CRAMG: A Communication-Reduced Algebraic Multigrid Method

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

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

Yunqing Huang
School of Mathematics and Computer Science, Xiangtan University
Xiangtan, China
huangyq@xtu.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

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

Xiaoqiang Yue
School of Mathematics and Computer Science, Xiangtan University
Xiangtan, China
yuexq@xtu.edu.cn

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

Hongxia Wang*
Department of Mathematics, National University of Defense Technology
Changsha, China
wanghongxia@nudt.edu.cn

Abstract

Algebraic multigrid (AMG) is widely used to accelerate large-scale sparse linear solvers. In distributed environments, neighboring communication overhead in AMG significantly impacts overall solution time. We propose Communication-Reduced Algebraic Multigrid (CRAMG) methods to minimize inter-process data exchange and message count by fusing interpolation/restriction operators with residual computations. This reduces communication frequency from four per level to as few as two. Experiments show up to 45% reduction in data exchange and 35% fewer messages. Performance evaluations on an Intel platform demonstrate significant improvements over classical multiplicative AMG and the mult-additive version in the Hypre package.

CCS Concepts

• **Mathematics of computing** → **Computations on matrices**; **Solvers**; **Mathematical software performance**.

Keywords

Algebraic Multigrid, Communication, Multiplicative Multigrid, Additive Multigrid, Incomplete LU, Symmetric Gauss-Seidel

ACM Reference Format:
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 *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3725764

*Corresponding author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725764

1 Introduction

Algebraic multigrid (AMG) is a widely used preconditioner or solver in large-scale scientific computing and a vital component of many simulation codes. AMG enhances computational efficiency and convergence by constructing a hierarchy of grids with varying levels of granularity [11, 27].

Unlike geometric multigrid methods, AMG is highly adaptable, as it does not depend on geometric grid information. Instead, it automatically generates multilevel grid structures through matrix analysis. This makes it especially effective for large sparse matrices, significantly reducing computational complexity and maintaining fast convergence.

AMG has demonstrated exceptional efficiency on distributed memory architectures [3, 4]. However, classical AMG methods face scalability issues due to high communication complexities at the coarse levels [7, 8, 33]. The multigrid algorithm divides the grid problem hierarchically into multiple levels. As the size of the coarse grid problem decreases, the number of nonzeros per row in the coarse grid operators increases, along with the number of neighboring processes. This leads to a significant increase in communication complexity, which in turn results in a notable decrease in both performance and scalability.

Numerous strategies have been employed to reduce communication complexity and enhance communication-computation overlap in AMG. These include aggressive coarsening [15], sparsification [7], and node-aware inter-node communication techniques [8]. Recently, an additive variant of AMG, termed *mult-additive*, was introduced in [33]. This method has shown improved solve times and reduced communication costs. The classical V-cycle AMG requires four neighbor communication steps per level, as outlined in Algorithm 1. By incorporating modified operators, the mult-additive approach reduces the number of communication steps from four to three, further enhancing parallel efficiency. However, a limitation of the mult-additive method in [33] is its reliance on the sparsity of the inverse of the smoothing operator, which is challenging to achieve in practice.

In this work, we introduce the Communication-Reduced Algebraic Multigrid (CRAMG) method. CRAMG introduces a novel class of modified operators that overcome the aforementioned constraint, making them compatible with Gauss-Seidel and incomplete LU smoothers [28]. CRAMG offers two variants: CR-D and CR-M. CR-D significantly reduces inter-process data exchange, while CR-M minimizes the overall number of messages transmitted. The core innovation of CRAMG is to decrease neighbor communications in AMG by fusing interpolation or restriction operators [27] with residual vector computations. These optimizations substantially reduce the communication frequency per level to three instances, or even two when communications are merged. We present CRAMG implementations based on two smoothers: Gauss-Seidel and incomplete LU factorization. Both implementations effectively reduce the generation overhead of modified operators during the AMG setup phase, further enhancing the method's practicality.

We evaluate CRAMG using a 3D Poisson equation discretized with 7-point and 27-point stencils, a 3D diffusion-convection equation to represent asymmetric problems, and a selection of representative sparse matrices from the SuiteSparse Matrix Collection [13]. Comparisons are made against the classical AMG method and the mult-additive version in the state-of-the-art AMG library (Hypre [18]). The evaluation includes setup time, solve time, communication costs and its scalability.

We evaluated our approach on a supercomputer with 128 nodes, each having a 64-core Intel Xeon CPU, using up to 8,192 MPI processes. Results show that CRAMG can significantly reduce communication overhead: CR-D variant decreases inter-process data exchange by 41-45%, while CR-M variant reduces message count by 31-35%. For GS smoothing, CR-M performs best, achieving **1.21-1.30x** speedup over traditional multiplicative methods and **1.69-1.85x** over Hypre's mult-additive method. CR-D shows greater advantage with ILU smoothing, achieving 1.09-1.20x speedup over multiplicative methods, and exhibiting 1.29x performance improvement compared to the mult-additive method for 7-point stencil problems. Both variants of CRAMG show good weak and strong scalability in the solve and setup phases. Moreover, CRAMG proves to be effective for asymmetric problems where Hypre's mult-additive methods typically perform badly. Our tests on SuiteSparse matrices further confirm the performance advantage of CRAMG in practical problems, with a maximum speedup of 1.16x on a single node.

This paper makes the following contributions:

(1) We propose two novel communication-reducing AMG variants: CR-D for reducing data exchange and CR-M for minimizing message count, decreasing neighbor communications per level from four to three and two, respectively.
(2) We implement CRAMG with Gauss-Seidel and Incomplete LU smoothers, integrating modified operator generation with coarse grid creation to reduce setup overhead.
(3) We evaluate CRAMG against classical AMG and mult-additive methods, demonstrating significant performance improvements and scalability across various problem types and computational scales.

2 Preliminaries

This section introduces key concepts and notation related to algebraic multigrid (AMG) and the solution of sparse linear systems, represented as $Ax = b$, where A is a large sparse matrix and b is a dense vector.

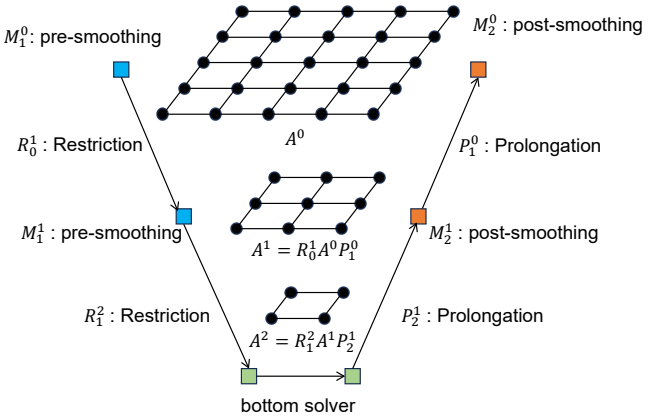


Figure 1: Flowchart of the V(1,1)-cycle MG algorithm with $\ell = 3$ levels, where V(1,1) denotes that each level of the grid uses one pre-smoothing and one post-smoothing, respectively.

2.1 Multiplicative Multigrid

Algebraic Multigrid (AMG) methods [27, 30] utilize a coarsening algorithm to recursively construct a multilevel grid hierarchy, based on the finest grid operator provided $A_0 = A$. Figure 1 shows the flowchart of a V-cycle multigrid with $\ell = 3$. Multigrid methods consist of four key operators: *smoother*, *restriction*, *bottom solver*, and *interpolation*.

For each grid $k = 0, \cdots, \ell - 1$, with grid ℓ representing the coarsest grid, we denote the following operators:

- The interpolation operator P_{k+1}^k, which transfers the solution from grid $k+1$ to k, and the restriction operator R_k^{k+1}, which transfers the residual errors from grid k to $k+1$. Typically, we have $R_k^{k+1} = (P_{k+1}^k)^T$.
- The coarse grid operator $A^{k+1} = R_k^{k+1} A^k P_{k+1}^k$ defined on grid $k+1$.
- The pre-smoothing operator M_1^k and the post-smoothing operator M_2^k applied on grid k.

With these definitions, we can recursively construct a multigrid V(1,1)-cycle operator. The notation "V(1,1)" signifies that both the pre-smoothing and post-smoothing steps, along with the coarse grid correction, each consist of a single iteration. Specifically, as detailed in [34, Eq. (2) of Sec. 2], the V(1,1)-cycle operator B^k on grid $k = 0, \cdots, \ell - 1$ can be defined recursively by,

$$\begin{aligned} B^k = {} & (M_2^k)^{-1}(M_1^k + M_2^k - A^k)(M_1^k)^{-1} \\ & + (I - (M_2^k)^{-1}A)P_{k+1}^k B^{k+1} R_k^{k+1}(I - A(M_1^k)^{-1}). \end{aligned} \tag{1}$$

In the coarsest grid ℓ, B^k is defined as $B^\ell = (A^\ell)^{-1}$.

The Algorithm 1 shows the procedure of the classical multigrid algorithm, and it forms the 'V' cycle. Just as the operator B^k is obtained recursively from multiplication, the classical multigrid algorithm is also called a multiplicative multigrid. In this configuration, each level of a multiplicative V-cycle involves four instances of neighboring communication. Specifically, two communications occur in each A^k matrix-vector multiplication, while one matrix-vector multiplication occurs in each of the P_{k+1}^k and R_k^{k+1} at level k.

Figure 3(a) illustrates the details of communication and computation scheduling of the standard AMG. The arrows indicate the execution order between computational units. Red notations signify communication that occur concurrently with the computation. The superscript denotes the grid level; for instance, "M_1^0" and "M_2^0" represent the pre-smoothing and post-smoothing on grid level 0, respectively. Subscripts are used to differentiate between local and external computations: "L" in "R_{L}^0" refers to the computation of the local component of R_0^1, while "E" denotes the computation of the external component. "H" signifies the halo exchange communication, while "H_P^k", "H_A^k" and "H_R^k" indicates the halo exchange associated with the application of P_{k+1}^k, A^k and R_k^{k+1}, respectively.

Algorithm 1 Multiplicative V(1,1)-cycle

for $k = 0, \cdots, \ell - 1$ (Sequential) **do**
  $x^k = (M_1^k)^{-1} b^k$
  $r^k = b^k - A^k x^k$ // First Residual
  $b^{k+1} = R_k^{k+1} r^k$

$x^\ell = (A^\ell)^{-1} b^\ell$

for $k = \ell - 1, \cdots, 0$ (Sequential) **do**
  $x^k := x^k + P_{k+1}^k x^{k+1}$
  $x^k := x^k + (M_2^k)^{-1}(b^k - A^k x^k)$ // Second Residual

Algorithm 2 Mult-additive V(1,1)-cycle

for $k = 0, \cdots, \ell - 1$ (Sequential) **do**
  $b^{k+1} = \overline{R}_k^{k+1} b^k$

for $k = 0, \cdots, \ell - 1$ (Parallel) **do**
  $x^k = (M_1^k)^{-1} b^k$
  $x^k := x^k + (M_2^k)^{-1}(b^k - A^k x^k)$

$x^\ell = (A^\ell)^{-1} b^\ell$

for $k = \ell - 1, \cdots, 0$ (Sequential) **do**
  $x^k := x^k + \overline{P}_{k+1}^k x^{k+1}$

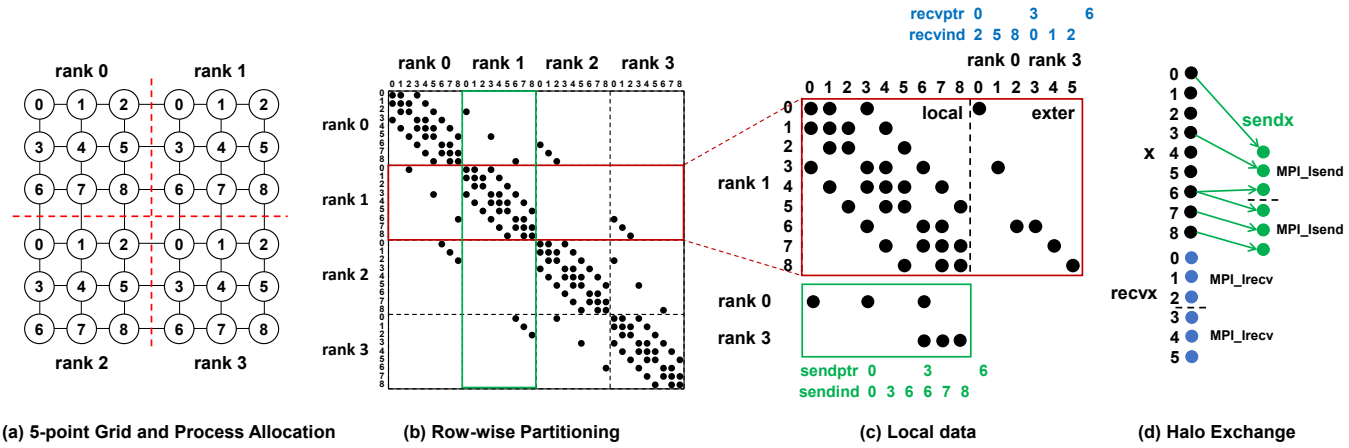


Figure 2: The row-wise distribution and the communication pattern of a 5-point matrix in process rank 1. Local data is split into the local and the external blocks. `nnb` indicates neighbor count, `nbrank` lists their ranks. `recvptr` and `recvind` show external block column indices, while `sendptr` and `sendind` detail elements for neighboring processes.

2.2 Mult-additive Multigrid Variant

A multiplicative additive variant, denoted as *mult-additive* [33], was proposed by introducing new operators *interpolation* and *restriction*. They are named smoothed interpolation and restriction operators, and are defined as follows, respectively.

- The smoothed interpolation operator:

$$\overline{P}_{k+1}^{k} = (I - (M_2^k)^{-1}A)P_{k+1}^{k}, \tag{2}$$

  and the smoothed restriction operator:

$$\overline{R}_{k}^{k+1} = R_{k}^{k+1}(I - A(M_1^k)^{-1}), \tag{3}$$

  for $k = 0, \cdots, \ell - 1$. It is assumed that both $(M_1^k)^{-1}$ and $(M_2^k)^{-1}$ are sparse, allowing $\overline{P}_{k+1}^{k}$ and $\overline{R}_{k}^{k+1}$ to be explicitly formed.
- The bi-smoothing operator:

$$\overline{M}^k = M_1^k(M_1^k + M_2^k - A^k)^{-1}M_2^k, \tag{4}$$

  and its inverse:

$$\Omega^k = (\overline{M}^k)^{-1}, \tag{5}$$

  for $k = 0, \cdots, \ell - 1$. We let $\Omega^\ell = (A^\ell)^{-1}$. ($\overline{M}^k$ is referred to as the "symmetrized" smoothing operator in [33, 34], where $M_2^k = (M_1^k)^T$ and A is symmetric).

The mult-additive V(1,1)-cycle can be derived by rewriting the multiplicative formula (1) as

$$\begin{aligned} B^k &= (M_2^k)^{-1}(M_1^k + M_2^k - A^k)(M_1^k)^{-1} \\ &\quad + (I - (M_2^k)^{-1}A)P_{k+1}^{k}B^{k+1}R_{k}^{k+1}(I - A(M_1^k)^{-1}) \\ &= \Lambda^k + \overline{P}_{k+1}^{k}B^{k+1}\overline{R}_{k}^{k+1} = \sum_{j=k}^{\ell} \overline{P}_{j}^{k}\Omega^j\overline{R}_{k}^{j}, \end{aligned} \tag{6}$$

Algorithm 3 Row-wise algorithm for $y = Ax$.

1: Begin halo exchange communication
2: Compute $y = A.\text{local} \cdot x$
3: Wait communication requests
4: Compute $y := y + A.\text{exter} \cdot \text{recvx}$

where the composite interpolation operators $\overline{P}_{j}^{k} = \overline{P}_{k+1}^{k}\overline{P}_{k+2}^{k+1}\cdots\overline{P}_{j}^{j-1}$ and the composite restriction operators $\overline{R}_{k}^{j} = \overline{R}_{j-1}^{j}\overline{R}_{j-2}^{j-1}\cdots\overline{R}_{k}^{k+1}$ for $j > k$, and $\overline{P}_{k}^{k} = \overline{R}_{k}^{k} = I$. The procedure is detailed in Algorithm 2.

Compared to the four neighboring communications for multiplicative V-cycle, the Mult-additive variant requires only three instances of neighboring communication: one for the matrix-vector multiplication for A^k, and one each for $\overline{P}_{k+1}^{k}$ and $\overline{R}_{k}^{k+1}$, respectively. An additional advantage of the Mult-additive variant is that smoothing operations are performed concurrently across all levels.

However, in the mult-additive variant, the smoothed interpolation $\overline{P}_{k+1}^{k}$ and restriction $\overline{R}_{k}^{k+1}$ obtained from the matrix-matrix multiplications tend to exhibit more filled non-zero elements, which may result in higher communication, computation and memory cost. Fortunately, these costs can be effectively managed by truncating the smoothed interpolation and restriction operators [14, 30].

2.3 Matrix-vector Multiplications

In distributed environments, popular sparse libraries such as Hypre [18], PETSc [5], and Trilinos [22] use one-dimensional distributions to take advantage of the sparsity structure of

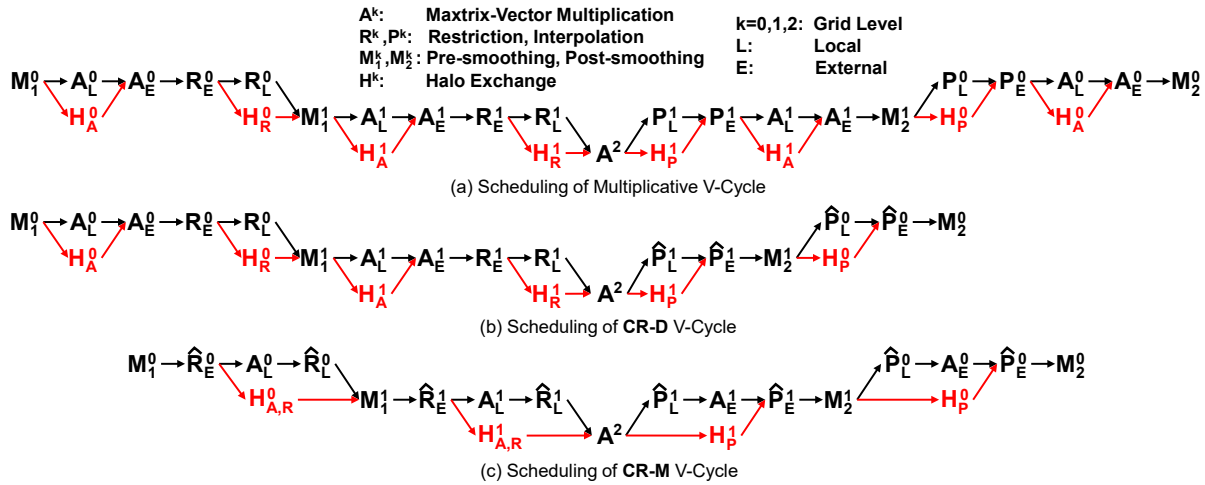


Figure 3: The communication-computation scheduling of various V-cycle variants. Arrows indicate execution order, red notations show concurrent communication. Superscripts denote grid levels (e.g., M_i^0 is smoothing on level 0). Subscripts "L" and "E" represent local and external components. "H" indicates halo exchange, while $H_{A,R}^k$ represents merged halo for A^k and $\widehat{R}_k^{k+1}$.

Algorithm 4 Column-wise algorithm for $y = Ax$.

1: Compute sendy = A.exter $\cdot\ x$
2: Begin halo exchange communication
3: Compute $y = A$.local $\cdot\ x$
4: Wait communication requests and compute $y := y +$ recvy

the input matrix. Depending on the partitioning dimension, there are primary two algorithms: row-wise algorithm and column-wise algorithm.

Row-wise algorithm. Figure 2 (a)(b) illustrate a 5-point stencil grid, and the row-wise distribution and the communication pattern of the corresponding matrix, respectively. Each process is assigned a contiguous range of rows, which are further split into the local and the external blocks by columns. The local block pertains to on-process columns, while the external block corresponds to off-process columns. External block can be compressed by columns to minimize its size. The original indices preserved in the arrays `recvptr` and `recvind`. Messages to be sent to neighbouring processes are stored in the arrays `sendptr` and `sendind`, and must be prepared during the preprocessing stage.

The row-wise algorithm for computing $y = Ax$ with overlapping communication and local computation is detailed in Algorithm 3. The halo exchange communication is initiated first using multiple non-blocking send and receive routines. This allows for concurrent execution of computations on the local block with the ongoing communication. Once the communication is complete, computations involving the external block are executed safely.

Column-wise algorithm. The column-wise algorithm exhibits characteristics similar to the row-wise algorithm. Each process is assigned a contiguous range of columns. These columns are then split into the local and the external blocks by rows.

The column-wise algorithm for $y = Ax$ differs from the row-wise algorithm. As shown in Algorithm 4, computations on the external block are performed first to obtain the result `sendy`. This is followed by halo exchange communication while the computations on the local block is executed. After completing both communication and computations, the received values `recvy` are added to their corresponding positions in y.

The primary distinction between the two algorithms is the content of the data exchanged: the row-wise algorithm requires the exchange of border data of x, and the amount of data received is related to the number of columns in the external block. In contrast, the column-wise algorithm involves exchanging border data of y, with the amount of data sent determined by the number of rows in the external block.

We examine the operators involved in V-cycle: R_k^{k+1}, P_{k+1}^k and A^k. Given that P_{k+1}^k are characterized as tall-and-skinny matrices, the row-wise algorithm generally proves more suitable for their implementation. This preference is intuitively justified, as the halo region for the coarse grid typically has smaller dimensions compared to that of the fine grid. Conversely, R_k^{k+1}, being short-and-wide matrices, are more efficiently handled using the column-wise algorithm. Thus, in

Algorithm 5 CR-D V(1,1)-cycle (Version 1)

1: **for** $k = 0, \cdots, \ell - 1$ (Sequential) **do**
2: $x^k = (M_1^k)^{-1} b^k$
3: $r^k = b^k - A^k x^k$
4: $b^{k+1} = R_k^{k+1} r^k$
5:
6: $x^\ell = (A^\ell)^{-1} b^\ell$
7:
8: **for** $k = \ell - 1, \cdots, 0$ (Sequential) **do**
9: $r^k := r^k + \widehat{P}_{k+1}^k x^{k+1}$
10: $x^k := x^k + (M_2^k)^{-1} r^k$

Algorithm 6 CR-D V(1,1)-cycle (Version 2)

1: **for** $k = 0, \cdots, \ell - 1$ (Sequential) **do**
2: $x^k = (M_1^k)^{-1} b^k$
3: $b^{k+1} = \widehat{R}_k^{k+1} x^k$
4:
5: $x^\ell = (A^\ell)^{-1} b^\ell$
6:
7: **for** $k = \ell - 1, \cdots, 0$ (Sequential) **do**
8: $x^k := x^k + P_{k+1}^k x^{k+1}$
9: $r^k := b^k - A^k x^k$
10: $x^k := x^k + (M_2^k)^{-1} r^k$

our implementation, matrices A^k and P_{k+1}^k are distributed using the row-wise partitioning, while R_k^{k+1} utilizes the column-wise distribution.

3 Communication-reduced Multiplicative AMG

In this section, we introduce two novel communication-reduced multiplicative variants designed to decrease the communication cost of multigrid V-cycles in large-scale parallel computing environments.

Our approach builds upon the standard multigrid hierarchy $\{A^k, P_{k+1}^k, R_k^{k+1}, M_1^k, M_2^k\}$, as defined in section 2.1. For level $k = 0, \cdots, \ell - 1$, we consider smoothing operators M_1^k and M_2^k derived from the following matrix splittings:

$$A^k = M_1^k - N_1^k = M_2^k - N_2^k. \tag{7}$$

For non-splitting type smoothing operators, N_1^k and N_2^k represents the difference between the smoothing operator and the system matrix. For instance, N_1^k can be defined as $N_1^k = M_1^k - A^k$.

To develop our communication-reduced multiplicative variants, we introduce the following modified operators:

- Modified smoothed interpolation operator:

$$\widehat{P}_{k+1}^k = N_2^k P_{k+1}^k. \tag{8}$$

Algorithm 7 CR-M V(1,1)-cycle

1: **for** $k = 0, \cdots, \ell - 1$ (Sequential) **do**
2: $x^k = (M_1^k)^{-1} b^k$
3: $b^{k+1} = \widehat{R}_k^{k+1} x^k$
4:
5: **for** $k = 0, \cdots, \ell - 1$ (Parallel) **do**
6: $r^k = b^k - A^k x^k$
7:
8: $x^\ell = (A^\ell)^{-1} b^\ell$
9:
10: **for** $k = \ell - 1, \cdots, 0$ (Sequential) **do**
11: $r^k := r^k + \widehat{P}_{k+1}^k x^{k+1}$
12: $x^k := x^k + (M_2^k)^{-1} r^k$

- Modified smoothed restriction operator:

$$\widehat{R}_k^{k+1} = R_k^{k+1} N_1^k. \tag{9}$$

- Modified bi-smoothing operator:

$$\widehat{\Omega}^k = M_1^k + M_2^k - A^k. \tag{10}$$

In this section, we show that the residual computations can be merged with our modified restriction and interpolation operators, respectively. When combining one residual calculation, CRAMG requires three neighbor communications, whereas it requires two neighbor communications when combining two residuals. For this purpose, we propose two optimization methods, CR-D, CR-M.

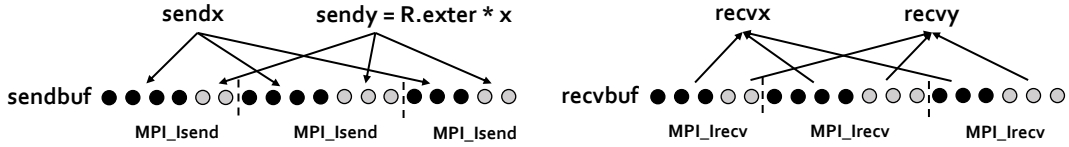
3.1 CR-D: Minimize Data Exchange

3.1.1 Fusion of Interpolation and Second Residual Computation. The CR-D variant fuses a single residual computation. Here the interpolation step is combined with the second residual calculation. The derivation of this optimization is as follows. We begin by substituting the x^k from line 8 of Algorithm 1 into line 9:

$$\begin{aligned} x^k &:= (x^k + P_{k+1}^k x^{k+1}) + (M_2^k)^{-1}(b^k - A^k(x^k + P_{k+1}^k x^{k+1})) \\ &= x^k + (M_2^k)^{-1}(b^k - A^k x^k + (M_2^k - A^k) P_{k+1}^k x^{k+1}) \\ &= x^k + (M_2^k)^{-1}(b^k - A^k x^k + N_2^k P_{k+1}^k x^{k+1}) \\ &= x^k + (M_2^k)^{-1}(r^k + \widehat{P}_{k+1}^k x^{k+1}) \end{aligned} \tag{11}$$

The last line of equation (11) introduces the modified smoothed interpolation operator $\widehat{P}_{k+1}^k$ and reuses the vector r^k from the first residual computation (line 3 in Algorithm 1).

Compared to the multiplicative V(1,1)-cycle, the CR-D variant reduces one matrix-vector multiplication and requires only three neighbor communications per level. Algorithm 5 shows the steps for the first version of the CR-D variant. Figure 3(b) illustrates the scheduling process of CRAMG in the corresponding CR-D mode. Compared to the scheduling



(a) Combined sending (b) Combined receiving

Figure 4: The combination of two halos for compute $\widehat{R}_k^{k+1}x^k$ and A^kx^k in Algorithm 7. Two messages sent to the same destination or received from the same source can be merged to one.

process of the standard AMG which is shown in Figure 3(a), our CR-D reduces the halo exchange associated with A^k on the second residual computation of V-cycle.

3.1.2 Fusion of First Residual Computation and Restriction. An alternative implementation of the CR-D model is to combine the first residual computation with the restriction step. The derivation of this combination is as follows:

$$\begin{aligned} b^{k+1} &= R_k^{k+1}(b^k - A^kx^k) = R_k^{k+1}(M_1^k - A^k)x^k \\ &= R_k^{k+1}N_1^kx^k = \widehat{R}_k^{k+1}x^k. \end{aligned} \tag{12}$$

The details of CR-D Version 2 are outlined in Algorithm 6. Both implementations of the CR-D variant are functionally equivalent. Unless stated otherwise, the subsequent sections refer to the first implementation of the CR-D variant.

3.2 CR-M: Minimize Messages

In the two implementations of CR-D, only one residual operator is fused at a time. To further reduce the number of messages, we develope the CR-M variant. As described in Algorithm 7, the CR-M variant combines the residual computations with the restriction and interpolation operators.

Figure 3(c) illustrates the details of our communication and computation scheduling for CR-M. To maximize communication-computation overlap, we strategically schedule the computation of A_{L}^k before R_{L}^k, thereby enhancing its overlap with the $H_{A,R}^k$ communication. Similarly, the computation of A_{E}^k is scheduled after P_{L}^k to optimize its overlap with H_P^k. Compared to the standard multiplicative version and the CR-D variants, CR-M requires only two neighbor communications per level.

Although this variant still requires one residual computation (as shown in line 6 of Algorithm 7), we show that the communication for $\widehat{R}_k^{k+1}x^k$ can be efficiently combined with that of A^kx^k. When these two messages are sent to the same destination or received from the same source, they can be merged, significantly reducing the total number of messages exchanged. The details of this merging process are illustrated in Figure 4.

CR-M for Gauss-Seidel. CR-M can be further modified for some split smoothers. Specifically, by introducing $z^k = M_2^kx^k + r^k$ in line 6 of Algorithm 7, we derive:

$$\begin{aligned} z^k &= M_2^kx^k + r^k = M_2^kx^k + b^k - A^kx^k \\ &= (M_2^k + M_1^k - A^k)x^k = \widehat{\Omega}^kx^k. \end{aligned} \tag{13}$$

Lines 11 and 12 of Algorithm 7 can then be reformulated as:

$$\begin{aligned} x^k &:= x^k + (M_2^k)^{-1}(r^k + \widehat{P}_{k+1}^kx^{k+1}) \\ &= x^k + (M_2^k)^{-1}(-M_2^kx^k + z^k + \widehat{P}_{k+1}^kx^{k+1}) \\ &= (M_2^k)^{-1}(z^k + \widehat{P}_{k+1}^kx^{k+1}). \end{aligned} \tag{14}$$

This reformulation leads to another variant, detailed in Algorithm 8. The algorithm replaces the computation of $b^k - A^kx^k$ with $\widehat{\Omega}^kx^k$, offering notable benefits for specific split smoothers.

For example, consider forward Gauss-Seidel as the pre-smoother $M_1^k = D^k + E^k$, and backward Gauss-Seidel as the post-smoother $M_2^k = D^k + F^k$, where D^k, E^k, and F^k denote the diagonal, strictly lower triangular, and strictly upper triangular components of A^k, respectively. This configuration is the default smoother option in BoomerAMG from Hypre-2.31.0.

In this case, the modified bi-smoothing operator $\widehat{\Omega}^k$ reduces to:

$$\widehat{\Omega}^k = M_1^k + M_2^k - A^k = D^k. \tag{15}$$

Consequently, Algorithm 8 achieves significant improvement over Algorithm 7, nearly eliminating one matrix-vector multiplication operation.

3.3 Comparison to Mult-additive variant

Although the CR-D variant (Algorithm 5) and the multi-additive variant (Algorithm 2, proposed in [33]) share notable similarities, their key distinction lies in the choice of modified interpolation and restriction operators.

According to equation (3), the smoothed restriction operator $\overline{R}_k^{k+1}$ is defined as the product of the restriction operator R_k^{k+1} and the pre-smoothing iteration term $I - A^k(M_1^k)^{-1}$. If the smoothing operators are defined based on matrix splittings (see equation (7)), $\overline{R}_k^{k+1}$ and $\widehat{R}_k^{k+1}$ are related by the following expression:

$$\overline{R}_k^{k+1} = \widehat{R}_k^{k+1}(M_1^k)^{-1}. \tag{16}$$

Algorithm 8 CR-M V(1,1)-cycle for Gauss-Seidel

for $k = 0, \cdots, \ell - 1$ (Sequential) **do**
  $x^k = (M_1^k)^{-1} b^k$
  $b^{k+1} = \widehat{R}_k^{k+1} x^k$

for $k = 0, \cdots, \ell - 1$ (Parallel) **do**
  $z^k = \widehat{\Omega}^k x^k$

$x^\ell = (A^\ell)^{-1} b^\ell$

for $k = \ell - 1, \cdots, 0$ (Sequential) **do**
  $z^k := z^k + \widehat{P}_{k+1}^k x^{k+1}$
  $x^k := (M_2^k)^{-1} z^k$

Similarly, from equation (2), the smoothed interpolation operator $\overline{P}_{k+1}^k$ is the product of the post-smoothing iteration term $I - A^k (M_2^k)^{-1}$ and the interpolation operator P_{k+1}^k, and it follows that

$$\overline{P}_{k+1}^k = (M_2^k)^{-1} \widehat{P}_{k+1}^k. \tag{17}$$

For Jacobi and polynomial-type preconditioners (where $(M_j^k)^{-1}$ can be expressed as a matrix polynomial of A^k, $j = 1, 2$), explicitly computing $\overline{R}_k^{k+1}$ or $\overline{P}_{k+1}^k$ is feasible, albeit with significant memory requirements. However, for preconditioners like Gauss-Seidel and incomplete LU factorization (ILU), the situation becomes considerably more complex. These methods require applying $(M_j^k)^{-1}$ to a sparse matrix, which can lead to fill-in phenomena and result in computational complexity that is difficult to predict in advance.

In contrast, our approach replaces $\overline{R}_k^{k+1}$ and $\overline{P}_k^{k+1}$ with $\widehat{R}_k^{k+1} = R_k^{k+1}(M_1^k - A^k)$ and $\widehat{P}_k^{k+1} = (M_2^k - A^k) P_{k+1}^k$, effectively circumventing the aforementioned computational challenges. Furthermore, we introduce an efficient method for applying $\widehat{P}_k^{k+1}$ and $\widehat{R}_k^{k+1}$ by reordering the smoothing operations to overlap computation with communication, as shown in Figures 3 and 4.

4 Use of Block Jacobi Smoother

In our AMG implementation, we employ block-Jacobi-like methods as both the smoother and the coarsest grid solver. This approach has been demonstrated to effectively achieve a good balance between the acceleration of AMG methods and parallel performance [2, 37]. Moreover, the choice of smoother is intrinsically linked to the generation of the modified smoothed interpolation operator $\widehat{P}_{k+1}^k$.

We partition matrix A^k into $p \times p$ blocks. Let A_{ii}^k denote the i-th diagonal block of A^k, and A_{ofd}^k represent the matrix of off-diagonal block. Thus, A^k can be decomposed as

$$A^k = A_{\text{dia}}^k + A_{\text{ofd}}^k. \tag{18}$$

with $A_{\text{dia}}^k = \text{diag}(A_{11}^k, A_{22}^k, ..., A_{pp}^k)$.

In this context, the smoother M^k is also chosen to be a block diagonal matrix:

$$M^k = \text{diag}(M_{11}^k, M_{22}^k, ..., M_{pp}^k). \tag{19}$$

We consider two types of smoothers as follows.

4.1 Block Jacobi Gauss-Seidel

4.1.1 Smoothing. When applying Gauss-Seidel relaxation in a V-cycle, it is common to use different smoothers for pre-smoothing and post-smoothing. Specifically, the pre-smoother M_1^k, which corresponds to forward Gauss-Seidel, is defined as the block diagonal matrix with

$$(M_1^k)_{ii} = D_{ii}^k - E_{ii}^k, \tag{20}$$

and the post-smoother M_2^k, corresponding to backward Gauss-Seidel, is defined as the block diagonal matrix with

$$(M_2^k)_{ii} = D_{ii}^k - F_{ii}^k, \tag{21}$$

where $A_{ii}^k = D_{ii}^k - E_{ii}^k - F_{ii}^k$, with D_{ii}^k representing the diagonal part, E_{ii}^k the strict lower triangular part, and F_{ii}^k the strict upper triangular part of A_{ii}^k. To preserve the symmetry of the V-cycle operator, we use a symmetric Gauss-Seidel relaxation at the coarsest level. B^ℓ is defined as the block diagonal matrix with

$$B_{ii}^\ell = (D_{ii}^\ell - F_{ii}^\ell)^{-1} D_{ii}^\ell (D_{ii}^\ell - E_{ii}^\ell)^{-1}. \tag{22}$$

The implementation of CR-M V-cycle for Gauss-Seidel smoothing is very efficient. This is because $\widehat{\Omega}^k$ can be expressed as

$$\widehat{\Omega}^k = M_1^k + M_2^k - A^k = D^k - A_{\text{ofd}}^k, \tag{23}$$

where the off-diagonal non-zero elements on processor are eliminated.

Furthermore, we can introduce a relaxation factor ω, defining $(M_1^k)_{ii} = \frac{1}{\omega} D_{ii}^k - E_{ii}^k$ and $(M_2^k)_{ii} = \frac{1}{\omega} D_{ii}^k - F_{ii}^k$. This leads to SOR smoothing, providing a more comprehensive range of choices. This part of the content is beyond the scope of this paper.

4.1.2 Generation of modified smoothed interpolation operator. Typically, the Galerkin product for coarse grid generation, $A^{k+1} = R_k^{k+1} A^k P_{k+1}^k$, involves two sparse matrix-matrix multiplications: $Z^k = A^k P_{k+1}^k$ and $A^{k+1} = R_k^{k+1} Z^k$, utilizing an intermediate matrix Z^k. To enhance efficiency, the generation of the modified smoothed interpolation operator, $\widehat{P}_{k+1}^k = N_2^k P_{k+1}^k$, can be combined with $A^k P_{k+1}^k$.

According to equation (21), we can get

$$N_2^k = \text{diag}(E_{11}^k, ..., E_{pp}^k) - A_{\text{ofd}}^k. \tag{24}$$

The computations are executed using a row-wise algorithm. For each row, the process initially computes $N_2^k P_{k+1}^k$ and stores the result in $\widehat{P}_{k+1}^k$. Subsequently, the remaining part

$M_2^k P_{k+1}^k$ is computed to obtain Z^k for that row. This approach reveals that generating $\widehat{P}_{k+1}^k$ incurs no additional computational cost beyond storing the results.

4.2 Block Jacobi ILU(0)

4.2.1 Smoothing. Alternatively, we consider using the incomplete LU factorization with zero fill-in (ILU(0)) as a smoother, which often offers greater robustness compared to the Gauss-Seidel method. For a V-cycle applying block Jacobi ILU(0), pre-smoother M_1^k and post-smoother M_2^k is defined as the block diagonal matrices with

$$(M_1^k)_{ii} = (M_2^k)_{ii} = L_{ii}^k U_{ii}^k, \tag{25}$$

where L_{ii}^k and U_{ii}^k are the ILU(0) factors of A_{ii}^k.

4.2.2 Generation of modified smoothed interpolation operator. For a block Jacobi ILU(0) smoother, the matrix N_2^k is defined as

$$N_2^k = \mathrm{diag}((N_2^k)_{11}, ..., (N_2^k)_{pp}) - A_{\mathrm{ofd}}^k, \tag{26}$$

where $(N_2^k)_{ii} = L_{ii}^k U_{ii}^k - A_{ii}^k$, for $i = 1, ..., p$. L_{ii}^k and U_{ii}^k represent the ILU(0) factors of A_{ii}^k, ensuring that the elements of $(N_2^k)_{ii}$ are zero in locations corresponding to the nonzero elements of A_{ii}^k. This characteristic allows for reduced computation and memory usage.

The computational cost for $N_2^k P_{k+1}^k$ and $A^k P_{k+1}^k$ can be further optimized by leveraging the shared component A_{ofd}^k between N_2^k and A^k. Moreover, $N_2^k P_{k+1}^k$ incurs no additional communication costs, as all data exchanges are encompassed in the computation of $A_{\mathrm{ofd}}^k P_{k+1}^k$.

4.3 Truncation

Moreover, CR-M additionally requires $\widehat{R}_k^{k+1}$, presenting a more challenging computational task. For symmetric (or slightly nonsymmetric) matrices, the optimal approach involves first computing $\widehat{P}_{k+1}^k$, followed by $\widehat{R}_k^{k+1} = (\widehat{P}_{k+1}^k)^T$. This method incurs minimal additional computational cost. In contrast, for asymmetric matrices, the generation of $\widehat{R}_k^{k+1}$ may necessitate computing the matrix-matrix product $R_k^{k+1} N_k^1$, which can be computationally expensive.

The matrix $\widehat{P}_{k+1}^k$ typically exhibits a large template, potentially leading to increased communication costs [33]. A common approach to address this issue is to truncate $\widehat{P}_{k+1}^k$, using techniques similar to those employed for truncating the interpolation matrix P_{k+1}^k [14, 30].

- Selecting a truncation factor θ and excluding weights with an absolute value less than this factor, i.e., those for which $|(\hat{P}_{k+1}^k)_{ij}| < \theta$;
- Restricting the number of coefficients per row by retaining only the k_{max} weights with the highest absolute values.

Table 1: Memory and flops estimates for various V-cycle variants.

	memory	flops
Mult.GS	$\sum_{k=0}^{\ell-1}[\mathrm{nnz}(A^k) + \mathrm{nnz}(R_k^{k+1}) + \mathrm{nnz}(P_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[4\mathrm{nnz}(A^k) + 2\mathrm{nnz}(R_k^{k+1}) + 2\mathrm{nnz}(P_{k+1}^k)]$
CR-D.GS	$\sum_{k=0}^{\ell-1}[\mathrm{nnz}(A^k) + \mathrm{nnz}(R_k^{k+1}) + \mathrm{nnz}(\widehat{P}_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[3\mathrm{nnz}(A^k) + 2\mathrm{nnz}(R_k^{k+1}) + 2\mathrm{nnz}(\widehat{P}_{k+1}^k)]$
CR-M.GS	$\sum_{k=0}^{\ell-1}[\mathrm{nnz}(A^k) + \mathrm{nnz}(\widehat{R}_k^{k+1}) + \mathrm{nnz}(\widehat{P}_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[2\mathrm{nnz}(A^k) + 2\mathrm{nnz}(\widehat{R}_k^{k+1}) + 2\mathrm{nnz}(\widehat{P}_{k+1}^k)]$
Mult.ILU	$\sum_{k=0}^{\ell-1}[2\mathrm{nnz}(A^k) + \mathrm{nnz}(R_k^{k+1}) + \mathrm{nnz}(P_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[8\mathrm{nnz}(A^k) + 2\mathrm{nnz}(R_k^{k+1}) + 2\mathrm{nnz}(P_{k+1}^k)]$
CR-D.ILU	$\sum_{k=0}^{\ell-1}[2\mathrm{nnz}(A^k) + \mathrm{nnz}(R_k^{k+1}) + \mathrm{nnz}(\widehat{P}_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[6\mathrm{nnz}(A^k) + 2\mathrm{nnz}(R_k^{k+1}) + 2\mathrm{nnz}(\widehat{P}_{k+1}^k)]$
CR-M.ILU	$\sum_{k=0}^{\ell-1}[2\mathrm{nnz}(A^k) + \mathrm{nnz}(\widehat{R}_k^{k+1}) + \mathrm{nnz}(\widehat{P}_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[6\mathrm{nnz}(A^k) + 2\mathrm{nnz}(\widehat{R}_k^{k+1}) + 2\mathrm{nnz}(\widehat{P}_{k+1}^k)]$

For the truncation of $\widehat{R}_k^{k+1}$, we consider two scenarios. When $\widehat{R}_k^{k+1}$ is obtained via $\widehat{R}_k^{k+1} = (\widehat{P}_{k+1}^k)^T$, no additional truncation is necessary. Alternatively, if $\widehat{R}_k^{k+1}$ is computed using $\widehat{R}_k^{k+1} = R_k^{k+1} N_1^k$, the same truncation methods can be applied to $(\widehat{R}_k^{k+1})^T$. Adjusting the truncation parameters is crucial to prevent an increase in the number of iterations for convergence, which could negate the benefits of truncation. Through numerical experiments, we established that limiting the matrix to 24 nonzeros per row optimally preserves numerical stability across most problem while maintaining sparsity. This value serves as our default truncation threshold.

5 Experimental Setup

5.1 Evaluation Platforms

The experiments in this section were conducted on a 128-node cluster, with each node equipped with dual Intel Xeon Platinum 8358P CPUs, offering 64 cores per node at 2.60 GHz. This configuration provides a total of 8,192 cores across the cluster. For the software environment, we employed MPICH-4.1.2 as the MPI library and Intel Compiler 23.0.

5.2 Experiment Details

We implemented the CRAMG variants in the YHAMG algebraic multigrid package [19, 40], incorporating block Jacobi Gauss-Seidel and block Jacobi ILU(0) smoothers. The AMG

Table 2: Comparison of CR variants and the multiplicative method for the 7-point stencil problem with a $100 \times 100 \times 100$ cube per core, using 8,192 cores.

	setup time	solve time	iter	cycle time	memory	flops	#msgs	data exch.
Mult.GS	3.0797	7.5661	27	0.1785	1.0000	1.0000	1.0000	1.0000
CR-D.GS	3.4551	6.8150	28	0.1563	1.2364	0.9648	0.7662	0.5892
CR-M.GS	3.6190	6.2455	27	0.1400	1.4722	0.9399	0.6844	0.6304
Mult.ILU	4.4598	9.8300	26	0.2607	1.0000	1.0000	1.0000	1.0000
CR-D.ILU	5.8600	8.1711	27	0.2176	1.2819	0.9389	0.7672	0.5883
CR-M.ILU	6.2995	8.5245	26	0.2379	1.5634	1.1452	0.6861	0.6286
Hypre.wJ	7.1656	12.3423	39	/	/	/	/	/
Hypre.MA	8.2322	10.552	39	/	/	/	/	/

Table 3: Comparison of CR variants and the multiplicative method for the 27-point stencil problem with a $80 \times 80 \times 80$ cube per core, using 8,192 cores.

	setup time	solve time	iter	cycle time	memory	flops	#msgs	data exch.
Mult.GS	2.0377	5.8727	25	0.1404	1.0000	1.0000	1.0000	1.0000
CR-D.GS	2.2833	5.3332	26	0.1188	1.1275	0.8436	0.7367	0.5503
CR-M.GS	2.3163	4.5252	25	0.0993	1.2551	0.6995	0.6462	0.5738
Mult.ILU	2.3067	9.8510	24	0.2557	1.0000	1.0000	1.0000	1.0000
CR-D.ILU	3.2952	9.0409	25	0.2120	1.1639	0.8037	0.7379	0.5491
CR-M.ILU	3.3810	9.4042	25	0.2256	1.3277	0.9168	0.6480	0.5714
Hypre.wJ	2.9102	13.1632	34	/	/	/	/	/
Hypre.MA	3.5859	8.3708	32	/	/	/	/	/

configuration employed HMIS coarsening and ext+i interpolation using matrix-matrix multiplication (truncated to a maximum of 4 elements per row), along with one level of aggressive coarsening.

For CRAMG, we applied additional truncation to the modified smoothed interpolation operator, limiting it to a maximum of 24 elements per row. In the case of symmetric systems, we compute the transpose of the modified smoothed interpolation operator to derive the modified smoothed restriction operator. This configuration has been found to yield optimal performance across our test cases.

To make a fair comparison, we refined the data structure for Gauss-Seidel smoothing by partitioning the local matrix into upper triangular, lower triangular, and diagonal components. In multiplicative and CR-D methods, we rewrote the matrix-vector multiplication following pre-smoothing to reduce floating-point operations, leveraging the equation:

$$r^k = b^k - A^k x^k = N_1^k x^k. \tag{27}$$

This optimization is made possible by the condition $M_1^k x^k = b^k$. For post-smoothing in multiplicative method, we utilized the more common matrix split iteration:

$$x^k := (M_2^k)^{-1}(b^k + N_2^k x^k). \tag{28}$$

Based on these details, we provide the memory and flops estimates for various V-cycle methods in Table 1.

5.3 Evaluation Methodology

To simulate a more communication-intensive scenario, we employed a configuration of 64 processes with one thread per node, utilizing up to 8,192 processes in total. In addition to runtime, our evaluation encompassed several key metrics for each variant: memory requirements, floating-point operations (flops), number of messages exchanged, and total volume of data transferred. We introduced a change factor [33], calculated by dividing the value of the CR variant by the value of the multiplicative method. A change factor less than *one* indicates an improvement. For memory and flops, the change factors were evaluated based on the data presented in Table 1. Our analysis consisted of comprehensive weak and strong scalability studies for 7- and 27-point stencil problems. We also examined the impact of asymmetry on CR methods through convection diffusion problem tests. Furthermore, we expanded our testing to matrices from various fields in the SuiteSparse collection and compared our methods against the mult-additive approach (**Hypre.MA**) in the Hypre v2.31.0. We tested BoomerAMG using a weighted Jacobi smoother (**Hypre.wJ**) with $\omega = 0.75$, as the multi-additive method for GS smoothing is not yet supported in the latest Hypre version. The selected parameters are consistent with those used in our YHAMG implementation.

In our experiments, AMG is primarily used as a preconditioner for the CG method in symmetric systems, while for

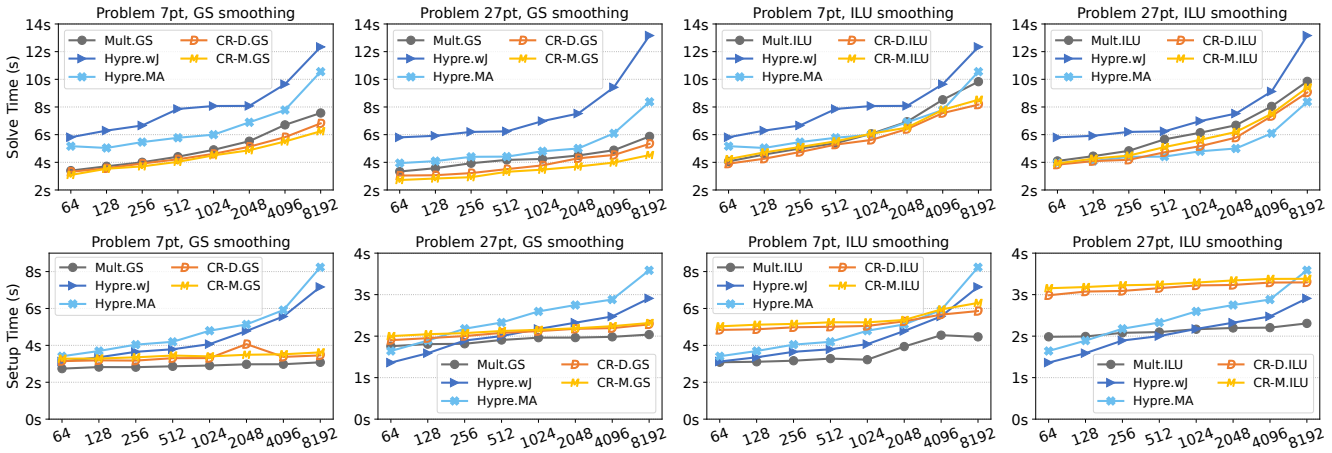


Figure 5: Comparison of various AMG methods for Problem 7pt and Problem 27pt with up to 8192 cores. For Problem 7pt, each core processes a $100 \times 100 \times 100$ cube. For Problem 27pt, an $80 \times 80 \times 80$ cube is used per core. All algorithms employ the stopping criterion $\|b - Ax\|_2 < 10^{-12}\|b\|_2$. (Above: solve phase, Below: setup phase)

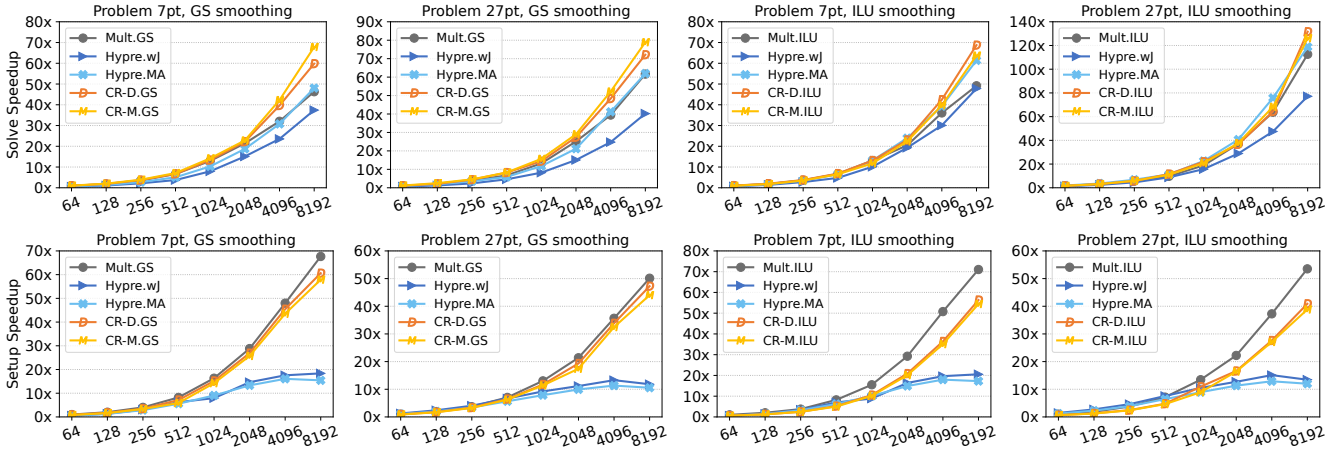


Figure 6: Comparison of various V-cycle for Problem 7pt and Problem 27pt, over multiplicative method on a single node with 64 cores, utilizing up to 8192 cores. For Problem 7pt, processing an $800 \times 800 \times 800$ cube. For Problem 27pt, processing a $640 \times 640 \times 640$ cube. (Above: solve phase, Below: setup phase)

nonsymmetric systems, we employ AMG as a preconditioner for GMRES with a restart parameter of 40.

6 Experimental Results

6.1 Large-scale Performance Analysis

In this section, we examine the 3D Poisson equation $-\Delta u = f$ with Dirichlet boundary conditions $u = 0$. Our experiments focus on two problems:

Problem 7pt: This problem is discretized using second-order central finite differences on a uniform grid, resulting in a symmetric positive definite linear system with a 7-point stencil.

Problem 27pt: This problem is also symmetric positive definite but exhibits a denser sparsity pattern with 27-point stencil.

Tables 2 and 3 compare CR variants with the multiplicative method for Problems 7pt (100^3 cube per core) and Problems 27pt (80^3 cube per core), respectively. The right-hand side vector b is computed as $b = Ae$, where e is a vector of all ones. Both experiments use AMG as a CG preconditioner on 8,192 cores, with a stopping criterion of $|b-Ax|_2 < 10^{-12}|b|_2$. We report AMG setup and solve times, iteration count, time per cycle, and change factors for memory usage, flops, message count, and data exchange. The CR variants converge at about the same rate as the standard multiplication method, whereas the Hypre methods converge at a much slower rate.

Table 4: Comparison of CR variants and the multiplicative method for Problem diffconv with a $100 \times 100 \times 100$ cube per core, using 4,096 cores. Results are recorded for a = 1, 10, and 100.

	$a = 1$			$a = 10$			$a = 100$		
	setup time	solve time	iter	setup time	solve time	iter	setup time	solve time	iter
Mult.GS	3.0662	7.4444	25	3.0481	8.6390	28	3.0418	8.3739	27
CR-D.GS	3.3579	6.8828	25	3.3480	7.5412	28	3.3681	7.4651	27
CR-M(T).GS	3.5035	6.5015	25	3.5504	8.4132	32	3.5922	36.075	138
CR-M.GS	4.4479	6.4815	25	4.5033	7.2449	28	4.5125	7.1181	27
Mult.ILU	4.2105	9.5727	23	4.1715	9.7014	26	4.1997	8.2471	21
CR-D.ILU	5.9140	8.2885	23	5.9148	8.6646	26	5.9576	7.3437	21
CR-M(T).ILU	6.1981	8.8625	23	6.2268	10.764	30	6.2444	18.756	53
CR-M.ILU	7.9157	8.8595	23	8.0556	9.1361	25	8.0925	8.0992	22
Hypre.wJ	5.7392	18.881	38	5.6222	25.561	51	5.8287	30.510	61
Hypre.MA	6.1646	14.460	39	6.0960	21.650	58	6.3992	48.957	129

CR methods demonstrate significant improvements in communication efficiency. CR-D achieves the smallest data exchange, reducing it by **41-45%** compared to the multiplicative method, while also decreasing message count by 23-26%. CR-M excels in minimizing message count, with a **31-35%** reduction compared to the multiplicative method, and reduces data exchange by 37-43%. Furthermore, the solve time analysis reveals that CR-M performs best for GS smoothing, with a speedup of **1.21-1.30x** over the classical multiplication method and **1.69-1.85x** over Hypre's mult-additive method. In contrast, CR-D demonstrates a more pronounced advantage when employing ILU smoothing, achieving speedups of **1.09-1.20x** over the multiplicative method. For the 7-point problem, CR-D exhibits a **1.29x** performance improvement compared to the mult-additive method. While in the case of the 27-point problem, the mult-additive method outperforms CR-D.

Both CR variants lead to an increase in setup time. Compared to the multiplicative method, the setup time of CR-D increases by 12.2% for GS smoothing and 31.4% for ILU. CR-M requires even more setup time, increasing by 17.5% for GS and 41.3% for ILU. Detailed results are presented in Tables 2 and 3. However, in scenarios where multiple solutions with different right-hand sides or preconditioner are used across time steps [21, 41], the initial computational investment can yield significant benefits in subsequent iterations, potentially enhancing overall performance.

6.2 Weak and Strong Scaling Tests

Figure 5 presents the weak scalability test results of various AMG methods. The evaluation is conducted by solving Problem 7pt with a $100 \times 100 \times 100$ cube per core and Problem 27pt with an $80 \times 80 \times 80$ cube per core, utilizing up to 8,192 cores. Analysis of the results indicates that as the core count increases, the solve time of CRAMG grows at a slower rate compared to the multiplicative method, suggesting superior parallel efficiency of CRAMG. Concurrently, the setup time of CRAMG increases also slowly with the increase in core count, indicating that CRAMG maintains robust scalability in the setup phase as well. It is noteworthy that our method outperforms Hypre's mult-additive method in almost all cases (except for the 27-point problem, where Mult-additive outperforms the CR method with ILU smoothing).

Figure 6 illustrates the strong scalability results for various methods. This evaluation involves solving Problem 7pt on a $800 \times 800 \times 800$ cube and Problem 27pt on a $640 \times 640 \times 640$ cube, utilizing up to 8,192 cores. As the number of kernels increases, the CR method significantly outperforms both the multiplicative and multi-additive methods in the solve phase; while in the setup phase, the CRAMG method maintains almost the same speedup ratio as the multiplicative method in GS smoothing processing, while significantly outperforming the Hypre's methods.

6.3 Diffusion Convection Problem

In this section, we examine the following problem:

Problem convdiff: This is a 3D diffusion-convection equation $-\Delta u + a\nabla \cdot u = f$ with Dirichlet boundary conditions. The diffusion part is discretized using central finite differences, while upwind finite differences are used for the advection term. This results in an asymmetric problem. The degree of asymmetry depends on the value of a. As a increases, the asymmetry of the problem increases accordingly. When $a = 0$, the problem reduces to Problem 7pt.

We demonstrate that the CR methods are effective in solving asymmetric problems. For such cases, CR-M requires additional computation of $\widehat{R}_{k+1}^k$ during the AMG setup phase. In weakly asymmetric scenarios, $\widehat{R}_{k+1}^k$ can be approximated by $(\widehat{P}_k^{k+1})^T$ to reduce computational overhead. We examined the impact of asymmetry on CR methods, include setup time increment, through Problem convdiff tests with a =

Table 5: Sparse matrices from SuiteSparse.

#	matrix	n	nnz	symm.	smooth
A	thermal2	1,228,045	8,580,313	yes	GS
B	atmosmodd	1,270,432	8,814,880	no	GS
C	atmosmodl	1,489,752	10,319,760	no	GS
D	Transport	1,602,111	23,487,281	no	GS
E	Serena	11,391,349	64,131,971	yes	SOR
F	ecology2	999,999	4,995,991	yes	ILU
G	cage14	1,505,785	27,130,349	no	ILU
H	CoupCons3D	416,800	17,277,420	no	ILU

1, 10, and 100 (representing increasing asymmetry). In this context, we introduce CR-M(T), which denotes the CR-M method that employs $(\widehat{P}_k^{k+1})^T$ as an approximation for $\widehat{R}_{k+1}^k$. The experimental results of our investigation are presented in Table 4. The solution method employed AMG as a preconditioner for the GMRES with restart 40. We conducted evaluations on a $100 \times 100 \times 100$ cube per core, utilizing 4,096 cores. The solution process employed a stopping criterion of $\|b - Ax\|_2 < 10^{-12}\|b\|_2$. The results in Table 4 show that CR-D and CR-M effectively solve asymmetric problems. For $a = 1$, CR-M(T) maintains convergence speed comparable to CR-M, demonstrating the feasibility of the approximation in weakly asymmetric cases. As asymmetry increases, with $a = 10$, CR-M(T) shows a slight increase in iteration count compared to CR-M. However, at $a = 100$, CR-M(T) shows significantly more iterations, indicating the approximation's limitations for highly asymmetric problems. The mult-additive method also struggles with high asymmetry, which requires more iterations than other variants.

6.4 Matrices from Suitesparse

We have selected a number of sparse matrices from SuiteSparse [13] that are suitable for AMG solution methods to assess the practical feasibility and performance of CR variants in the real world. The full list of selected matrices is given in Table 5.

This experiments were conducted on a single compute node equipped with dual Intel Xeon Gold 6348 processors at 2.6GHz, for a total of 56 physical cores. We utilized all 56 cores by running 56 MPI processes. To optimize load balancing and minimize communication overhead, we employed ParMETIS for matrix partitioning [23]. For matrices that do not include a predefined right-hand side vector b, we generate b by computing $b = Ae$, where e is a vector of all ones. We use AMG as the preconditioner and use CG for symmetric matrices and GMRES with restart 40 for nonsymmetric matrices. For each matrix, we selected an appropriate smoother from among GS, SOR, or ILU. The solution process adheres to the stopping criterion $\|b - Ax\|_2 < 10^{-12}\|b\|_2$.

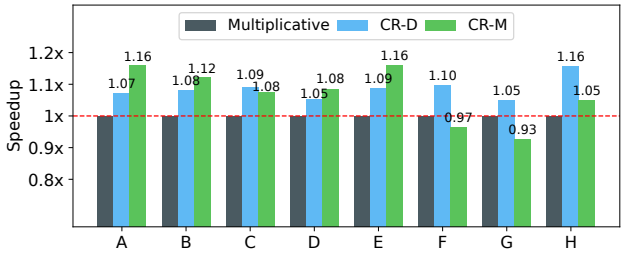


Figure 7: Performance of various V-cycle variants on a single node.

The experimental results are presented in Figure 7. When using GS or SOR smoothing, CR methods consistently exhibit better speedup compared to the multiplicative method. The CR-M variant generally achieves the highest speedup, closely followed by the CR-D variants. The CR-M variant typically achieves a speedup of 1.05x to 1.16x over the multiplicative method. Although CR-D shows slightly lower speedup than CR-M, it still outperforms the classical multiplicative version, with speedups ranging from 1.03x to 1.09x. For ILU smoothing, CR-M may sometimes be slower than the multiplicative version, but CR-D consistently outperforms it, with notable speedups of up to 1.16x on matrix H.

7 Related Works

Many methods have been proposed to reduce communication costs in AMG, and most of them focus on a redesign of the method or on the underling sparse matrix operations. Aggressive coarsening [14, 15, 38] reduces the density of coarse levels and therefore the communication cost. Similarly, the smoothed aggregation approach coarsens a large number of fine grids into a single coarse grid [31, 32]. Some methods systematically remove entries on coarse grids, leading to improved commmunication cost [7, 17]. The additive version of AMG [10] is revisited to improve parallelism and communication-computation overlapping [20, 33] and reduce synchronization [36]. Communication-avoiding techniques can also be used to improve the performance of multigrid methods, for example s-step Krylov subspace methods is used as the bottom solver of geometric multigrid in [35].

Topology-aware methods and message agglomeration are also used to reduce the AMG communication costs [25, 29]. Some node-aware communication approaches are used to reduce the number and size of internode messsages [8, 9]. Neighborhood collectives operations [24] in MPI 3/4 have been used for optimizing the irregular communications in AMG [12]. CRAMG's communication optimizations have wide-ranging potential in distributed memory systems, particularly for solving large-scale sparse linear systems and in heterogeneous computing environments where efficient communication is critical.

Sparse matrix-vector multiplication (SpMV) is a critical kernel in multigrid methods. At the algorithmic level, kernel fusion techniques have been developed to jointly optimize SpMV with related operations [39, 40, 42], thereby reducing memory bandwidth usage and access latency while improving overall computational efficiency. For matrices with symmetry or other specific structural properties, tailored optimization strategies have been proposed [1, 26], including the use of compressed storage formats and matrix reordering techniques, to further enhance computational performance. Moreover, heterogeneous optimization of SpMV has emerged as a research hotspot [6, 16], focusing on exploiting the parallel computing capabilities and memory hierarchy characteristics of hardware accelerators while efficiently distributing tasks and data across different computational units. In this work, we optimize SPMV by merging it with other operators to reduce communication costs, and the previous optimizations can be combined with our strategy.

The additional costs largely depend on the choice of smoothing operators. Vassilevski and Yang [33] analyzed l_1 Jacobi and l_1 Gauss-Seidel smoothers, showing that the l_1 Jacobi smoother requires about twice the memory of the multiplicative method. The l_1 Gauss-Seidel increases the density of the smoothed interpolation, requiring aggressive truncation that may degrade convergence rates. Its higher memory demands during setup add further challenges. The latest Hypre (Version 2.31.0) supports the mult-additive variant with l_1-Jacobi smoothing, but l_1-Gauss-Seidel smoothing is not yet implemented.

8 Conclusion

We propose two communication-reducing AMG variants: CR-D and CR-M. The CR-D method combines the interpolation and the second residual computation, reducing the number of neighbor communications per level from 4 to 3. CR-M further merges the first residual vectors computation with restriction operations, and by combining two Halo communications, it reduces the number of neighbor communications per level from 4 to 2. Additionally, we implemented CRAMG based on two smoothing techniques: GS and ILU. We also combined the generation of the modified smoothed interpolation operator with the Galerkin product for coarse grid generation, which improves the efficiency of the AMG Setup.

Experiments on a 128-node supercomputer with 64-core Intel Xeon CPUs, using up to 8,192 MPI processes, show CR-D reduces inter-process data exchange by 41-45%, while CR-M reduces message count by 31-35% compared to multiplicative AMG. CRAMG achieves up to 1.30x speedup over traditional methods and up to 1.85x over Hypre's mult-additive method. Both CRAMG variants demonstrate excellent weak and strong scalability in both solve and setup phases. Furthermore, CRAMG proves effective for asymmetric problems, while Hypre's multi-additive approach performs poorly. Our tests on SuiteSparse matrices further confirm CRAMG's performance advantages on real-world problems, with a maximum speedup of 1.16x observed on a single node. These results comprehensively demonstrate the effectiveness and efficiency of the CRAMG method across various application scenarios.

The focus of this work is on distributed-memory CPU systems. We plan to extend CRAMG to GPU architectures in future work. Notably, CRAMG's design (e.g., message reduction) is orthogonal to GPU-specific optimizations, and it could complement existing GPU-accelerated AMG frameworks.

Acknowledgments

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

References

[1] 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 Trans. Parallel Comput.* 7, 3, Article 19 (June 2020), 37 pages. doi:10.1145/3399732

[2] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. 2011. Multigrid Smoothers for Ultraparallel Computing. *SIAM Journal on Scientific Computing* 33, 5 (2011), 2864–2887. doi:10.1137/100798806

[3] Allison H Baker, Robert D Falgout, Tzanio V Kolev, and Ulrike Meier Yang. 2012. Scaling hypre's multigrid solvers to 100,000 cores. In *High-performance scientific computing: algorithms and applications*. Springer, 261–279.

[4] Allison H Baker, Todd Gamblin, Martin Schulz, and Ulrike Meier Yang. 2011. Challenges of scaling algebraic multigrid across modern multicore architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 275–286.

[5] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William Gropp, et al. 2019. PETSc users manual. (2019).

[6] Deshun Bi, Shengguo Li, Dezun Dong, Peng Zhang, and Jianbin Fang. 2024. Optimizing SpMV on Heterogeneous Multi-Core DSPs through Improved Locality and Vectorization. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 1145–1155. doi:10.1145/3673038.3673061

[7] Amanda Bienz, Robert D Falgout, William Gropp, Luke N Olson, and Jacob B Schroder. 2016. Reducing parallel communication in algebraic multigrid through sparsification. *SIAM Journal on Scientific Computing* 38, 5 (2016), S332–S357.

[8] Amanda Bienz, William D Gropp, and Luke N Olson. 2020. Reducing communication in algebraic multigrid with multi-step node aware communication. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 547–561.

[9] Amanda Bienz and Luke N Olson. 2017. RAPtor: parallel algebraic multigrid v0. 1. *Release 0.1* (2017).

[10] James H Bramble, Joseph E Pasciak, and Jinchao Xu. 1990. Parallel multilevel preconditioners. *Mathematics of computation* 55, 191 (1990), 1–22.

[11] William L. Briggs. 1987. A multigrid tutorial. https://api.semanticscholar.org/CorpusID:33704151

[12] Gerald Collom, Rui Peng Li, and Amanda Bienz. 2023. Optimizing Irregular Communication with Neighborhood Collectives and Locality-Aware Parallelism. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 427–437.

[13] T. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25.

[14] Hans De Sterck, Robert D. Falgout, Joshua W. Nolting, and Ulrike Meier Yang. 2008. Distance-two interpolation for parallel algebraic multigrid. *Numerical Linear Algebra with Applications* 15, 2-3 (2008), 115–139. doi:10.1002/nla.559

[15] Hans De Sterck, Ulrike Meier Yang, and Jeffrey J Heys. 2006. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM J. Matrix Anal. Appl.* 27, 4 (2006), 1019–1039.

[16] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. Alphasparse: Generating high performance spmv codes directly from sparse matrices. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[17] Robert D. Falgout and Jacob B. Schroder. 2014. Non-Galerkin Coarse Grids for Algebraic Multigrid. *SIAM Journal on Scientific Computing* 36, 3 (2014), C309–C334. doi:10.1137/130931539

[18] Robert D Falgout and Ulrike Meier Yang. 2002. hypre: A library of high performance preconditioners. In *International Conference on computational science*. Springer, 632–641.

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

[20] L Fournier and S Lanteri. 2001. Multiplicative and additive parallel multigrid algorithms for the acceleration of compressible flow computations on unstructured meshes. *Applied Numerical Mathematics* 36, 4 (2001), 401–426.

[21] Eldad Haber, Matthias Chung, and Felix Herrmann. 2012. An Effective Method for Parameter Estimation with PDE Constraints with Multiple Right-Hand Sides. *SIAM Journal on Optimization* 22, 3 (2012), 739–757. doi:10.1137/11081126X

[22] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. 2005. An overview of the Trilinos project. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 397–423.

[23] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. Parmetis parallel graph partitioning and sparse matrix ordering library. https://api.semanticscholar.org/CorpusID:9818727

[24] Seyed H Mirsadeghi, Jesper Larsson Traff, Pavan Balaji, and Ahmad Afsahi. 2017. Exploiting common neighborhoods to optimize MPI neighborhood collectives. In *2017 IEEE 24th international conference on high performance computing (HiPC)*. IEEE, 348–357.

[25] Jintao Peng, Jianbin Fang, Jie Liu, Min Xie, Yi Dai, Bo Yang, Shengguo Li, and Zheng Wang. 2023. Optimizing MPI Collectives on Shared Memory Multi-Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[26] Haozhong Qiu, Chuanfu Xu, Jianbin Fang, Jian Zhang, Liang Deng, Yue Ding, Qingsong Wang, Shizhao Chen, Yonggang Che, and Jie Liu. 2024. A Conflict-aware Divide-and-Conquer Algorithm for Symmetric Sparse Matrix-Vector Multiplication. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41406.2024.00054

[27] J. W. Ruge and K. Stüben. 1987. Algebraic multigrid. In *Multigrid methods*. Frontiers Appl. Math., Vol. 3. SIAM, Philadelphia, PA, 73–130.

[28] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[29] Edgar Solomonik, Abhinav Bhatele, and James Demmel. 2011. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[30] Klaus Stüben. 2000. Algebraic Multigrid (AMG) : An Introduction With Applications.

[31] Ray S Tuminaro and Charles Tong. 2000. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 5–5.

[32] Petr Vanvek, Marian Brezina, and Jan Mandel. 2001. Convergence of algebraic multigrid based on smoothed aggregation. *Numer. Math.* 88 (2001), 559–579.

[33] Panayot Vassilevski and Ulrike Yang. 2014. Reducing communication in algebraic multigrid using additive variants. *Numerical Linear Algebra with Applications* 21 (03 2014). doi:10.1002/nla.1928

[34] Panayot S Vassilevski. 2008. *Multilevel block factorization preconditioners: Matrix-based analysis and algorithms for solving finite element equations*. Springer Science & Business Media.

[35] Samuel Williams, Mike Lijewski, Ann Almgren, Brian Van Straalen, Erin Carson, Nicholas Knight, and James Demmel. 2014. s-step Krylov subspace methods as bottom solvers for geometric multigrid. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1149–1158.

[36] Jordi Wolfson-Pou and Edmond Chow. 2019. Asynchronous multigrid methods. In *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 101–110.

[37] Ulrike Meier Yang. 2004. On the use of relaxation parameters in hybrid smoothers. *Numerical Linear Algebra with Applications* 11 (2004). doi:10.1002/nla.375

[38] Ulrike Meier Yang. 2010. On long range interpolation operators for aggressive coarsening. *Numerical Linear Algebra with Applications* 17, 2-3 (2010), 453–472. doi:10.1002/nla.689

[39] 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 International Conference on Supercomputing*. ACM, New York, NY, USA, 227–239. doi:10.1145/3577193.3593726

[40] 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* 35, 5 (2024), 768–779. doi:10.1109/TPDS.2024.3372473

[41] L.P. Zhang and Z.J. Wang. 2004. A block LU-SGS implicit dual time-stepping algorithm for hybrid dynamic meshes. *Computers & Fluids* 33, 7 (2004), 891–916. doi:10.1016/j.compfluid.2003.10.004

[42] Qianchao Zhu, Hao Luo, Chao Yang, Mingshuo Ding, Wanwang Yin, and Xinhui Yuan. 2021. Enabling and Scaling the HPCG Benchmark on the Newest Generation Sunway Supercomputer with 42 Million Heterogeneous Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, Article 57, 13 pages. doi:10.1145/3458817.3476158