ ▷ Except for $i == d$
  $y+ = a_i v_0$
end

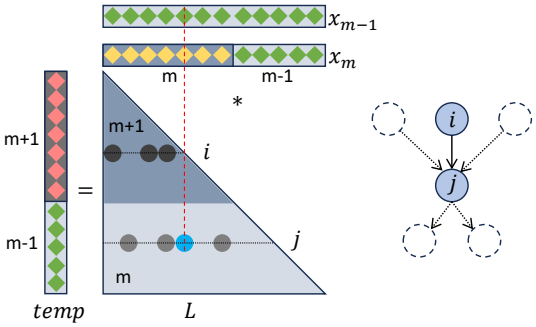


Figure 2. The dependencies in FBMPK. When the j-th row prepares to compute the m-th and $(m+1)$-th SpMV, it requires the i-th result from the vector x_m, which comes from the m-th and $(m+1)$-th SpMV computation of the i-th row. This creates a dependency where the row j relies on the row i.

sequential SpMV into a parallel-bound problem, similar to that in the sparse triangular solve (SpTRSV) kernel.

Many works have been proposed to improve the parallelism of SpTRSV [31, 40, 51], all of which are applicable to the optimized MPK methods mentioned above. For example, FBMPK adopts the Algebraic Block Multicolor (ABMC) reordering method [31] to enhance parallelism. However, these techniques may incur substantial preprocessing costs, potentially exceeding the resources required for the problem itself. Methods that achieve higher acceleration with lower preprocessing costs will be more practical.

3 Diagonal Block MPK

3.1 Matrix Partitioning

To enhance the parallelism of data reuse strategy while reducing preprocessing overhead, we propose a novel parallel partitioning strategy, termed *Diagonal Block MPK* (DBMPK).

As shown in Figure 3, the global sparse matrix is first partitioned across multiple processes by rows according to a predefined strategy. Within each process, the local matrix is further subdivided based on the number of threads. Each local matrix is then decomposed into $A = L + U + D + E$, where L, U, and D are the lower triangular part, upper triangular part, and diagonal entries inside the diagonal blocks, respectively, and E contains the off-diagonal block parts. The F parts denote the inter-process off-diagonal regions.

In the calculation of DBMPK, the L and U matrices follow the idea of the FBMPK method, but only the diagonal blocks are reused instead of the entire matrix. According to Figure 2, data dependencies occurs only between rows in triangular matrices. Unlike FBMPK, which must resolve complex dependencies in large triangular matrices, DBMPK achieves data reuse and improves parallelism by assigning at least one diagonal block to each thread. Since these diagonal blocks are distributed across different rows and columns, inter-thread data dependency issues are avoided.

The DBMPK method therefore has a natural 1D domain decomposition property, enabling efficient extension to distributed memory environments. The E and F parts are computed sequentially according to the nature of matrix powers and can be accelerated with mature SpMV optimizations. Before computing F, inter-process communication is required to update the vector elements needed for multiplication. In this way, a distributed DBMPK is realized.

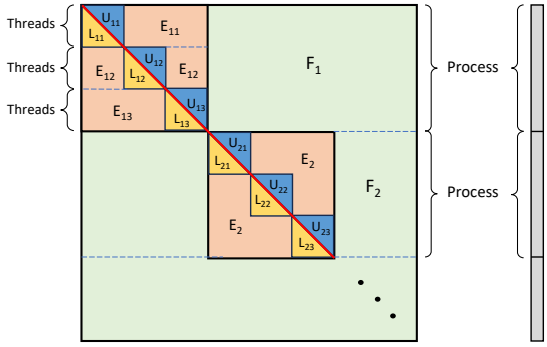


Figure 3. Sparse matrix partitioning strategy of our DBMPK method in shared and distributed memory environments. $L/U/E/F_{ij}$ indicates that this part is calculated by the j-th thread of the i-th process.

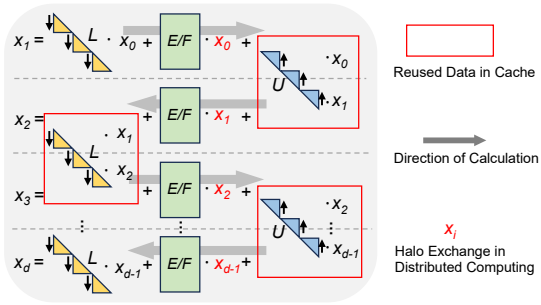


Figure 4. Matrix scheduling flow of the DBMPK method.

Algorithm 3: Polynomials in DBMPK method.

Input: Matrix L, U, E, F; Vector D, x; and $a_{0,\ldots,d}$.
Output: Vector y.

1 $v_0 = x$; $y = a_0 v_0$; $temp = Lx$
2 **for** i *from* 1 *to* d **do**
3 $\quad temp+ = (\mathbf{E}+\mathbf{F})\mathbf{v_0}$ ▷ Full parallelism.
4 $\quad v_1 = (U+D)v_0 + temp$ ▷ Inter-block parallelism.
5 $\quad$ **if** $i < d$: $temp = (U+D)v_1$
6 $\quad y+ = a_i v_1$
7 $\quad i++$, **if** $i > d$: **break**
8 $\quad temp+ = (\mathbf{E}+\mathbf{F})\mathbf{v_1}$ ▷ Full parallelism.
9 $\quad v_0 = Lv_1 + temp$ ▷ Inter-block parallelism.
10 $\quad$ **if** $i < d$: $temp = Lv_0$
11 $\quad y+ = a_i v_0$
12 **end**

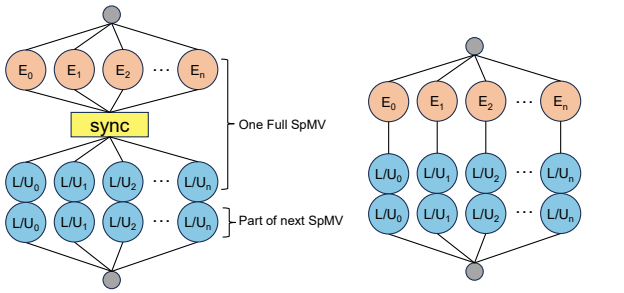


Figure 5. Parallel execution diagram of DBMPK. (a) shows the synchronization issues between different matrix regions caused by matrix block partitioning in parallel computing. (b) assigns the diagonal and off-diagonal matrix regions of the same row block to a single thread, eliminating synchronization and preventing data conflicts.

3.2 Algorithm Description

Figure 4 illustrates the invocation sequence of each matrix block in the DBMPK method. Algorithm 3 details the computational steps for the polynomial evaluation.

Each lower triangular block L is calculated first (line 1), which can be fully parallelized. The off-diagonal blocks E and F are then processed (line 3), where global communication (halo exchange) is required to obtain vector segments from other processes before calculating the matrix F. This communication step is not explicitly included in the algorithm, but computation–communication overlap can be applied by exchanging vector segments for F concurrently with the computation of E. Next, the upper triangular blocks U are computed, completing a full SpMV, i.e., $Av = (L+U+D+E+F)v$. As shown in line 4 of Algorithm 3, the result is stored in v_1. Since the current rows of matrix U remains in the cache, the following SpMV involving U (line 5) can be executed immediately, enabling data reuse.

It is important to emphasize that data reuse occurs at the matrix row level, represented here in matrix form. Since there is no dependency between diagonal blocks (L or U), inter-block parallelism is achieved. This enables recursive computation of matrices U and L, where data reuse occurs at all stages except the last computation.

3.3 Synchronization Free and Load Balance

DBMPK effectively splits the sparse matrix A into two small-scale SpMV operations. The matrix E can be computed using mature SpMV optimization interfaces. However, as shown in Figure 5(a), data reuse introduces dependencies such that the computation of L/U must wait for E to finish, creating a synchronization barrier.

To address this issue, we allocate both E and L/U within the same row block to the same thread. After completing the E and the first L/U computations for that row block, the corresponding portion of the result vector is updated through a full SpMV operation, which can then be reused for the subsequent computation of the matrix L/U of the next SpMV. This scheduling strategy, illustrated in Figure 5(b), eliminates costly synchronization while avoiding data conflicts.

On this basis, load balancing is further applied. Considering the efficiency of the cache, the additional computational

overhead introduced by data reuse is negligible. The matrix rows are partitioned into row blocks according to their number of nonzeros and evenly distributed across threads.

3.4 Performance Analysis

Since DBMPK only reuses the nonzeros located in the diagonal blocks of the matrix, we calculate the proportion of reusable data, i.e.

$$diag_ratio = \frac{L_{nonzeros} + U_{nonzeros}}{A_{nonzeros}}. \tag{3}$$

In general, the larger the $diag_ratio$, the more elements can be reused, leading to better performance. We want the nonzeros of the matrix to be more distributed around the diagonal. Block partitioning and reordering techniques, such as Metis and RCM [34, 55], can be applied together with DBMPK to further improve diagonal density.

In terms of storage overhead, LBMPK requires multiple vectors to store all intermediate results of SpMVs. Our DBMPK method is similar to FBMPK, requiring only one additional vector $temp$ to connect the intermediate results of two consecutive SpMV operations.

For an n dimensional sparse matrix A with nnz nonzeros, the *arithmetic intensity* (AI) of performing the SpMV operation $y = Ax$ in CSR format, defined as the floating-point operations on the volume of data accessed, is

$$\mathtt{AI}(CSR) = \frac{2nnz}{(M_{A_{CSR}} + M_{x,y})}. \tag{4}$$

Here, $M_{A_{CSR}}$ includes $8nnz$ bytes for values, $4nnz$ bytes for column indices, and $4n$ bytes for row indices, while $M_{x,y}$ represents the vectors x and y, each contributing $8n$ bytes. For an MPK of power d, both the numerator and denominator scale by a factor of d. According to the Roofline model [19, 61], the upper bound of MPK equals the product of AI and memory bandwidth, up to the hardware peak performance. Therefore, we estimate the speedup of DBMPK over the classical serial MPK by comparing their AI. Assuming a large power d, DBMPK can be viewed as performing d SpMVs that access the off-diagonal part E and the lower or upper triangular parts of the diagonal blocks (L/U), with an additional intermediate vector $temp$. Its speedup can then be estimated as

$$\begin{aligned}\frac{\mathtt{AI}(DBMPK)}{\mathtt{AI}(MPK_{CSR})} &= \frac{M_{A_{CSR}} + M_{x,y}}{M_{E_{CSR}} + M_{x,y} + M_{L/U_{CSR}} + M_{x,y,temp}} \\ &= \frac{12nnz + 20n}{12(1 - diag_ratio)nnz + 20n + 12 \cdot \frac{diag_ratio}{2}nnz + 28n} \\ &\approx \frac{2r + 3.33}{(2 - diag_ratio)r + 8},\end{aligned} \tag{5}$$

where r denotes $\frac{nnz}{n}$, the average number of nonzeros per row. When equation (5) is expected to be larger than one, we obtain

$$diag_ratio > 5.67/r. \tag{6}$$

Therefore, when $diag_ratio$ is sufficiently large, DBMPK is expected to outperform the classical approach.

Regarding preprocessing costs, DBMPK requires only a simple row-block partition, introducing minimal overhead compared with the recursive level-set partitioning [2] in LBMPK or the algebraic block multicolor method [31] used in FBMPK. Although the amount of reused data is slightly reduced, this trade-off is clearly worthwhile. Due to the substantial number of SpMV and MPK operations required by the solver, many studies argue that the preprocessing overhead can be amortized by the accelerated computations. However, excessively high preprocessing costs remain undesirable in industrial applications. Our experimental results confirm that DBMPK achieves significantly lower preprocessing costs while delivering substantial performance gains, making it highly practical for real-world applications.

4 DBMPK in Polynomial Preconditioning

The DBMPK method can be seamlessly integrated into practical applications while delivering significant performance benefits. This section provides a detailed demonstration using a concrete polynomial preconditioner.

4.1 Standard Form

An effective GMRES polynomial preconditioning for accelerating the solution of linear systems has been presented and integrated into the Trilinos library [20, 43, 44]. The form of this preconditioning can eventually be simplified as

$$p(A)x = \left(\sum_{k=1}^{d} \frac{1}{\theta_k}\left(I - \frac{1}{\theta_1}A\right)\left(I - \frac{1}{\theta_2}A\right)\cdots\left(I - \frac{1}{\theta_{k-1}}A\right)\right)x, \tag{7}$$

where the θ_i are harmonic Ritz values generated via an initial GMRES run. For each subterm in the equation (7)

$$v_k = \left(I - \frac{1}{\theta_1}A\right)\left(I - \frac{1}{\theta_2}A\right)\cdots\left(I - \frac{1}{\theta_{k-1}}A\right)x. \tag{8}$$

When A is real-valued, any complex harmonic Ritz value arise in conjugate pairs. If $\theta_k = p + qi$ is one such value, then $\theta_{k+1} = p - qi$. Thus we can avoid complex arithmetic by combining conjugate pairs. The recursive formula for its next subterm is

$$v_{\text{new}} = \begin{cases} \left(I - \frac{1}{\theta_k}A\right)v_k, \\ \left(I - \frac{A}{\theta_k}\right)\left(I - \frac{A}{\theta_{k+1}}\right)v_k, \end{cases} = \begin{cases} -\frac{1}{\theta_k}Av_k + v_k, & k++ \\ \frac{(A^2v_k - 2pAv_k)}{p^2+q^2} + v_k, & k+=2, \end{cases} \tag{9}$$

which takes into account the case where the harmonic Ritz values are real- and complex-valued, respectively. Combining the above equations, we can derive the formula for the sum of polynomials as follows:

$$p(A)x \mathrel{+}= \begin{cases} \frac{1}{\theta_k}v_k, \\ \frac{v_k}{\theta_k} + \frac{v_{k+1}}{\theta_{k+1}}, \end{cases} = \begin{cases} \frac{1}{\theta_k}v_k, & k++ \\ -\frac{(Av_k - 2pv_k)}{p^2+q^2}, & k+=2. \end{cases} \tag{10}$$

Algorithm 4: Polynomials Preconditioning in DBMPK method.

Input: Matrix L, U, R; Vector D, x; Harmonic Ritz values $\theta_{1,\ldots,d}$.
Output: Vector y.

1 **for** i *from* 1 *to* d **do**
2 $\quad \alpha_i = -1/real(\theta_i)$; $\beta_i = 2 * real(\theta_i)\gamma_i$; $\gamma_i = 1/|\theta_i|^2$
3 **end**
4 $y = \alpha_1 x$; $temp = \alpha_1 Lx$; $g_1 = x$
5 **for** k *from* 1 *to* $d-1$ **do**
6 $\quad$ **if** $k == odd : X = (U + D)$
7 $\quad$ **else if** $k == even : X = L$
8 $\quad swap(g_0, g_1)$
9 $\quad$ **if** $imag(\theta_k) == 0$ **then** ▷ θ_k is real
10 $\quad\quad t_k = \alpha_k$
11 $\quad\quad$ **if** $imag(\theta_{k+1})! = 0 :$ $t_{k+1} = 1$
12 $\quad\quad$ **else if** $imag(\theta_{k+1}) == 0 : t_{k+1} = \alpha_{k+1}$
13 $\quad\quad temp+ = t_k Rg_0 + g_0$
14 $\quad\quad g_1 = t_k Xg_0 + temp$ ▷ Inter-block parallelism.
15 $\quad\quad temp = t_{k+1} Xg_1$ ▷ Except $k == d-1$
16 $\quad$ **else if** $imag(\theta_k)! = 0$ **then** ▷ θ_k Conjugate θ_{k+1}
17 $\quad\quad t_k = 1$
18 $\quad\quad t_{k+1} = \gamma_k$
19 $\quad\quad temp+ = t_k Rg_0$
20 $\quad\quad g_1 = t_k Xg_0 + temp$ ▷ Inter-block parallelism.
21 $\quad\quad temp = t_{k+1} Xg_1 - \beta_k g_1 + g_0$
22 $\quad\quad y+ = -\gamma_{k+1} g_1$
23 $\quad\quad k++$ ▷ θ_{k-1} Conjugate θ_k
24 $\quad\quad t_k = \gamma_k$
25 $\quad\quad$ Repeat lines 6, 7, 8, 11 and 12.
26 $\quad\quad temp+ = t_k Rg_0$
27 $\quad\quad g_1 = t_k Xg_0 + temp$ ▷ Inter-block parallelism.
28 $\quad\quad temp = t_{k+1} Xg_1$ ▷ Except $k == d-1$
29 $\quad$ **end**
30 $\quad$ **if** $imag(\theta_{k+1})! = 0 :$ $y+ = \beta_{k+1} g_1$
31 $\quad$ **else if** $imag(\theta_{k+1}) == 0 : y+ = -\alpha_{k+1} g_1$
32 **end**

4.2 Apply DBMPK to Polynomials

Algorithm 4 describes the computation of GMRES preconditioned polynomials in DBMPK method derived from equations (9) and (10). The algorithm distinguishes between real and complex harmonic Ritz values θ_k, where $real(\theta_k)$ and $imag(\theta_k)$ denote their real and imaginary parts, respectively. Here, g_0 stores the current v_k and g_1 stores the updated v_{new}, and X denotes the part of the triangular matrix selected for this computation.

For real θ_k, only one subterm is calculated in each iteration. The coefficients of v_k and v_{k+1} are first determined (lines 10-12). Then, v_{k+1} is calculated according to equation (9) and stored in g_1, while the matrix block X is reused for the $(k+1)$-th round SpMV (lines 14-15). Finally, the v_{k+1} is accumulated into the polynomial sum (lines 30-31).

For complex θ_k, two consecutive subterms are computed to account for the conjugate pair. In the first step, the intermediate product Av_k of v_{new} is computed, and the term $\frac{-2pAv_k}{p^2+q^2} + v_k$ from equation (9) is stored in $temp$ (lines 20-21). Then, the contribution $\frac{-Av_k}{p^2+q^2}$ of the polynomial sum from equation (10) is added (line 22). The remaining part $\frac{2pv_k}{p^2+q^2}$ has already been incorporated using the v_k obtained in the previous iteration (line 30). In the second step, the remainder part $\frac{A^2 v_k}{p^2+q^2}$ of v_{new} is computed and stored in g_1, thereby completing another round of data reuse (lines 27-28). Similarly, the resulting v_{new} is eventually added to the polynomial sum, completing the update for this conjugate pair.

Table 1. Hardware platforms used in evaluation.

Intel(R) Xeon(R) Gold 6348	Kunpeng (KP) 920
28 #Cores with 2.6GHz Freq.	32 #Cores with 2.6GHz Freq.
L1 cache(I/D) 32K/48K	L1 cache(I/D) 64K/64K
L2 cache 1.25MB	L2 cache 0.5MB
L3 cache 42MB	L3 cache 64MB

5 Evaluation

5.1 Setup

We evaluate our polynomial optimization strategy on two multicore CPUs: one Intel Xeon Gold-6348 X86 CPU@2.6GHz and one Kunpeng 920 (KP 920) ARMv8 CPU@2.6GHz. Their information is shown in Table 1. The experiments are configured with GCC-11.2.0, MPICH-4.0.2 and LAPACK-3.10.0, with the '-O3' option enabled.

The experiment uses 108 sparse matrices from the SuiteSparse Matrix Collection [15]. All matrices are large in size ($rows$ > 20,000) and avoid duplicate matrix types. Several representative matrices are selected for detailed analysis.

Our experiment consists of two parts: (1) analyzing MPK methods and (2) applying polynomial preconditioning in practice (Section 6). We evaluate five variants of MPK optimization: sequential SpMVs using the CSR format (**CSR**), sequential SpMVs with Intel MKL (**MKL**), an LBMPK-based optimization (**LBMPK**) [2–4], FBMPK-based optimization (**FBMPK**) [67], and our proposed method (**DBMPK**). For fairness, we apply the same load balancing to the CSR version as in DBMPK, focusing solely on the algorithmic advantages. The performance of LBMPK is strongly influenced by cache size (see Figure 15 in [3]), and selecting the optimal power for different matrices is empirically challenging. To ensure stable results, we configure LBMPK with recursive calls up to power $d = 5$. Subsection 5.4.2 shows that choosing $d = 5$ offers a reasonable trade-off between efficiency and generality. The FBMPK algorithm employs a parallel SpTRSV heuristic scheduling strategy [31].

To isolate the impact of algorithmic differences, we conduct all MPK evaluations in a single-process (single-node) environment without additional system-level optimizations. In the subsequent polynomial preconditioning experiments,

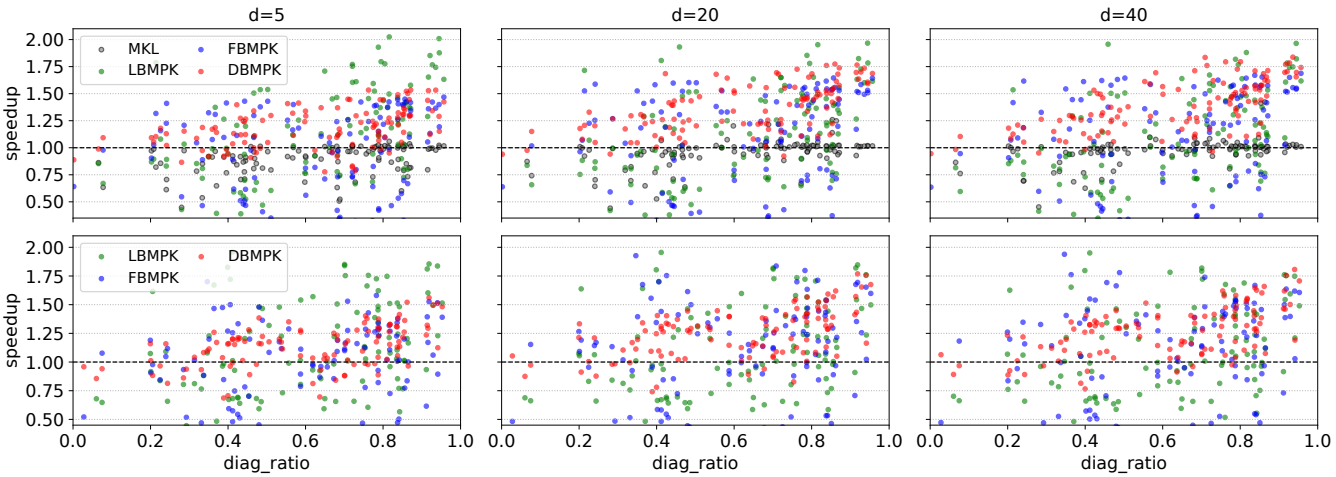


Figure 6. Speedups of different MPK implementations compared to sequential SpMVs in CSR format for 108 sparse matrices on Intel (upper) and KP 920 (lower). The MPK powers are $d = 5$, 20, and 40, with the horizontal axis representing *diag_ratio*.

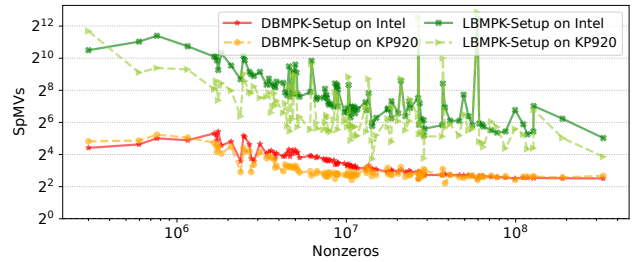


Figure 7. Preprocessing overhead of different MPK implementations for 108 sparse matrices.

we demonstrate that DBMPK delivers strong performance and scalability in distributed computing scenarios.

5.2 Overall Performance

Figure 6 shows the speedups of several optimized MPK implementations relative to the serial CSR version across 108 matrices on two platforms, with polynomial degrees $d = 5$, $d = 20$, and $d = 40$, respectively. The x-axis of each subplot represents the proportion of diagonal block elements *diag_ratio* in equation (3) (corresponding to 28 row blocks on the Intel platform and 32 row blocks on the KP 920 platform). The DBMPK method demonstrates more stable performance improvements when *diag_ratio* is high. However, when *diag_ratio*<30%, the benefit of DBMPK is limited due to the smaller number of reusable elements. The LBMPK and FBMPK methods may exhibit higher acceleration limits for certain matrices; however, for many matrices, their performance gains are modest and can even result in negative speedups. Overall, on the Intel platform, the performance of the MKL implementation is comparable to that of the CSR baseline. LBMPK achieves average accelerations of 16.1%, 13.2%, and 12.8% for all matrices at $d = 5$, 20, and 40, respectively. FBMPK attains speedups of −4.8%, 8.2%, and 9.2%, while DBMPK delivers average improvements of 18.4%, 36.7%, and 38.4% over the same settings. On the KP 920 platform, LBMPK achieves average speedups of 12.5%, 11.0%, and 9.9%, respectively; FBMPK shows improvements of −5.7%, 5.3%, and 5.8%; DBMPK obtains average speedups of 13.6%, 25.2%, and 26.6%, respectively. As the power d increases, DBMPK becomes increasingly advantageous, showing a stable upward trend in acceleration, while LBMPK reaches its performance limit at lower powers.

Table 2. The information of representative matrices. *diag_ratio* is the result when there are 28 row blocks.

ID	Matrix	rows	nnz	nnz/rows	diag_ratio
1	Laplace(7pt)*	7,077,888	49,324,032	6.97	81.6%
2	Laplace(27pt)*	7,077,888	189,119,224	26.72	86.8%
3	FEM_3D_thermal2	147,900	3,489,300	23.59	91.5%
4	BenElechi1*	245,874	13,150,496	53.48	93.6%
5	xenon2	157,464	3,866,688	24.56	86.8%
6	oilpan*	73,752	3,597,188	48.77	83.8%
7	cfd2*	123,440	3,087,898	25.02	77.4%
8	Serena*	1,391,349	64,531,701	46.38	56.8%
9	m_t1*	97,578	9,753,570	99.96	55.4%
10	ship_001*	34,920	4,644,230	133	46.6%
11	cage14	1,505,785	27,130,349	18.02	62.4%
12	msdoor*	415,863	20,240,935	48.67	46.7%
13	CoupCons3D	416,800	22,322,336	53.56	48.3%
14	Transport	1,602,111	23,500,731	14.67	45.9%
15	poisson3Db	85,623	2,374,949	27.74	7.76%

* These matrices are real symmetric, while the rest are nonsymmetric.

5.3 Preprocessing Overhead

Preprocessing overhead is a critical concern in high performance computing, as excessive overhead is undesirable in practice. DBMPK, with its low preprocessing overhead and consistent speedup, presents a more favorable choice in practice. Figure 7 compares the preprocessing overhead of DBMPK and LBMPK on both platforms, where the time cost has been converted into the number of SpMV operations. Specifically, the average preprocessing overhead of DBMPK on the Intel platform is only 12.9 SpMV operations, and on the KP 920 platform is only 9.5 SpMVs. In contrast, the

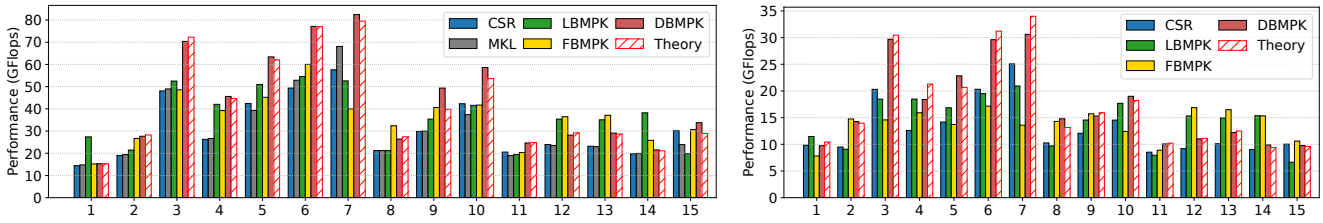
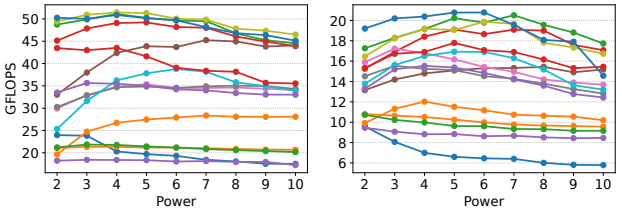


Figure 8. Performance of different MPK implementations on representative matrices on Intel (left) and KP 920 (right) platforms, where the power $d = 20$. #1-15 represent matrix indices in Table 2.



Power	2	3	4	5	6	7	8	9	10
Sum	493	523	538	538	530	528	515	505	499

Power	2	3	4	5	6	7	8	9	10
Sum	205	217	221	222	219	216	207	199	192

Figure 9. Performance of LBMPK on Intel (left) and KP 920 (right) platforms for powers ranging from 2 to 10, with power d=5 exhibiting globally optimal performance.

LBMPK method requires an average of 393 and 302 SpMVs, respectively, and some matrices even require thousands of SpMVs. More critically, the core preprocessing routines of LBMPK have low parallelism, and its relative overhead will be further amplified as the computing resources increase.

5.4 More Analysis

Table 2 lists the characteristics of 15 representative matrices, where *diag_ratio* is calculated with 28 diagonal blocks. These matrices originate from various scientific fields such as structural analysis, computational fluid dynamics, circuit simulation, and chemical analysis, as well as 7-point and 27-point `Laplace` matrices with dimensions of 192^3.

5.4.1 Performance Analysis. Figure 8 presents the performance of various MPK implementations for these matrices. "Theory" refers to the result computed from equation (5), and the actual performance of DBMPK is consistent with it. DBMPK demonstrates substantial speedups over the serial CSR version for matrices #2-11, while maintaining competitive results for the others. This performance advantage primarily stems from higher *diag_ratio* values (46.6%-93.6%) in matrices #2-11, indicating greater opportunities for data reuse. However, high *diag_ratio* alone does not guarantee optimal performance. Equations (5) and (6) demonstrate that DBMPK performance also depends on the matrix's average number of nonzeros per row. When a matrix's nonzeros are concentrated near the diagonal, DBMPK exhibits higher efficiency, especially when there are more nonzeros in each row. However, such matrix pattern poses significant cache challenges for the LBMPK and FBMPK methods.

5.4.2 Sweep of LBMPK. Figure 9 shows the performance of LBMPK as the polynomial power increases from 2 to 10 for the matrices in Table 2 on two platforms. The table beneath the figure summarizes the aggregated performance across all matrices at each power. LBMPK struggles at high powers, and its performance generally degrades once the power exceeds 7. Each matrix attains its peak performance at a different power. Empirically, we find that $d = 5$ provides the best overall performance on both platforms. Consequently, all high-order power evaluations of LBMPK in this work use recursive calls to the power-5 kernel.

5.4.3 Ablation Analysis. DBMPK incorporates two additional optimizations: removing synchronization between the matrix blocks E and L/U, and applying load balancing based on the nonzero count of each row block. Figure 10 illustrates their impact. The synchronization between E and L/U, introduced by matrix partitioning, can significantly degrade performance on parallel systems; eliminating it is thus essential for maintaining DBMPK's scalability. When the off-diagonal blocks have highly imbalanced nonzero distributions–as in matrix ID=#15, `poisson3Db`–load balancing yields substantial performance gains.

5.4.4 Memory Traffic. We measured the memory traffic of several MPK methods using the LIKWID tool [57]. Figure 11 reports their memory traffic relative to the serial CSR-based MPK. All three optimized methods reduce memory traffic in most cases. LBMPK attains the lowest traffic but also suffers from the poorest bandwidth utilization. DBMPK reuses data only within diagonal blocks, giving it slightly higher traffic than FBMPK. However, both LBMPK and FBMPK introduce inter-block data dependencies that incur substantial synchronization overhead, ultimately reducing effective bandwidth compared to DBMPK. By eliminating inter-block synchronization, DBMPK achieves more stable and robust performance.

5.4.5 Matrix Reordering. Reordering is a fundamental strategy in sparse matrix computations as it redistributes nonzeros and enhances data locality. The LBMPK and FBMPK methods incorporate reordering after row scheduling. DBMPK

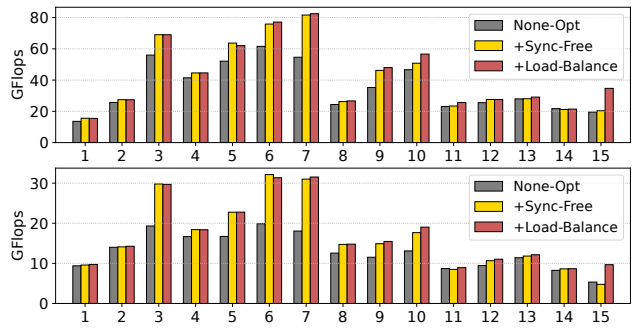


Figure 10. Ablation analysis of DBMPK on Intel (upper) and KP 920 (lower) platforms.

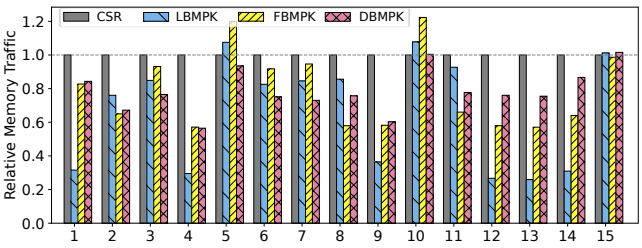


Figure 11. Memory traffic of several optimized MPK methods relative to the serial MPK method in CSR format.

Table 3. Several commonly used sorting algorithms improve the proportion of diagonal block elements in the initial row division of the DBMPK methods.

ID	Initial diag_ratio	+RCM		+GP		+ND	
		ratio	speedup	ratio	speedup	ratio	speedup
1	81.6%	78.8%	1.09×	84.4%	1.16×	79.6%	0.93×
2	86.8%	79.3%	0.96×	94.0%	1.10×	90.6%	0.84×
3	91.5%	81.0%	0.97×	92.1%	0.98×	85.2%	0.86×
4	93.6%	93.5%	1.04×	95.2%	1.01×	88.5%	0.98×
5	86.7%	74.8%	0.97×	90.3%	0.99×	80.8%	0.89×
6	83.9%	73.9%	0.96×	92.5%	0.98×	78.6%	0.92×
7	77.5%	76.0%	0.92×	90.4%	0.93×	80.1%	0.67×
8	56.9%	57.8%	1.02×	93.9%	1.33×	87.9%	1.11×
9	55.6%	56.4%	1.02×	91.2%	1.06×	76.9%	1.07×
10	46.2%	71.8%	1.19×	82.5%	1.24×	63.2%	1.08×
11	63.8%	44.1%	0.86×	85.4%	1.03×	70.2%	0.80×
12	46.7%	85.0%	1.40×	96.4%	1.48×	92.6%	1.44×
13	48.3%	77.7%	1.26×	94.5%	1.39×	87.2%	1.33×
14	45.9%	89.3%	1.40×	91.2%	1.47×	87.7%	1.32×
15	7.8%	56.5%	1.63×	87.6%	1.81×	72.8%	1.56×

can also employs reordering to increase *diag_ratio* and consequently enhance data reuse. We evaluate the effect of several common reordering strategies on diagonal-block occupancy. Specifically, we apply Reverse Cuthill–McKee (RCM) from the SpMP library [30], as well as Graph Partitioning (GP) and Nested Dissection (ND) from Metis [33]. Results for the 15 matrices are summarized in Table 3. For matrices with a high initial *diag_ratio*, DBMPK already delivers strong speedups without reordering. In contrast, for matrices with *diag_ratio* < 50%, applying reordering helps DBMPK achieve more stable and consistent performance improvements.

We emphasize that this subsection only demonstrates that DBMPK can potentially benefit from reordering techniques. However, no reordering strategies are used in any of the experimental analyses presented in this paper.

6 Applications

YHAMG is a parallel algebraic multigrid algorithm library [64], in which we have implemented the polynomial preconditioning for GMRES method, the eigenvalue solvers with polynomial filtering, and the multigrid method using polynomials as smoothers. All of these methods use the polynomials, so we integrate the DBMPK approach into YHAMG and compare it with equivalent methods in state-of-the-art libraries, including Trilinos [27], EVSL [37], and HYPRE [21]. All evaluations of DBMPK in this section are conducted **without** any reordering. ParMetis [35] is used **only** in distributed experiments, where it serves as a standard tool for workload balancing and reducing inter-process communication.

6.1 Polynomial Preconditioning for GMRES

A polynomial preconditioned GMRES for solving sparse linear systems is proposed in [43] and has been integrated into Trilinos library [44]. We reproduce this method in YHAMG and incorporate the DBMPK strategy. Details can be found in Section 4. Table 4 reports the solution times for the matrices in Table 2 on the Intel platform, where GMRES employs a restart frequency of 50 and terminates iteration when the relative residual reaches 10^{-8}. The polynomial power is d=40, except for matrix #8 which is 30. α**N:** β**P**×γ**T** indicates that β processes run on α nodes, with each process containing γ threads. Iteration counts *iters_Tr* and *iters_Y* from both libraries are nearly identical, validating the accuracy of our reproduction. In all tests, DBMPK maintains the same number of iterations for these problems. Due to the extremely large condition number of the matrix `serena` (ID =#8), the use of random right-hand sides led to substantial differences in convergence behavior between the two libraries.

In various resource configurations, YHAMG consistently outperforms Trilinos. The integration of DBMPK further accelerates YHAMG, with especially pronounced benefits in multi-process scenarios. On average, DBMPK improves YHAMG's performance by 29.4%, 31.2%, 22.7%, and 18.6% in the four tested configurations. In the 4N:4P×28T setup, Trilinos yields better results on some matrices. However, in the 4N:16P×7T configuration, Trilinos exhibits substantial slowdowns on some matrices when multiple processes are mapped to the same NUMA node, likely due to memory access optimization mechanisms specific to its NUMA-aware design. In contrast, YHAMG with DBMPK relies solely on algorithmic improvements, ensuring stable performance as the number of processes varies.

Table 4. Performance analysis of solving matrix linear equations based on GMRES polynomial preconditioning with power d = 40. (N: nodes, P: processes, T: threads, Tr: Trilinos, Y: YHAMG, opt: DBMPK)

			1N: 1P×28T			4N: 4P×28T			4N: 16P×7T			4N: 112P×1T		
ID	iters_Tr	iters_Y	time_Tr	time_Y	Y+opt	time_Tr	time_Y	Y+opt	time_Tr	time_Y	Y+opt	time_Tr	time_Y	Y+opt
1	12	13	6.551	5.855	5.641	1.863	1.213	1.084	3.512	1.149	0.999	1.575	0.933	0.810
2	8	8	8.951	8.336	6.070	2.344	1.993	1.477	3.718	2.313	1.598	2.233	1.834	1.181
3	11	11	0.131	0.135	0.108	0.047	0.067	0.056	0.037	0.032	0.031	0.028	0.021	0.025
4	79	79	4.252	4.238	2.477	0.776	0.817	0.623	12.930	0.580	0.474	0.607	0.531	0.367
5	23	25	0.303	0.330	0.240	0.109	0.143	0.116	0.083	0.074	0.069	0.051	0.070	0.070
6	1,635	1,537	16.700	16.623	12.854	5.347	11.600	8.621	3.910	4.278	3.451	2.700	2.459	2.779
7	72	68	0.723	0.694	0.569	0.278	0.456	0.374	0.214	0.236	0.169	0.168	0.141	0.156
8	232	208	71.740	59.15	44.949	20.171	14.992	11.213	65.891	13.895	9.808	19.05	14.291	8.598
9	1,123	1,145	42.980	43.501	28.913	7.338	9.888	9.443	6.987	6.344	5.576	5.374	4.280	4.454
10	377	363	5.687	5.471	4.391	1.855	3.108	2.148	12.880	1.355	1.219	4.042	1.483	1.515
11	2	1	0.190	0.268	0.204	0.068	0.105	0.088	0.579	0.095	0.080	0.091	0.106	0.077
12	1,172	1,150	110.20	102.12	82.789	29.140	20.426	16.775	301.10	16.387	12.735	39.120	14.937	9.598
13	843	859	117.60	116.20	90.723	34.510	25.282	18.565	224.70	18.702	13.912	28.730	16.807	11.027
14	24	24	3.987	3.201	2.810	1.189	1.233	0.599	7.473	0.572	0.413	0.907	0.562	0.445
15	8	8	0.076	0.087	0.077	0.064	0.047	0.039	0.099	0.023	0.024	0.349	0.050	0.069

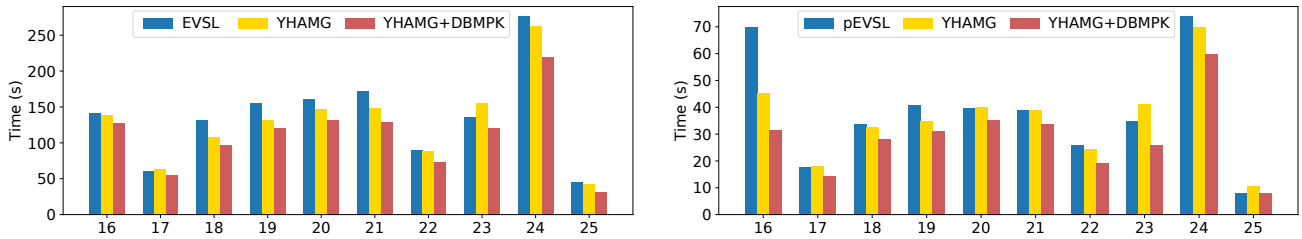


Figure 12. Time comparison of YHAMG+DBMPK, YHAMG and EVSL (left, 1N: 1P×28T). Time comparison of YHAMG+DBMPK, YHAMG and pEVSL (right, 4N: 4P×28T). #16-25 represent matrix indices in Table 5.

6.2 Polynomial Filter for Eigenvalue Problems

Polynomial filtering is a widely used technique for accelerating eigenvalue computations and has been implemented in several eigenvalue solvers, including EVSL [37], SLEPc [26], and ChASE [62]. DBMPK can be seamlessly used to accelerate polynomial filtering for symmetric eigenvalue problems. To this end, we conduct experiments using sparse matrices from the SuiteSparse and ELSES matrix collections [15, 28], as well as the 7-point and 27-point Laplacian matrices.

We adopt the same polynomial filter used in EVSL, a Chebyshev polynomial approximation to the Dirac delta function [38]. We use the polynomial filtering thick-restart Lanczos algorithm (`FiltLanTR`) to solve Hermitian eigenvalue problems. Since the coefficients of the polynomials are strictly real, DBMPK only needs to apply the real-value version of Algorithm 4 (lines 10–15). Our implementation closely follows that of EVSL [37], and pEVSL[3]. The results on the Intel platform are shown in Figure 12. YHAMG demonstrates comparable results to both EVSL and pEVSL, while DBMPK gives YHAMG an average gain of 17.9% to 26.7%.

[3]pEVSL is an MPI version of EVSL that is under development. We compare `FiltLanTR` without restarting in pEVSL against our corresponding one.

Table 5. Eigenvalue information of sparse matrix.

ID	Matrix	rows	nnz	Interval	N_evals
16	Laplace(7pt)	216,000	1,490,400	[0.6, 0.8]	950
17	Laplace(27pt)	216,000	5,639,752	[−4, −2.5]	565
18	Ge87H76	112,985	782,195	[−0.64, −0.8]	1,039
19	Ge99H100	112,985	8,451,395	[−0.65, 0.8]	1,125
20	Si41Ge41H72	185,639	15,011,265	[−0.64, 0.6]	738
21	Si87H76	240,369	10,661,631	[−0.66, 0.4]	677
22	VCNT40000std	40,000	4,059,424	[−0.5, −0.4]	1,479
23	VCNT100000std	100,000	3,999,424	[−0.64, −0.61]	1,020
24	VCNT1000000std	1,000,000	3,099,424	[−0.85, −0.835]	518
25	NCCS430080std	430,080	19,165,216	[−0.8, −0.79]	69

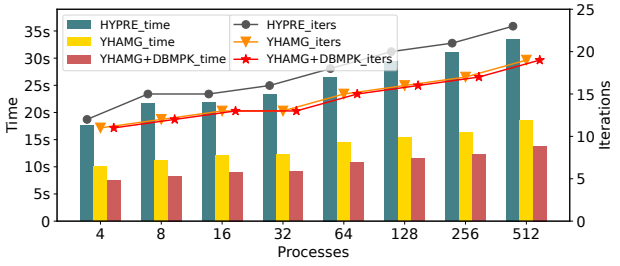


Figure 13. Weak scalability performance of polynomial preconditioned GMRES optimized using the DBMPK strategy.

The version YHAMG_DBMPK is approximately 24.6% to 32.0% faster than EVSL and pEVSL on average.

6.3 Smoothing Operators in Multigrid

Matrix polynomials can serve as smoothers in multigrid methods. We apply multigrid algorithms to solve 3D Laplace (7-point stencil) problem to evaluate the scalability of DBMPK in large-scale parallel environments. The experiments are conducted on the Intel platform with 128 nodes, with each node running 4 processes and each process enabling 7 threads. Each process computes a 192^3 cube subdomain. The multigrid employs HMIS coarsening strategy with interpolation truncated to at most 4 elements per row, and utilizes a 4-th order **Jacobi-Chebyshev** smoother. The system is solved using a CG solver with AMG preconditioner, with the relative residual converging to 10^{-8}. We compare three different implementations: HYPRE, YHAMG, and YHAMG with DBMPK.

Figure 13 shows the test results. As the scale of the problem increases, YHAMG and HYPRE exhibit similar convergence behavior. Due to differences in the implementation of HMIS and interpolation, YHAMG achieves higher speed than HYPRE. The DBMPK strategy does not affect the number of iterations required by YHAMG to solve the problem. In all computational configurations, DBMPK brings an average performance improvement of 34% to YHAMG.

7 Related Work

Chebyshev polynomials and least-squares polynomials can effectively compress the matrix spectrum within a given interval to minimize error and improve the convergence of iterative methods [36, 53, 54]. Therefore, they are common choices for polynomial preconditioners. The polynomial preconditioner for GMRES was proposed in [1, 39], where the coefficients of the polynomial are calculated using power bases and regular equations. This method is only applicable to low-order polynomials. [20] proposed a method that uses harmonic Ritz values generated by initial GMRES iterations to perform a spectral transformation, thereby solving eigenvalues via the Arnoldi method. The literature [43] applied this method to GMRES polynomial preconditioning, achieving more efficient and stable results.

Compared to other preconditioning methods, polynomial preconditioning offers advantages in terms of high parallelism, memory efficiency, and implementation simplicity, with its core consisting of a series of SpMV operations. Extensive research has been devoted to optimize this kernel [11, 41, 42, 45, 50, 60]. Beyond optimizing individual kernels, researchers have increasingly focused on kernel fusion strategies, which aim to reuse cache-resident matrix blocks across successive SpMVs. This approach is particularly effective for higher-order polynomial operations, as it reduces memory traffic by operating on smaller, reused subsets of data [46–48], and has culminated in the development of the RACE library [2]. FBMPK focuses solely on two consecutive SpMV operations [67]. To eliminate data dependencies similar to those in sparse triangular solvers, algebraic block multi-color (ABMC) [31] reordering is required, but this procedure is computationally expensive. Our DBMPK method avoids data conflicts by locally applying the FBMPK method through an offset partitioning of diagonal blocks.

Partitioning row blocks into diagonal and off-diagonal parts is common in sparse matrix libraries such as Hypre. A hierarchical diagonal blocking (HDB) method has been proposed and applied to optimize SpMV [9, 49]. Compared with HDB and Hypre, this paper achieves an effective trade-off among performance, portability, and preprocessing overhead for polynomial preconditioning. DBMPK naturally adopts a 1D distributed partitioning scheme where only halo vectors must be communicated across processes. This communication pattern aligns well with modern network designs [18] that aim to isolate and accelerate small-to-medium jobs. A common 1D row partitioning method is to evenly distribute n rows among p processes, with each process thus having approximately n/p consecutive rows. This distribution method does not guarantee balanced distribution of nonzeros. In the matrix of graph- or grid-based simulations, the distribution of nonzeros is typically highly irregular, making graph partitioning the preferred method. Graph and hypergraph partitioning problems have been extensively studied, and there are several high-quality software tools available, including Scotch [52], PaToH [13], and Zoltan [10]. The distributed application in this paper uses the ParMetis library [35].

8 Conclusion

We have presented DBMPK, a diagonal block memory-aware optimization algorithm for matrix power kernel (MPK) computations in polynomial preconditioners. By partitioning sparse matrices into independent diagonal blocks and off-diagonal regions, DBMPK achieves data reuse of the upper and lower triangular submatrices, eliminates data dependencies, and ensures parallelism. Evaluations on 108 matrices show that DBMPK achieves an average performance improvement of 38.4% on Intel and 26.6% on KP920 platforms for high-order MPK. This design achieves extremely low preprocessing overhead, equivalent to only 12.9/9.5 SpMV operations on Intel/KP 920 platforms, respectively. DBMPK demonstrates outstanding performance across multiple polynomial preconditioning applications. Specifically, it accelerates the solver's runtime by 18.6%–34.0% in distributed parallel environments without altering convergence.

Acknowledgments

The authors would like to thank the referees for their valuable comments. This work is supported by the National Key R&D Program of China (2023YFB3001702), NSF of China Innovative Research Group Project (62421002), and NSF of China (No. U24B20151, 62032023, 42104078, 61902411).

References

[1] Abdou M Abdel-Rehim, Ronald B Morgan, and Walter Wilcox. 2014. Improved Seed Methods for Symmetric Positive Definite Linear Equations with Multiple Right-hand Sides. *Wiley Online Library on Numerical Linear Algebra with Applications* 21, 3 (2014), 453–471.

[2] Christie Alappat, Achim Basermann, Alan R Bishop, Holger Fehske, Georg Hager, Olaf Schenk, Jonas Thies, and Gerhard Wellein. 2020. A Recursive Algebraic Coloring Technique for Hardware-efficient Symmetric Sparse Matrix-vector Multiplication. *ACM Transactions on Parallel Computing (TOPC)* 7, 3 (2020), 1–37.

[3] Christie Alappat, Georg Hager, Olaf Schenk, and Gerhard Wellein. 2022. Level-Based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 34, 2 (2022), 581–597.

[4] Christie Alappat, Jonas Thies, Georg Hager, Holger Fehske, and Gerhard Wellein. 2025. Algebraic Temporal Blocking for Sparse Iterative Solvers on Multi-core CPUs. *SAGE on The International Journal of High Performance Computing Applications (IJHPCA)* 39, 2 (2025), 230–250.

[5] Steven F Ashby. 1991. Minimax Polynomial Preconditioning for Hermitian Linear Systems. *SIAM J. Matrix Anal. Appl.* 12, 4 (1991), 766–789.

[6] Zhaojun Bai, Dan Hu, and Lothar Reichel. 1994. A Newton Basis GMRES Implementation. *IMA J. Numer. Anal.* 14, 4 (1994), 563–581.

[7] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. 2014. Communication Lower Bounds and Optimal Algorithms for Numerical Linear Algebra. *Cambridge University Press Acta Numerica* 23 (2014), 1–155.

[8] Klaus-Jürgen Bathe. 2006. *Finite Element Procedures.* Klaus-Jurgen Bathe.

[9] Guy E Blelloch, Ioannis Koutis, Gary L Miller, and Kanat Tangwongsan. 2010. Hierarchical Diagonal Blocking and Precision Reduction Applied to Combinatorial Multigrid. In *Proceedings of the 22nd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[10] Erik G Boman, Ümit V Çatalyürek, Cédric Chevalier, and Karen D Devine. 2012. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering and Coloring. *Wiley Online Library on Scientific Programming* 20, 2 (2012), 129–150.

[11] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Ssing Compressed Sparse Blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 233–244.

[12] Erin Carson, Nicholas Knight, and James Demmel. 2013. Avoiding Communication in Nonsymmetric Lanczos-based Krylov Subspace Methods. *SIAM Journal on Scientific Computing* 35, 5 (2013), S42–S61.

[13] Umit V Catalyurek and Cevdet Aykanat. 2002. Hypergraph-partitioning-based Decomposition for Parallel Sparse-matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 10, 7 (2002), 673–693.

[14] Edmond Chow and Aftab Patel. 2015. Fine-Grained Parallel Incomplete LU Factorization. *SIAM journal on Scientific Computing* 37, 2 (2015), C169–C193.

[15] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[16] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2008. Avoiding Communication in Sparse Matrix Computations. In *Proceedings of the 22th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–12.

[17] James Demmel, Mark Frederick Hoemmen, Marghoob Mohiyuddin, and Katherine A. Yelick. 2007. *Avoiding Communication in Computing Krylov Subspaces.* Technical Report UCB/EECS-2007-123. Computer Science Division, University of California at Berkeley.

[18] Dezun Dong, Ziyu Wang, and Fei Lei. 2025. Zettafly: A Network Topology with Flexible Non-blocking Regions for Large-scale AI and HPC Systems. In *Proceedings of the 52nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 835–848.

[19] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-vector Multiplication on Modern Multi-and Many-core Processors. In *Proceedings of the 46th IEEE International Conference on Parallel Processing (ICPP)*. 292–301.

[20] Mark Embree, Jennifer A. Loe, and Ronald Morgan. 2021. Polynomial Preconditioned Arnoldi with Stability Control. *SIAM Journal on Scientific Computing* 43, 1 (2021), A1–A25.

[21] Robert D Falgout, Jim E Jones, and Ulrike Meier Yang. 2006. The Design and Implementation of Hypre, A Library of Parallel High Performance Preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. 267–294.

[22] Robert D Falgout and Ulrike Meier Yang. 2002. Hypre: A Library of High Performance Preconditioner. In *Proceedings of the Springer International Conference on Computational Science (ICCS)*. 632–641.

[23] Bernd Fischer. 2011. *Polynomial Based Iteration Methods for Symmetric Linear Systems.* SIAM.

[24] Gene H Golub and Charles F Van Loan. 2013. *Matrix Computations.* JHU press.

[25] William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. 2001. High-performance Parallel Implicit CFD. *Elsevier Parallel Computing* 27, 4 (2001), 337–362.

[26] Vicente Hernandez, Jose E Roman, and Vicente Vidal. 2005. SLEPc: A Scalable and Flexible Toolkit for The Solution of Eigenvalue Problems. *ACM Trans. Math. Software* 31, 3 (2005), 351–362.

[27] Michael A Heroux and James M Willenbring. 2012. A New Overview of The Trilinos Project. *IOS Press Scientific Programming* 20, 2 (2012), 83–88.

[28] Takeo Hoshi. [n. d.]. ELSES Matrix Library. http://www.elses.jp/matrix/

[29] Huawei. 2019. Kunpeng 920. https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920

[30] IntelLabs. 2015. Sparse Matrix Pre-processing (SpMP) library. https://github.com/IntelLabs/SpMP

[31] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. 2012. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 474–483.

[32] Olin G Johnson, Charles A Micchelli, and George Paul. 1983. Polynomial Preconditioners for Conjugate Gradient Calculations. *SIAM J. Numer. Anal.* 20, 2 (1983), 362–376.

[33] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[34] George Karypis and Vipin Kumar. 1998. Multilevel K-way Partitioning Scheme for Irregular Graphs. *Elsevier Journal of Parallel and Distributed Computing (JPDC)* 48, 1 (1998), 96—-129.

[35] George Karypis, Kirk Schloegel, and Vipin Kumar. 2003. Parmetis. *Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version* 2 (2003).

[36] Cornelius Lanczos. 1952. Chebyshev Polynomials in The Solution of Large-scale Linear Systems. In *Proceedings of the ACM National Meeting (Toronto)*. 124–132.

[37] Ruipeng Li, Yuanzhe Xi, Lucas Erlandson, and Yousef Saad. 2019. The Eigenvalues Slicing Library (EVSL): Algorithms, Implementation, and Software. *SIAM Journal on Scientific Computing* 41, 4 (2019), C393–C415.

[38] Ruipeng Li, Yuanzhe Xi, Eugene Vecharynski, Chao Yang, and Yousef Saad. 2016. A Thick-Restart Lanczos Algorithm with Polynomial Filtering for Hermitian Eigenvalue Problems. *SIAM Journal on Scientific Computing* 38, 4 (2016), A2512–A2534.

[39] Quan Liu, Ronald B Morgan, and Walter Wilcox. 2015. Polynomial Preconditioned Gmres and Gmres-dr. *SIAM Journal on Scientific Computing* 37, 5 (2015), S407–S428.

[40] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. 2016. A Synchronization-free Algorithm for Parallel Sparse Triangular Solves. In *Proceedings of the 22nd Springer European Conference on Parallel Processing (Euro-Par)*. 617–630.

[41] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-platform Sparse Matrix-vector Multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS)*. 339–350.

[42] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-vector Multiplication on X86-based Many-core Processors. In *Proceedings of the 27th ACM International conference on Supercomputing (ICS)*. 273–282.

[43] Jennifer A Loe and Ronald B Morgan. 2022. Toward Efficient Polynomial Preconditioning for GMRES. *Wiley Online Library on Numerical Linear Algebra with Applications* 29, 4 (2022), e2427.

[44] Jennifer A. Loe, Heidi K. Thornquist, and Erik G. Boman. 2020. *Polynomial Preconditioned GMRES in Trilinos: Practical Considerations for High-Performance Computing*. 35–45.

[45] Yuechen Lu and Weifeng Liu. 2023. Dasp: Specific Dense Matrix Multiply-accumulate Units Accelerated General Sparse Matrix-vector Multiplication. In *Proceedings of the 35th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–14.

[46] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. 2009. Minimizing Communication in Sparse Matrix Solvers. In *Proceedings of the 24th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[47] Jeffrey Morlan, Shoaib Kamil, and Armando Fox. 2013. Auto-tuning The Matrix Powers Kernel with SEJITS. In *Proceedings of the 10th Springer International Conference High Performance Computing for Computational Science (VECPAR)*. 391–403.

[48] Takayuki Muranushi and Junichiro Makino. 2015. Optimal Temporal Blocking for Stencil Computation. *Elsevier on Procedia Computer Science* 51 (2015), 1303–1312.

[49] Ryo Muro, Akihiro Fujii, and Teruo Tanaka. 2019. Acceleration of Symmetric Sparse Matrix-Vector Product Using Improved Hierarchical Diagonal Blocking Format. In *Proceedings of the ACM International Conference on High Performance Computing in Asia-Pacific Region (HPC-Asia)*. 63–70.

[50] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. Tilespmv: A Tiled Algorithm for Sparse Matrix-vector Multiplication on GPUs. In *Proceedings of the 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 68–78.

[51] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying Synchronization for High-performance Shared-memory Sparse Triangular Solver. In *Proceedings of the 29th Springer International Supercomputing Conference (ISC)*. 124–140.

[52] François Pellegrini and Jean Roman. 1996. Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of the Springer International Conference and Exhibition on High-Performance Computing and Networking Europe (HPCN-Europe)*. 493–498.

[53] Yousef Saad. 1985. Practical Use of Polynomial Preconditionings for The Conjugate Gradient Method. *SIAM J. Sci. Statist. Comput.* 6, 4 (1985), 865–881.

[54] Yousef Saad. 1987. Least Squares Polynomials in The Complex Plane and Their Use for Solving Nonsymmetric Linear Systems. *SIAM J. Numer. Anal.* 24, 1 (1987), 155–169.

[55] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM.

[56] Yousef Saad. 2011. *Numerical Methods for Large Eigenvalue Problems*. SIAM.

[57] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A Lightweight Performance-oriented Tool Suite for X86 Multicore Environments. In *Proceedings of the 39th IEEE International Conference on Parallel Processing Workshops (ICPPW)*. 207–216.

[58] Henk A Van der Vorst. 2003. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press.

[59] Oriol Vinyals and Daniel Povey. 2012. Krylov Subspace Descent for Deep Learning. In *Proceedings of the 15th PMLR International Conference on Artificial Intelligence and Statistics (AISTATS)*. 1261–1268.

[60] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 19th ACM/IEEE Conference on Supercomputing (SC)*. 1–12.

[61] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[62] Jan Winkelmann, Paul Springer, and Edoardo Di Napoli. 2019. ChASE: Chebyshev Accelerated Subspace Iteration Eigensolver for Sequences of Hermitian Eigenvalue Problems. *ACM Transactions on Mathematical Software (TOMS)* 45, 2 (2019), 1–34.

[63] Xiaojian Yang, Shengguo Li, Fan Yuan, Dezun Dong, Chun Huang, and Zheng Wang. 2023. Optimizing Multi-Grid Computation and Parallelization on Multi-Cores. In *Proceedings of the 37th ACM International Conference on Supercomputing (ICS)*. 227–239.

[64] Fan Yuan and Shengguo Li. 2022. YHAMG. https://gitee.com/e-level-parallel-algorithm/yhamg

[65] Fan Yuan, Xiaojian Yang, Yunqing Huang, Dezun Dong, Chuanfu Xu, Jie Liu, Xiaoqiang Yue, Shengguo Li, and Hongxia Wang. 2025. CRAMG: A Communication-reduced Algebraic Multigrid Method. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS)*. 397–411.

[66] Fan Yuan, Xiaojian Yang, Shengguo Li, Dezun Dong, Chun Huang, and Zheng Wang. 2024. Optimizing Multi-Grid Preconditioned Conjugate Gradient Method on Multi-Cores. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 35, 5 (2024), 768–779.

[67] Yichen Zhang, Shengguo Li, Fan Yuan, Dezun Dong, Xiaojian Yang, Tiejun Li, and Zheng Wang. 2023. Memory-aware Optimization for Sequences of Sparse Matrix-vector Multiplications. In *Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 379–389.

[68] Yunkai Zhou, Yousef Saad, Murilo L Tiago, and James R Chelikowsky. 2006. Parallel Self-consistent-field Calculations Via Chebyshev-filtered Subspace Acceleration. *APS Physical Review E* 74, 6 (2006), 066704.

Received 2025-08-31; accepted 2025-11-10