REDUCING COMMUNICATION IN ALGEBRAIC MULTIGRID USING ADDITIVE VARIANTS

PANAYOT S. VASSILEVSKI AND ULRIKE MEIER YANG

Abstract. Algebraic multigrid (AMG) has proven to be an effective scalable solver on many high performance computers, however its increasing communication complexity on coarser levels has shown to seriously impact its performance on computers with high communication cost. Additive AMG variants provide increased parallelism as well as decreased numbers of messages per cycle, but can also lead to decreased convergence. We present various new additive variants with convergence rates that are signifcantly improved compared to the classical additive algebraic multigrid and investigate their potential for decreased communication, and improved communication/computation overlap, features that are essential for good performance on exascale architectures.

1. Introduction

Algebraic multigrid (AMG) is a popular solver for large-scale scientific computing and an essential component of many simulation codes. AMG has shown to be extremely efficient on distributed memory architectures [3, 4]. However, with single-core speeds plateauing, future increases in computing performance have to rely on increased concurrency provided by the architecture, leading to potentially billions of cores or threads. Applications have to match this increased level of concurrency to exploit the performance potential and hence face additional communication requirements. Future systems will also be subject to strict power limitations for overall system power, and data movement, which includes communication, is responsible for a majority of the power consumed in a system. Therefore to address these challenges, increased parallelism and reduced availibility of power, and to successfully exploit future architectures, it is crucial to develop algorithms with reduced communication. Such an algorithm will consume less power, and its performance will be less affected by a reduction in power.

AMG obtains its optimal computation complexity by using smaller coarse grid problems to approximate the solution of the original fine grid problem. For traditional, matrix-based AMG with Galerkin or variational coarsening, which we consider here, the number of nonzeros per row for the coarse grid operators grows, and with it the number of neighbor processes. The communication complexity increases significantly, leading to a large number of messages. Contention at these levels can lead to a significant decrease in performance and scalability on current architectures with slower networks [5, 12] and is expected to be a bottleneck for future exascale machines. To counter the high communication complexities at the coarse levels, new variants with reduced communication complexity and improved communication-computation

overlap are needed. It is in this context that we revisit additive versions of multigrid methods.

Additive multigrid methods were originally developed in the eighties [14, 6] to increase parallelism within multigrid. Theoretical and practical investigations showed however that additive multigrid converges signicifantly slower than multiplicative multigrid and so erased any gains in time improvements [1]. Chan and Tuminaro [7, 8, 21] improved convergence of the additive formulation through the introduction of filtering operators. However the additional cost of the filtering operators eliminated the gains in improved convergence [18]. Fournier and Lanteri [13] were able to achieve modest performance improvements using a modified version of filtering multigrid on the two coarsest levels of the grid hierarchy for compressible flow computations.

We investigate here an additive AMG variant, which exhibits identical convergence behavior to multiplicative AMG (see Section 5.7.2 in [23]), and several new variants of this method, that are less expensive, but still exhibit very good convergence properties. We analyze their computational and communication cost as well as memory requirements. We demonstrate improved solve times for the new variants over multiplicative AMG on a parallel Linux cluster at Lawrence Livermore National Laboratory. The results show that several of the new AMG variants are effective at aleviating the difficulties caused by the increased size of coarse-grid stencils typical for variational or Galerkin coarsening in traditional (i.e., matrix-based only) AMG.

2. The V-cycle revisited: multiplicative and additive versions

Let A be a given sparse $n \times n$ symmetric positive definite (s.p.d.) matrix, and M a given A–convergent smoother, i.e., $\|I - M^{-1}A\|_A = \|I - A^{\frac{1}{2}}M^{-1}A^{\frac{1}{2}}\| < 1$, or equivalently, $M + M^T - A$ to be s.p.d.

We assume that M^{-1} is a sparse matrix, for example a diagonal matrix, or a polynomial in terms of a diagonal matrix times A.

We generate a hierarchy of coarse level matrices A_k based on a sequence of interpolation matrices P_{k+1}^k of size $n_k \times n_{k+1}$, $k = 0, \ldots, \ell$. Here, $n_0 = n$ and $n_{k+1} < n_k$. Also, with $A_0 = A$, we have

$$A_{k+1} = (P_{k+1}^k)^T A_k P_{k+1}^k \text{ for } k = 0, \ldots, \ell - 1.$$

Finally, let M_k be an A_k–convergent smoother for A_k. We assume for a later purpose that M_k^{-1} is sparse, in the simplest case a diagonal matrix. We also need the symmetrized smoothers

$$\overline{M}_k = M_k \left(M_k + M_k^T - A_k\right)^{-1} M_k^T. \tag{2.1}$$

A multiplicative, symmetric $V(1,1)$–cycle operator $B = B_0$ is defined by recursion as follows:

- $B_\ell = A_\ell^{-1}$.
- For $k < \ell$, assuming that B_{k+1} has been defined, we define

$$B_k = \overline{M}_k^{-1} + (I - M_k^{-T}A_k)P_{k+1}^k B_{k+1}(P_{k+1}^k)^T(I - A_k M_k^{-1}). \tag{2.2}$$

Formula (2.2) follows from the more familiar product iteration formula

$$I - B_k A_k = (I - M_k^{-T} A_k) \left(I - P_{k+1}^k B_{k+1} \left(P_{k+1}^k \right)^T A_k \right) (I - M_k^{-1} A_k), \tag{2.3}$$

which represents the fact that the error propagation matrix at level k is a product of the pre-smoothing iteration with M_k represented by the term $I - M_k^{-1} A_k$, coarse-grid correction corresponding to $I - P_{k+1}^k B_{k+1} \left(P_{k+1}^k \right)^T A_k$ which involves restriction with $\left(P_{k+1}^k \right)^T$, application of B_{k+1} (defined recursively), and interpolation with P_{k+1}^k, and finally post-smoothing iteration with M_k^T giving rise to the last term $I - M_k^{-T} A_k$. To prove (2.2) from (2.3), we use (2.1), or more specifically the identity

$$I - \overline{M}_k^{-1} A_k = (I - M_k^{-T} A_k)(I - M_k^{-1} A_k).$$

For the actual details, we refer to [23], Section 5.1.

We also note that we use both M_k and M_k^T in the definition of the V-cycle operator which makes it symmetric (and positive definite if M_k is A_k-convergent), hence suitable for preconditioning in the conjugate gradient (or CG) method. For example, if M_k corresponds to forward Gauss-Seidel, i.e., $M_k = D_k + L_k$, where D_k is the diagonal of A_k and L_k the strictly lower-triangular part of A_k, then $M_k^T = D_k + L_k^T$ corresponds to the backward Gauss–Seidel smoothing process.

Modified interpolation matrices and smoothers. Having constructed or available the operators:

- hierarchy of matrices $\{A_k\}_{k=0}^{\ell}$,
- smoothers $\{M_k\}_{k=0}^{\ell-1}$, and
- two-level interpolation matrices $\{P_{k+1}^k\}_{k=0}^{\ell-1}$,

that define a (symmetric) $V(1,1)$-cycle, we can construct the following modified operators:

- For $k = 0, \ldots, \ell - 1$ form the "smoothed" two-level interpolants:

  $$\overline{P}_{k+1}^k = (I - M_k^{-T} A_k) P_{k+1}^k.$$

  Under the assumption that M_k^{-1} is sparse, we can expect that $\overline{P}_{k+1}^k$ is also sparse and hence can be explicitly formed and stored.
- Form the inverses of the symmetrized smoothers, i.e.,

  $$\begin{aligned} \Lambda_k &= \left(\overline{M}_k \right)^{-1} \\ &= M_k^{-T} \left(M_k + M_k^T - A_k \right) M_k^{-1} \\ &= M_k^{-1} + M_k^{-T} - M_k^{-T} A_k M_k^{-1}. \end{aligned}$$

  Again, under the assumption that M_k^{-1} is sparse, we can expect that Λ_k is also sparse and hence can be explicitly formed and stored.
- Define $\Lambda_\ell = A_\ell^{-1}$.

Additive representation of symmetric $V(1,1)$–cycle. Based on the recursive definition (2.2), using the newly introduced parameters, we have

$$\begin{aligned} B_k &= \overline{M}_k^{-1} + (I - M_k^{-T}A_k)P_{k+1}^k B_{k+1}(P_{k+1}^k)^T(I - A_k M_k^{-1}) \\ &= \Lambda_k + \overline{P}_{k+1}^k B_{k+1}\left(\overline{P}_{k+1}^k\right)^T \\ &= \Lambda_k + \sum_{j=k+1,\,...,\,\ell} \overline{P}_j^k \Lambda_j \left(\overline{P}_j^k\right)^T. \end{aligned}$$

Above, we have used the composite interpolants for any $j > k$,

$$\overline{P}_j^k = \overline{P}_{k+1}^k \overline{P}_{k+2}^{k+1} \ \dots \ \overline{P}_j^{j-1}. \tag{2.4}$$

We let $\overline{P}_k^k = I$.

Then, for $k = 0$, we have (with $\overline{P}_j = \overline{P}_j^0$)

$$B = \sum_{j=\ell,\,\ell-1,\,...,\,0} \overline{P}_j \Lambda_j \left(\overline{P}_j\right)^T. \tag{2.5}$$

The last identity represents the additive representation of the multiplicative V-cycle that is found in [23], Section 5.7.2. It is mathematically equivalent to the original multiplicative V-cycle.

We now recall the definition of the classical additive MG method, also referred to as BPX [6]. Introduce the composite interpolants from coarse level j all the way up to the finest level,

$$P_j = P_1^0 P_2^1 \dots P_j^{j-1}, \quad 1 \le j \le \ell, \ (P_0 = I).$$

For any given s.p.d. smoothers Λ_j, for example $\Lambda_j^{-1} = \overline{M}_j$, we form the additive MG (or BPX) operator

$$B_{add} = \sum_{j=0,\,...,\,\ell} P_j \Lambda_j P_j^T, \tag{2.6}$$

where typically $\Lambda_\ell = A_\ell^{-1}$. We notice the similarity with formula (2.5); the difference is in the interpolation operators used; the BPX method uses the standard interpolation operators P_j, whereas in (2.5), the product of the smoothed out interpolation matrices are employed. Also, B_{add}, unless properly scaled, is generally not a convergent method for solving $A\mathbf{x} = \mathbf{b}$, hence it is typically used as a preconditioner in the CG method. While the BPX method provides a spectrally equivalent preconditioner for matrices A that have been constructed by finite elements and successive geometric refinement applied to 2nd order elliptic equations, it converges significantly slower than the corresponding multiplicative method with the same smoothers and interpolation matrices.

In the s.p.d. case, assuming A-convergent smoothers, as is well-known, the V-cycle operator B provides a convergent iteration, i.e., we have the inequalities

$$0 < \mathbf{v}^T B \mathbf{v} \le \mathbf{v}^T A^{-1} \mathbf{v}, \ \text{for all } \mathbf{v}.$$

Based on (2.5), the following result is readily seen.

Proposition 2.1. *Assume that the smoothing operators Λ_j are replaced by s.p.d. approximations $\overline{\Lambda}_j$ such that*

$$\mathbf{v}_j^T \Lambda_j \mathbf{v}_j \geq \mathbf{v}_j^T \overline{\Lambda}_j \mathbf{v}_j \text{ for all } \mathbf{v}_j.$$

Then, the modified additive operator

$$\overline{B} = \sum_{j=\ell,\ \ell-1,\ \ldots,\ 0} \overline{P}_j \overline{\Lambda}_j \left(\overline{P}_j\right)^T. \tag{2.7}$$

also provides a convergent method for A.

Proof. The proof is readily seen since by construction $\overline{B}$ is s.p.d., and also by the properties of $\overline{\Lambda}_j$, we have

$$0 \leq \mathbf{v}^T \overline{B} \mathbf{v} \leq \mathbf{v}^T B \mathbf{v} \leq \mathbf{v}^T A^{-1} \mathbf{v}, \text{ for all } \mathbf{v}.$$

The latter inequalities imply

$$0 \leq \mathbf{v}^T A(I - BA)\mathbf{v} \leq \mathbf{v}^T A(I - \overline{B}A)\mathbf{v} \leq \left(1 - \lambda_{\min}(A\overline{B})\right) \mathbf{v}^T A \mathbf{v},$$

which shows the desired convergence property of $\overline{B}$. □

As a corollary, in the case of s.p.d. smoothers, where $M_j = M_j^T$, we get

$$\Lambda_j = 2M_j^{-1} - M_j^{-1} A_j M_j^{-1}.$$

Assuming that

$$\mathbf{v}_j^T M_j \mathbf{v}_j \geq \mathbf{v}_j^T A_j \mathbf{v}_j \text{ for all } \mathbf{v}_j, \tag{2.8}$$

it is easily seen that we have

$$\mathbf{v}_j^T \Lambda_j \mathbf{v}_j = \mathbf{v}_j^T (2M_j^{-1} - M_j^{-1} A_j M_j^{-1}) \mathbf{v}_j \geq \mathbf{v}_j^T M_j^{-1} \mathbf{v}_j, \text{ for all } \mathbf{v}_j.$$

Therefore, we can let

$$\overline{\Lambda}_j = M_j^{-1},$$

and then based on formula (2.7), it follows that the modified additive version of the V-cycle

$$\overline{B} = \sum_{j=\ell,\ \ell-1,\ \ldots,\ 0} \overline{P}_j \overline{\Lambda}_j \left(\overline{P}_j\right)^T = \sum_{j=\ell,\ \ell-1,\ \ldots,\ 0} \overline{P}_j M_j^{-1} \left(\overline{P}_j\right)^T,$$

provides a convergent method. The latter fact is particularly attractive in the case when M_j is a diagonal matrix, as is the case of the ℓ_1 Jacobi smoother that we investigate in the following sections. Note that at fine levels we may keep $\overline{\Lambda}_j = \Lambda_j$ and use approximation only at coarse levels.

Other ways to define sparser approximations to B, that are exploited in this paper are to utilize truncation of the smoothed interpolation operators $\overline{P}_{k+1}^k$. Our tests confirm that the resulting approximation to B is convergent for moderate truncation.

Multiplicative Version	Additive Version
$r_0 = b$	$r_0 = b$
For $k = 0, ..., \ell - 1$ (sequential) $x_k = M_k^{-1} r_k$ $r_{k+1} = (P_{k+1}^k)^T (r_k - A_k x_k)$	For $k = 0, ..., \ell - 1$ (sequential) $r_{k+1} = (\overline{P}_{k+1}^k)^T r_k$
	For $k = 0, ..., \ell - 1$ (parallel) $x_k = M_k^{-1} r_k$ $x_k := x_k + M_k^{-T}(r_k - A_k x_k)$
Solve $A_\ell x_\ell = r_\ell$	Solve $A_\ell x_\ell = r_\ell$
For $k = \ell - 1, ..., 0$ (sequential) $x_k := x_k + P_{k+1}^k x_{k+1}$ $x_k := x_k + M_k^{-T}(r_k - A_k x_k)$	For $k = \ell - 1, ..., 0$ (sequential) $x_k := x_k + \overline{P}_{k+1}^k x_{k+1}$

Table 1: Multiplicative and additive formulation of an AMG V-cycle, when used as a preconditioner, i.e. $x_0 = 0$. We also take advantage of the fact that $x_k = 0, k = 1, ..., \ell - 1$ at the beginning of each level.

3. Implementation Details and Cost Analysis

We will now consider the implementation cost of both the multiplicative and additive implementation of the AMG V-cycle. Since we generally use AMG as a preconditioner, i.e. we perform one cycle of AMG with $x_0 = 0$ in each iteration of an iterative solver such as conjugate gradient or GMRES, we list both versions when used as a preconditioner in Table 1.

Note that in our implementation we take advantage of the fact that at the beginning of each level $x_k = 0, k = 0, ..., \ell - 1$. Therefore

$$x_k := x_k + M_k^{-1}(r_k - Ax_k)$$

can be replaced with

$$x_k = M_k^{-1} r_k.$$

The smoothing step in the additive V-cycle

$$x_k = M_k^{-1} r_k,\ x_k := x_k + M_k^{-T}(r_k - A_k x_k) \tag{3.1}$$

is equivalent to

$$x_k = \Lambda_k r_k, \tag{3.2}$$

where

$$\Lambda_k = M_k^{-T}(M_k + M_k^T - A_k) M_k^{-1}. \tag{3.3}$$

Both formulations will be of interest when used with different smoothers. We will also investigate the variant, in which Λ_k is approximated with

$$\overline{\Lambda}_k = M_k^{-1}, \tag{3.4}$$

for smoothers M_k that are s.p.d. and satisfy $\mathbf{v}_k^T M_k \mathbf{v}_k \geq \mathbf{v}^T A_k \mathbf{v}_k$, i.e., estimate (2.8), which is required in the corollary after Proposition 2.1. That is, steps (3.1), as combined in (3.2), are replaced with

$$x_k = \overline{\Lambda}_k r_k = M_k^{-1} r_k. \tag{3.5}$$

From here on, we will refer to the classical additive algorithm (corresponding to (2.6)), which uses the unsmoothed interpolation, $\overline{P}_{k+1}^k = P_{k+1}^k$, in Table 1 as *additive AMG*, the additive formulation with smoothed interpolation (corresponding to (2.5)) as *mult-additive AMG*, and the algorithm with (3.4) (corresponding to (2.7)) as *simplified mult-additive AMG*. Note that classical additive AMG is not equivalent to multiplicative AMG.

Communication cost analysis. We assume that the application of M_k^{-1} and M_k^{-T} requires no communication. This is a reasonable assumption since it holds for various smoothers that are often used in practice, such as weighted Jacobi or any other diagonal smoother as well as for the hybrid Gauss-Seidel (for the latter smoother see, e.g., [2]).

Then for the multiplicative cycle, communication is required for two matrix-vector multiplications for A_k, one for P_{k+1}^k and one for its transpose on each level k. For the additive and mult-additive approach, only one multiplication with A_k as well as multiplications with $\overline{P}_{k+1}^k$ and its transpose are required per level. The simplified mult-additive V-cycle avoids all communications for A_k. Since additive AMG uses the unsmoothed interpolation, one additive AMG cycle requires one less matrix-vector multiplication with A_k than the multiplicative AMG cycle and consequently less communication. For the mult-additive variant and the simplified version, it is more complicated to compare their communication cost to that of multiplicative AG, since it depends on the smoothed interpolation operator $\overline{P}_{k+1}^k$, which has a larger stencil and requires more communication than the original interpolation. We will further investigate this in Section 4. Additionally due to the fact that smoothing on all levels can now be performed in parallel, it is possible to combine the communication of all levels leading to a reduction of the number of messages for all three approaches.

Memory and number of operations analysis. Besides communication, it is also of interest to see the impact of the changes on memory usage and number of operations. In order to get some actual estimates it is necessary to specify the smoothers.

We will consider here ℓ_1 smoothers (introduced in [17] and available in *hypre*), since AMG with ℓ_1 smoothing always converges, if the original matrix A is s.p.d. (cf. e.g., [2]).

The simplest ℓ_1 smoother is ℓ_1 Jacobi, where M is a diagonal matrix $\tilde{D}$ with diagonal elements

$$\tilde{d}_{ii} = \sum_{j=1}^{n} |a_{ij}|, \tag{3.6}$$

for an $n \times n$ matrix A. Note that the ℓ_1 Jacobi smoother is s.p.d. and fulfills $v^T \tilde{D} v \geq v^T A v$ and is therefore a good choice for the simplified mult-additive approach.

We will also consider ℓ_1 Gauss-Seidel, which generally leads to better convergence than ℓ_1 Jacobi. To define ℓ_1 Gauss-Seidel, we partition A into $k \times k$ blocks, which do not have to be of the same size. Let A_{mm} be the mth diagonal block of A, which is decomposed into $A_{mm} = L_{mm} + D_{mm} + U_{mm}$, where L_{mm} is the lower triangular and U_{mm} the upper triangular portion of A_{mm}. We define M_H as the block diagonal matrix with $(M_H)_{mm} = L_{mm} + D_{mm}$. The ℓ_1 Gauss-Seidel smoother is defined as

$$M_{\ell_1} = M_H + D_{\ell_1}, \tag{3.7}$$

where D_{ℓ_1} is a diagonal matrix with entries

$$(d_{\ell_1})_{ii}^m = \sum_{j \in \Omega_o^m} |a_{ij}|, \tag{3.8}$$

and Ω_o^m is the column index set outside of the block A_{mm}. Generally the blocks are chosen according to the partitioning across cores, however we will also consider smaller block sizes.

We first investigate the mult-additive approach with ℓ_1 Jacobi smoothing, which is of interest due to its simplicity and its large degree of parallelism. Many of our conclusions, although not all of them, will also carry over to ℓ_1 Gauss-Seidel.

Use of ℓ_1 Jacobi smoothing. The additive V-cycle with Jacobi smoothing can be implemented very efficiently by taking advantage of (3.2). Since M_k is a diagonal matrix $\tilde{D}_k$, it follows that

$$\Lambda_k = 2\tilde{D}_k^{-1} - \tilde{D}_k^{-1} A_k \tilde{D}_k^{-1}. \tag{3.9}$$

The explicit computation of Λ_k is fairly cheap, and Λ_k has the same data structure as A_k. Additionally the matrix vector multiplications (3.2) on all levels can be combined into one large matrix vector multiplication

$$\tilde{x} = \Lambda \tilde{r}, \tag{3.10}$$

leading to a reduced number of messages.

Table 2 shows the memory usage and matrix-vector multiplications required for V-cycles of various variants when using Jacobi-smoothing. For the additive and mult-additive approach, $A_k, k = 1, ..., \ell$ and $P_{k+1}^k, k = 0, ..., \ell - 1$ can be discarded immediately after computing Λ_k and $\overline{P}_{k+1}^k$, since they are no longer needed. (We list Λ_k as A_k, since they have the same data structure.) We however need to keep the original matrix A_0, which is needed in the solver if we use AMG as a preconditioner. Note that the computation of Λ_k and the extra storage of Λ_0 can be avoided if one implements (3.1) instead of (3.2) at the cost of extra flops. Finally, in the simplified

	multiplicative	additive	mult-additive	simplified
memory	$\sum_{k=0}^{\ell-1}[nnz(A_k)$ $+nnz(P_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[nnz(A_k)$ $+nnz(P_{k+1}^k)]$ $+nnz(A_0)$	$\sum_{k=0}^{\ell-1}[nnz(A_k)$ $+nnz(\overline{P}_{k+1}^k)]$ $+nnz(A_0)$	$\sum_{k=0}^{\ell-1}[nnz(\overline{P}_{k+1}^k)$ $\{+nnz(A_{k+1})\}]$ $+nnz(A_0)$
MatVecs	$\sum_{k=0}^{\ell-1}[2MV(A_k)$ $+2MV(P_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[MV(A_k)$ $+2MV(P_{k+1}^k)]$	$\sum_{k=0}^{\ell-1}[MV(A_k)$ $+2MV(\overline{P}_{k+1}^k)]$	$\sum_{k=0}^{\ell-1} 2MV(\overline{P}_{k+1}^k)$

Table 2: Memory usage and matrix-vector multiplications in a V-cycle when using Jacobi smoothing. The terms in curly brackets can be discarded in the setup phase and are not needed during the solve phase.

version, it is possible to discard all $A_k, k = 1, ...\ell$, since they are no longer needed after smoothing the interpolation.

4. Comparison of variants relative to multiplicative V-cycle

In order to compare the V-cycles of the variants to that of the multiplicative method, we compute *change factors* for memory and flops by dividing the values for each of the additive variants by those of the multiplicative variant. Change factors for memory and flops are evaluated based on the numbers given in Table 2, when using Jacobi smoothing. The terms in curly brackets refer to matrices that can be discarded after the setup and are no longer needed in the solve phase. We include these terms however in the memory change factors that we present. For the number of messages and the amount of data sent, we used the average number of messages and amount of data sent per core obtained in the test runs. We then divided the values for each of the additive variants by those of the multiplicative variant. Therefore, quantities with a change factor smaller than 1 indicate an improvement of the V-cycle of the new method over the multiplicative V-cycle. Since we generally use AMG as a preconditioner, we assumed that $x_0 = 0$, and therefore did not include the computation of Ax_0 in the computation of the change factors. Note that generating Λ_k and the smoothed interpolation is not considered here, since it occurs during the setup phase ileading to an increase in setup times. We will comment on this in Section 5.

We implemented the mult-additive AMG variant with ℓ_1 Jacobi smoothing in the linear solvers library *hypre* [15] and applied them to a 3-dimensional (3D) 7-point Laplace problem on a cube with $50 \times 50 \times 50$ variables per core using HMIS coarsening [9], ext+i interpolation [10] truncated to at most 4 elements per row and one level of aggressive coarsening [20, 24]. This combination of coarsening and interpolation often leads to the fastest solution when using *hypre*'s AMG solver BoomerAMG.

In Table 3, we have listed the change factors for the mult-additive over the multiplicative approach, varying the number of cores. We see a significant increase in memory, mainly caused by the fact that we keep the original matrix as well as Λ_0.

no of cores	Memory	flops	no messages	data sent
64	2.203	1.013	0.712	0.670
512	2.206	1.016	1.169	0.688
4096	2.207	1.012	1.655	0.727

Table 3: Change factors for various quantities for mult-additive / multiplicative approach. Factors < 1 are good.

There is an additional, but less significant increase caused by the smoothed interpolant. We see a small increase in flops, which is of no concern, since we expect flops to be cheap on future exascale computers. For this method, there is no need to consider convergence, since it uses exactly the same number of iterations as multiplicative AMG. The amount of data sent across cores decreases, which is positive, since data movement is expensive. However, it appears that there is a tendency of slight increase when increasing the number of cores. More importantly, there is a significant increase in the number of messages sent, caused by the increased stencil size of the smoothed interpolation operators. We can reduce this effect by truncating interpolation which, however, we expect to affect convergence. We investigate this in the following section.

Truncating the smoothed interpolation matrix. In order to reduce the increase in number of nonzeros in $\overline{P}^k_{k+1}$, we consider truncating this matrix. There are essentially two ways we can truncate interpolation operators: we can choose a truncation factor θ and eliminate every weight whose absolute value is smaller than this factor, i.e. for which $|(\overline{P}^k_{k+1})_{ij}| < \theta$ [20], or we can limit the number of coefficients per row, i.e. choose only the k_{max} largest weights in absolute value. In both cases the new weights need to be rescaled so that the row sums remain unchanged.

We vary the numbers for both truncation strategies for the 7-point 3D Laplace test problems using 4096 processes. We also investigate a 3D diffusion problem with a 27-point stencil to see the effects on a larger stencil. The results are illustrated in Figure 1, where 'msg' denotes the change factor for the number of messages, 'data' for the amount of data sent and 'flops' the change in the amount of operations. Solid lines illustrate results for the 7-point stencil, and dashed lines those for the 27-point stencil. Results per cycle, as well as change factors adjusted for the increase in iterations are presented. For the adjustment, we multiply the change factors with the number of iterations of the new method divided by the number of iterations for the multiplicative cycle to adjust for the increase in iterations.

The change factors for a cycle show good improvement when increasing the truncation factor or decreasing the maximal number of elements per row. However, the picture changes when one adjusts for the increase in iterations. As expected, if one truncates too much, the increasing number of iterations quickly erases any potential gains and leads to on overall worse method. For the considered problems, a 7-point and a 27-point stencil with $50 \times 50 \times 50$ grid points per core, a good truncation factor is 0.025, or restricting the number of nonzeros per row to at about 8. We see that

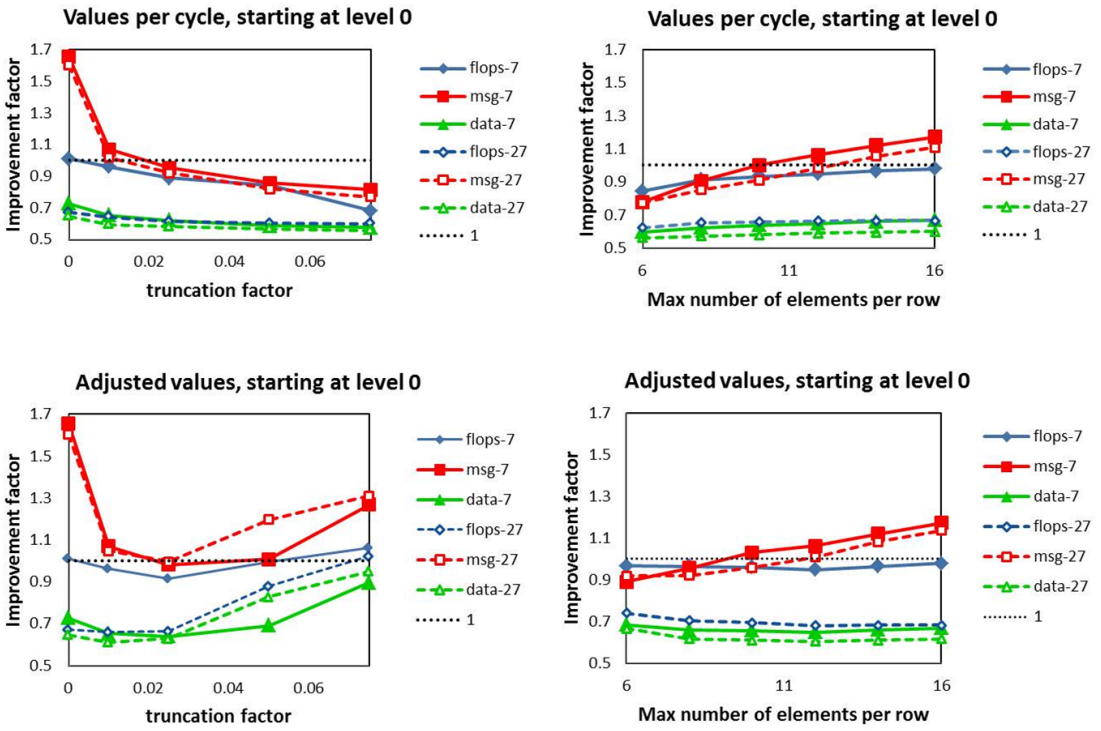


Figure 1: Effect of truncation of smoothed interpolation on communication and number of flops for a system with a 7-point and a 27-point stencil, using 4096 cores.

the change factor for the number of messages now drops slightly below 1, the other change factors also improve somewhat. Note also that for the 27-point stencil there is a significant decrease in flops.

Note that we found $k_{max} = 8$ and $\theta = 0.025$ also to be the values that generally led to the best solve times for the test problems considered in Section 5. The development of a procedure to automatically determine the optimal values is a topic of future research.

Performance profiling of V-cycles. It is of interest to look at performance profiles of the V-cycles to see how much time is spent in computation and communication as well as whether there is any overlap. Note that matrix-vector multiplications have been implemented in such a way that communication to send and receive off processor data is started first using non-blocking sends and receives, so that the local portion of the matrix vector multiplication can be computed at the same time while data is being sent. The off processor portion is computed as soon as the needed data has arrived. We used Vampir [22] to generate performance profiles of the multiplicative, the additive and the mult-additive V-cycle using smoothed interpolation truncated to at most 8 elements per row on 64 cores of the Linux cluster Hera at LLNL. The timeline for each individual process is colored green to mark time spent for performing

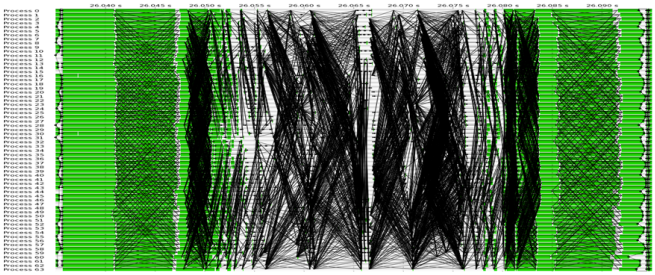
computation and white for idle time. Communication is marked by black lines leading from one process to the next.

Results for a 3D 7-point Laplace problem with $40 \times 40 \times 40$ grid points per core are illustrated in Figure 2. The widths of the profiles were adjusted for the actual times spent, indicating that the additive cycle is the fastest, and the multiplicative cycle the slowest. For the multiplicative approach there is initial good computation-communication overlap on the finest level as can be seen on the left and the right side of the top profile. There is however significant contention at the coarser levels where computation time is very small in comparison to communication time. For the additive and the mult-additive method, where smoothing is replaced by one large matrix-vector multiplication, one can see good overlap of computation and communication for the smoothing in the center of the profiles. As expected the mult-additive cycle with truncated interpolation is more expensive and requires more communication than the additive cycle, but it also shows good overlap, and its significantly faster convergence leads to overall better solve times. We do not show a profile of the simplified version, however for its truncated version the center profile in the bottom performance profile would just be replaced by a very small green band for the multiplication with the inverse of the diagonal matrix, which requires no communication.

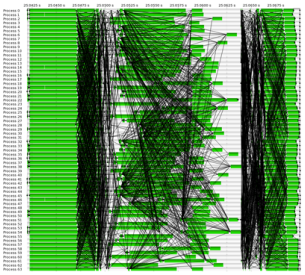
Switching to additive variants at coarser levels. Since the contention we observed is mainly at the coarser levels, it is of interest to investigate whether it is beneficial to start the new approaches at later levels. Based on the results in the previous section, we include two mult-additive variants with truncation, one using a truncation factor of 0.025, denoted 'ma tr 0.025', and a second one in which we truncate the smoothed interpolation matrix to at most 8 coefficients per row, denoted 'ma Pmx 8'. We also include a simplified method denoted with 's-m-a' and its truncated version 'sma Pmx 8'. We now investigate the effect of starting the various approaches at later levels of the AMG hierarchy, and using the multiplicative approach at the finer levels above the switch level.

Figure 3 shows the number of iterations as well as the change factors for memory, number of messages, amount of data sent as well as flops, for both a 7-point and a 27-point stencil. We did not include the results for the number of iterations and the amount of data sent for the 27-point stencil, since the results for the 7-point stencil are representative for the 27-point stencil as well. The fine matrix with a 7-point stencil is relatively small compared to problems with larger stencils, and therefore we do not see the large reduction in flops seen for the 27-point stencil at the finest level. The effect is even more pronounced since we use aggressive coarsening on the finest level.

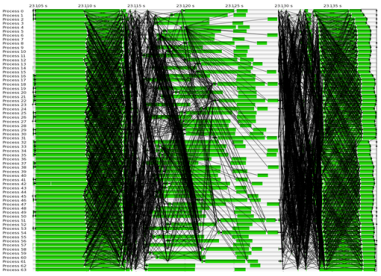
The truncated versions show good convergence, whereas the simplified versions converge slower, but still significantly better than the additive method. The amount of memory needed decreases significantly when starting the mult-additive or additive method at a later level. The simplified versions require much less memory when started at the finest level. Note that if we choose the less efficient implementation and do not evaluate Λ_0, memory usage for the mult-additive methods drops to that



Multiplicative
V-cycle with
L1-Jacobi



Additive
V-cycle with
L1-Jacobi



Mult-additive V-cycle,
smoothed interpolation truncated
to at most 8 elements per row
with L1-Jacobi

Figure 2: Performance profiles of one AMG cycle with Jacobi smoothing using various AMG variants, showing portions of computation, idle time and MPI calls, using 64 cores. Green: computation, white: idle time, black: MPI calls

of the simplified methods, and memory usage for the additive method is equivalent to that of the multiplicative method.

The results show that for all methods except additive AMG, there is a very slight increase in the change factor for the number of messages when increasing the level at which the method is started. For the truncated mult-additive methods it stays below or around 1, and for the truncated simplified method it is clearly below 1 in the case of the 27-point stencil, but around 1 at level 0 and then clearly above 1 in the case of the 7-point stencil. For the additive method it starts to drop below 1 around level 2 and continues decreasing as the number of iterations decreases. For the additive method the amount of data sent as well as the number of operations stay above 1, for most of the other methods they are clearly below 1 at level 0, especially for the 27-point stencil, but increase at later levels. They are significantly reduced for the simplified methods at level 0.

Overall, the truncated simplified method leads to the best communication savings if started at level 0, in spite of the increased number of iterations.

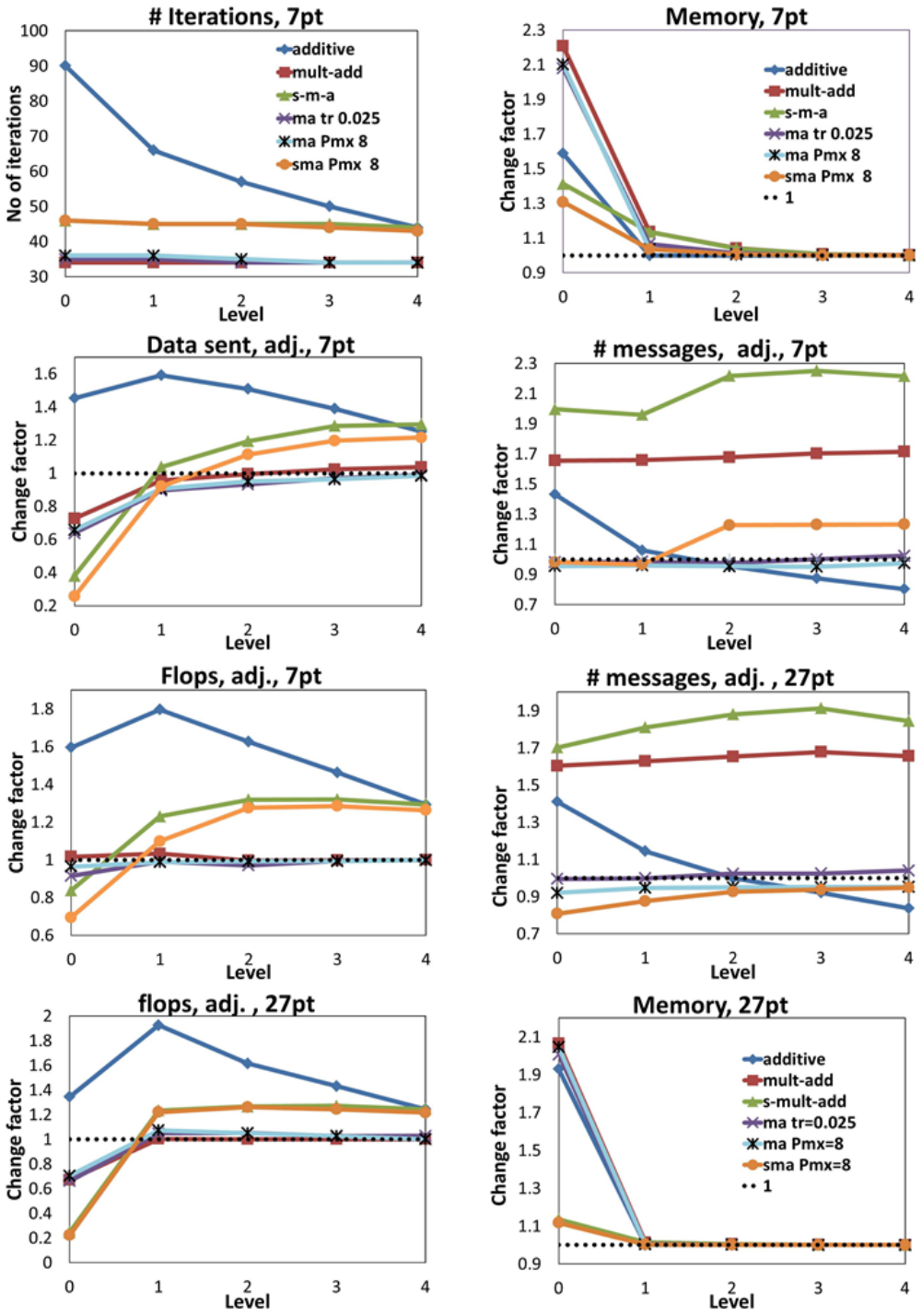


Figure 3: Effect of starting various versions at different levels - using 4096 cores. Change factors for flops, number of messages and amount of data sent were adjusted to reflect the change in number of iterations.

	7-point stencil			27-point stencil		
n	mult-additive	Pmx 8	tr 0.025	mult-additive	Pmx 8	tr 0.025
20	21.79	1.54 (20.73)	1.63	5.40	1.22 (5.42)	1.16
30	53.46	1.54 (50.67)	1.66	11.77	1.21 (11.77)	1.16
40	107.39	1.54 (101.69)	1.68	22.99	1.21 (22.90)	1.17

Table 4: Increase in memory uisng mult-additive approach over multiplicative approach with ℓ_1 Gauss-Seidel smoothing on 64 cores using $n \times n \times n$ grid points per core, (numbers in parentheses is temporary memory increase during setup)

	7-point stencil			27-point stencil		
ℓ_{GS}	mult-additive	Pmx 8	tr 0.025	mult-additive	Pmx 8	tr 0.025
0	53.46	1.54 (50.67)	1.66	11.77	1.21 (11.77)	1.16
1	3.97	1.04 (3.80)	1.12	1.07	1.00 (1.06)	1.01
2	1.19	1.01 (1.19)	1.02	1.01	1.00 (1.01)	1.00

Table 5: Increase in memory using mult-additive approach over multiplicative approach with ℓ_1 Gauss-Seidel smoothing on 64 cores using $30 \times 30 \times 30$ grid points per core, starting at different levels ℓ_{GS} (numbers in parentheses is temporary memory increase during setup)

Using ℓ_1 Gauss-Seidel as a smoother. Since Gauss-Seidel smoothing generally converges faster than Jacobi and is often used in practice, we also want to investigate it in the present context. At first view, it does not appear to be a suitable smoother here, since using it to smooth the interpolation will significantly increase the density of the smoothed interpolation, and require much more memory. While this can be mitigated by applying truncation again, one needs to truncate a lot more, which could lead to much worse convergence. Also, we need to first generate the smoothed interpolation, before we can truncate it. This requires temporary storage during the setup time, which can be just as expensive. Table 4 gives some data showing the extreme increase in memory required.

These memory requirements can be significantly decreased when starting at a later level. Memory increases for $n = 30$ are given in Table 5.

Another disadvantage of using an ℓ_1 Gauss-Seidel smoother is the fact that the algorithm is highly sequential within a core, thus preventing the overlap of communication and computation that were achieved using Jacobi smoothing. This lack of overlap is demonstrated in the performance profile of the additive V-Cycle using hybrid Gauss-Seidel in the center of Figure 4.

Computation-communication overlap can be achieved with a block ℓ_1 Gauss-Seidel with several smaller blocks per core. Now the portion outside the blocks, but local to the core, can be computed while communicating data, and the remaining local portion as well as the off processor portion of the smoothing step can be finalized once the data

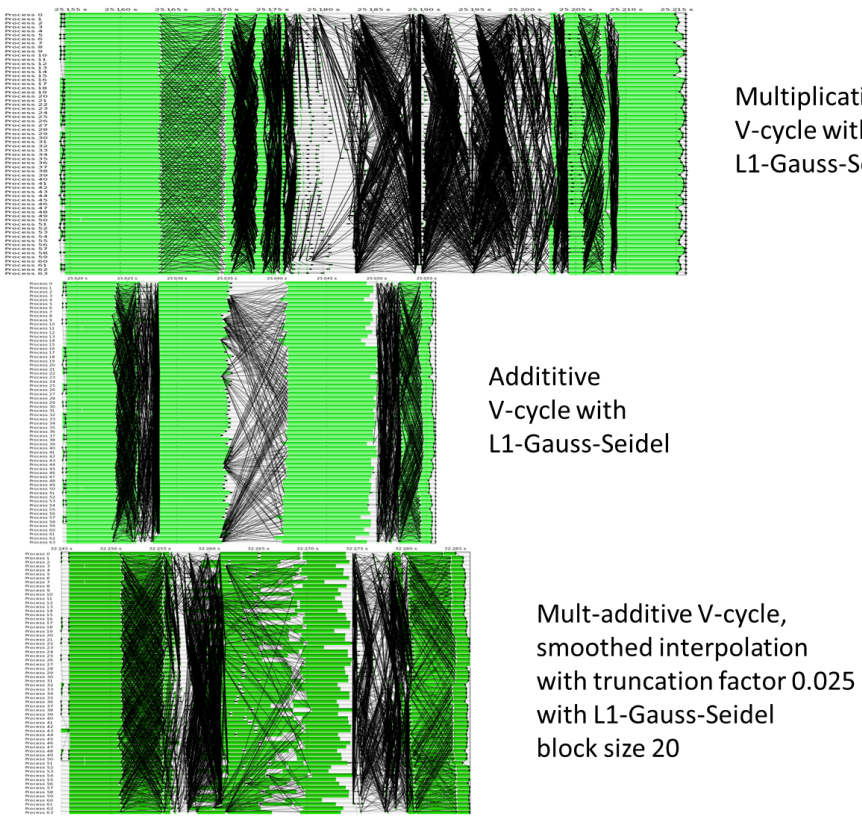


Figure 4: Performance profile of one AMG cycle with ℓ_1 Gauss-Seidel smoothing using various AMG variants, showing portions of computation, idle time and MPI calls, using 64 cores. Green: computation, white: idle time, black: MPI calls

have been received. The improved overlap is demonstrated in the bottom performance profile of Figure 4, for a block size of 20 and a truncated smoothed interpolation.

The block ℓ_1 Gauss-Seidel also can significantly reduce memory requirements, especially when the block sizes are small, since now the nonzero portion of M_k^{-1} is also much smaller. There is also increased parallelism in the setup, which an efficient implementation can take advantage of.

It is also possible to consider a simplified version here, however M_k is not s.p.d., and therefore Proposition 2.1 does not apply. Also, we generally use AMG as a preconditioner for CG, which requires a symmetric smoother. We tried to use M_k^{-1} and wrapped restarted GMRES around it, and obtained fairly good convergence. We also used block symmetric ℓ_1 Gauss-Seidel and wrapped conjugate gradient around it. We got better convergence than with the ℓ_1 Jacobi approach, but the overall times were slower.

5. Numerical Experimentss

We tested the variants descibed in the previous sections on a variety of test problems, which are described below.

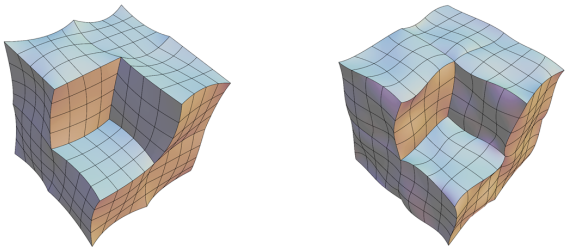


Figure 5: A coarse version of the unstructured mesh on the Fichera domain using quadratic (left) or cubic (right) finite elements.

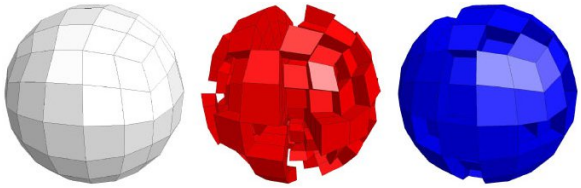


Figure 6: A coarse version of the unstructured mesh used for problem U-Jumps, with the two material subdomains shown on the right.

Test problems. We consider two structured problems and four unstructured problems. All problems with the exception of Problem 2 solve the scalar diffusion problem

$$-\nabla \cdot (a(x,y,z)\nabla u) = f,$$

with $a(x,y,z) = 1$ for Problems 1, 3, 4, and 5. Problems 3 through 6 were discretized using the MFEM finite element package [19].

Problem 1: *S-7pt* - This is 7-point 3D Laplace problem on a cube generated by finite differences using $50 \times 50 \times 50$ grid points per core.

Problem 2: *S-27pt* - This is a 3D problem with a 27-point stencil on a cube using $50 \times 50 \times 50$ grid points per core.

Problem 3: *U-Cube* - This is a 3D unstructured Laplace problem on a cube with tetrahedral elements.

Problem 4: *U-Quadratic* - This is a 3D unstructured Laplace problem with quadratic finite elements on the Fichera domain illustrated in Figure 5.

Problem 5: *U-Cubic* - This is a 3D unstructured Laplace problem with cubic finite elements on the Fichera domain illustrated in Figure 5.

Problem 6: *U-Jumps* - This is a 3D unstructured diffusion problem on a sphere with trilinear hexahedral finite elements. It contains two material subdomains, with $a(x,y,z) = 1$ and $a(x,y,z) = 1000$. The domains are illustrated in Figure 6.

Implementation and experiment details. The various V-cycles were implemented in the BoomerAMG code [16] in the *hypre* library [15], however in order to have a fair comparison, we also rewrote the multiplicative V-cycle. We took advantage of the fact that $x_k = 0$ wherever feasible, including the finest level, since AMG was used here as a preconditioner for CG. This led to savings in communication for all

smoothers, and savings in flops for ℓ_1 Jacobi smoothing as well as for ℓ_1 Gauss-Seidel smoothing with small blocks, mostly on the finest level. The matrix data structure (ParCSR) consists of two matrices stored in CSR format, one for the local portion and one for the off processor part [11]. While we were able to save flops for ℓ_1 Jacobi and ℓ_1 Gauss-Seidel with small blocks using $x_k = 0$, there was no advantage to do so for the full ℓ_1 Gauss-Seidel smoother, since it would have required to check each index on whether it is in the lower or in the upper triangular part as well as a full sweep of the matrix. For small block ℓ_1 Gauss-Seidel we changed the data structure by splitting the local portion into two CSR matrices in order to generate computation-communication overlap and avoid the inefficiency mentioned above.

All test runs were performed on Hera, a Linux cluster at LLNL, with a fat tree network and a slow DDR interconnect. While on many current architectures multiplicative variants work well, we anticipate communication to be much more expensive on future exascale computers, and therefore chose an architecture with a relatively slow network, but fast cores, to demonstrate potential performance gains.

For all runs we used the settings in *hypre* that we recommend for 3-dimensional problems and that generally obtain the fastest performance, i.e HMIS [9] with extended+i interpolation truncated to at most 4 elements per row [10], and one level of aggressive coarsening with multipass interpolation [20, 24]. For the ℓ_1 Gauss-Seidel smoother, we used the variant given in (6.5) in [2].

We use the systematic naming scheme 'Variant.Level.Smoother':

- 'Variant' defines the version used, and we use 'mult' for a multiplicative cycle, 'add', for a classical additive cycle, 'ma' for the mult-additive variant, 'maP8' for the mult-additive variant using a smoothed interpolation that has been truncated to at most 8 nonzeroes per row, 'matr' for truncation with a factor of 0.025, 'sma' for the simplified mult-additive version and 'smaP8' for the truncated simplified version.
- 'Level' defines the first level (starting at '0') at which the variant was used.
- For 'Smoother', 'j' stands for ℓ_1 Jacobi, 'gs' for ℓ_1 Gauss-Seidel, 'gs20' for ℓ_1 Gauss-Seidel using block sizes of at most 20, 'gs1000' for ℓ_1 Gauss-Seidel with block sizes of 1000.

In Tables 6 through 11, we record number of iterations, memory usage, change factors for number of messages, amount of data sent and flops, as well as cycle times. For the methods using ℓ_1 Gauss-Seidel smoothing, the memory change factors in parentheses indicate necessary memory increases during AMG setup. For the sake of space, we listed change factors per cycle only, but since we give the number of iterations, one can easily adjust for increased number of iterations. Note that number of messages and amount of data is the same for a multiplicative V-cycle regardless whether ℓ_1 Jacobi or Gauss-Seidel smoothing is used. While these numbers give an idea of savings that can be obtained within a cycle, they do not take into account additional effects on performance, such as communication-computation overlap as well as cache use, nor consider the solver which is preconditioned by AMG. For the Gauss-Seidel flops, we use $\sum_{k=0}^{\ell-1}(4nnz(A_k) + 4nnz(\overline{P}_{k+1}^k))$ for the mult-additive cycle and $\sum_{k=0}^{\ell-1}(4nnz(A_k) + 4nnz(P_{k+1}^k))$ for the additive and multiplicative cycle.

Performance results. Figures 7 through 12 show the solve times we achieved for all test problems. We present solve times only, since we have not investigated nor used an efficient implementation of the setup, however we note here that computation and communication increases during setup, and setup time can increase significantly, particularly when using ℓ_1 Gauss-Seidel with large blocks starting on the finest level. However, for Jacobi and Gauss-Seidel with small blocks, we observed that setup times increased about 10-30 percent, when started on the finest level, and the increases became insignificant on coarser levels. Note that it is possible to combine the coarse grid generation $(P_{k+1}^k)^T A_k P_{k+1}^k$ with the generation of the smoothed interpolation $(I - M_k^{-T} A_k) P_{k+1}^k$ for a more efficient setup, which was not done in our experiments.

While the increase in setup times might make the methods less attractive, if only one system is solved, this issue is less significant in applications where the same system needs to be solved many times with different right hand sides, or in time stepping algorithms where it is possible to reuse the preconditioner for many time steps.

In all figures, solid lines generally show timings for variants using ℓ_1 Gauss-Seidel smoothing, whereas dashed, dotted and dash-dotted lines are used for Jacobi. For all problems, the use of ℓ_1 Gauss-Seidel leads to better convergence than ℓ_1 Jacobi, however generally one cycle with Jacobi is faster. Overall best cycle times are achieved with smaP8.0.j. For most problems considered here maP8.0.j is faster than mult.gs, and for all of them smaP8.0.j is faster than mult.gs in spite of its slower convergence.

For Problems S-7pt and S-27pt, we were unable to run maP8.0.gs, since we ran out of memory during setup; however it led to generally good solve times for the other problems, because of its good convergence. We achieved up to 50 percent improved performance compared to mult.gs using maP8 starting on the finest level using ℓ_1 Gauss-Seidel with block sizes of 20 or 1000. Convergence for both methods was similar. Decreasing the block size leads to a decerease in convergence, but to significantly improved memory usage and cycle efficiency. We obtained up to 80 percent improvement in speed compared to mult.gs and achieved a speedup of up to 2.5 compared to mult.j, using smaP8.

Overall, we found that starting maP8 and smaP8 at the finest level was faster than starting at a later level. However for the classical additive method, we achieved the best timings when starting at level 4. While in some cases the untruncated mult-additive method performed better than the multiplicative method, truncating the smoothed iterpolation always improved performance in spite of decreased convergence.

6. Conclusions

We have investigated mult-additive AMG, a new additive AMG variant with convergence properties equivalent to multiplicative AMG. Its improved convergence is achieved by smoothing the interpolation operators. We also investigated a simplified mult-additive method with potential for significant reduction in communication at the expense of decreased convergence. Since the smoothed interpolation operators can become quite expensive on the coarse grids, we introduce some new variants with truncated smoothed interpolation operators. When choosing appropriate truncation factors, these new methods only show a slight deterioration in convergence compared

Variant	iter.	memory	no. msgs	data sent	flops	cycle time
mult.j	34	1.00	1.000	1.000	1.000	0.218
add.4.j	44	1.00	0.621	0.968	0.999	0.165
ma.0.j	34	2.21	1.655	0.727	1.017	0.204
ma.1.j	34	1.14	1.659	0.957	1.033	0.226
maP8.0.j	36	2.10	0.918	0.623	0.911	0.140
maP8.1.j	36	1.04	0.922	0.857	0.933	0.159
matr.0.j	35	2.15	0.953	0.621	0.889	0.144
matr.1.j	35	1.09	0.957	0.871	0.900	0.161
sma.0.j	44	1.41	1.477	0.298	0.620	0.161
sma.1.j	44	1.14	1.481	0.798	0.930	0.192
smaP8.0.j	44	1.31	0.725	0.193	0.514	0.099
smaP8.1.j	44	1.04	0.729	0.697	0.830	0.125
mult.gs	26	1.00	1.000	1.000	1.000	0.218
add.4.gs	29	1.00	0.621	0.968	0.999	0.166
ma.2.gs	26	1.66	1.678	0.996	1.445	0.240
maP8.1.gs	27	1.04(11.05)	0.915	0.857	0.952	0.164
maP8.2.gs	26	1.01(1.64)	0.935	0.924	0.974	0.168
maP8.0.gs20	28	1.51	0.911	0.622	1.081	0.157

Table 6: Various quantities for Problem S-7pt, using 4096 cores. Change factors for no msgs, data sent, flops are given per cycle.

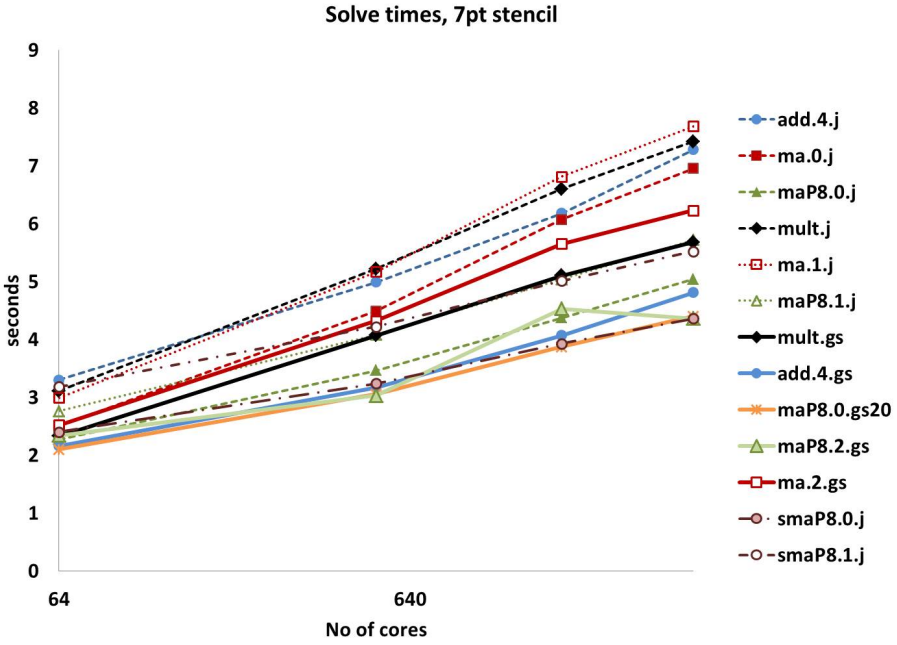


Figure 7: Solve times for Problem S-7pt.

Variant	iter.	memory	no. msgs	data sent	flops	cycle time
mult.j	37	1.00	1.000	1.000	1.000	0.275
add.4.j	46	1.00	0.673	0.973	1.000	0.238
ma.0.j	37	2.07	1.603	0.647	0.671	0.227
ma.1.j	37	1.01	1.628	1.002	1.003	0.294
maP8.0.j	40	2.05	0.857	0.571	0.652	0.168
maP8.1.j	40	1.00	0.883	0.926	0.992	0.223
matr.0.j	40	2.01	0.920	0.581	0.612	0.173
matr.1.j	39	1.00	0.947	0.940	0.994	0.227
sma.0.j	44	1.14	1.431	0.186	0.207	0.182
sma.1.j	46	1.01	1.457	0.919	0.992	0.267
smaP8.0.j	44	1.12	0.679	0.110	0.187	0.119
smaP8.1.j	46	1.00	0.704	0.843	0.981	0.200
mult.gs	28	1.00	1.000	1.000	1.000	0.315
add.4.gs	30	1.00	0.673	0.973	1.000	0.282
ma.2.gs	28	1.02	1.653	1.017	1.014	0.280
maP8.1.gs	30	1.00(1.22)	0.876	0.928	0.995	0.266
maP8.2.gs	29	1.00(1.02)	0.901	0.951	0.998	0.271
maP8.0.gs20	33	1.18	0.851	0.571	0.807	0.186

Table 7: Various quantities for Problem S-27pt using 4096 cores. Change factors for no msgs, data sent, flops are given per cycle.

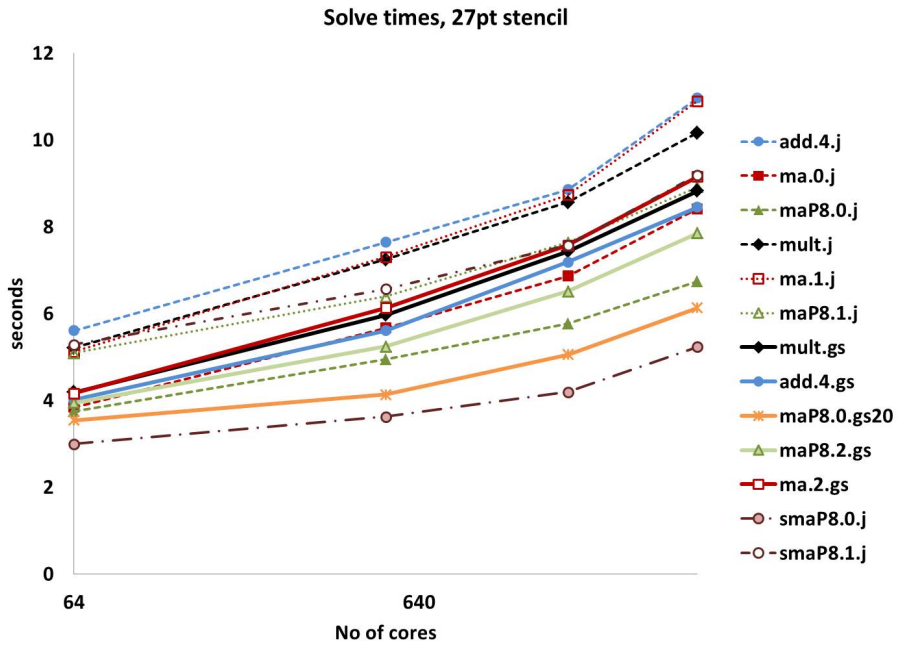


Figure 8: Solve times for Problem S-27pt.

Variant	iter.	memory	no. msgs	data sent	flops	cycle time
mult.j	48	1.00	1.000	1.000	1.000	0.151
add.4.j	56	1.00	0.670	0.967	1.000	0.106
ma.0.j	48	2.37	1.538	0.694	1.020	0.163
ma.1.j	48	1.11	1.568	0.953	1.028	0.168
maP8.0.j	51	2.18	0.869	0.577	0.834	0.099
maP8.1.j	51	1.02	0.892	0.843	0.932	0.100
sma.0.j	63	1.47	1.372	0.276	0.572	0.129
sma.1.j	53	1.12	1.395	0.803	0.941	0.138
smaP8.0.j	64	1.28	0.697	0.159	0.386	0.050
smaP8.1.j	52	1.02	0.720	0.693	0.845	0.069
mult.gs	31	1.00	1.000	1.000	1.000	0.163
add.4.gs	32	1.00	0.670	0.967	1.000	0.121
ma.2.gs	31	1.35	1.591	0.974	1.216	0.281
maP8.0.gs	33	1.31(9.08)	0.867	0.577	0.905	0.104
maP8.1.gs	32	1.02(3.28)	0.890	0.844	0.953	0.121
maP8.0.gs20	40	1.30(1.60)	0.868	0.577	0.895	0.093
maP8.0.gs1000	39	1.30(1.95)	0.867	0.577	0.899	0.092

Table 8: Various quantities for Problem U-Cube using 4096 cores. Change factors for no msgs, data sent, flops are given per cycle.

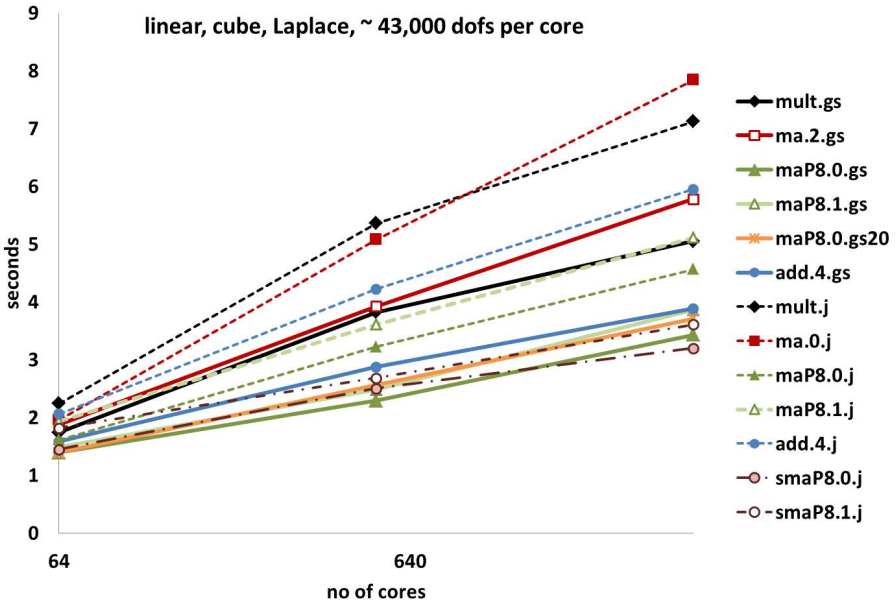


Figure 9: Solve times for Problem U-Cube.

Variant	iter.	memory	no. msgs	data sent	flops	cycle time
mult.j	33	1.00	1.000	1.000	1.000	0.197
add.4.j	39	1.00	0.733	0.982	1.000	0.167
ma.0.j	33	2.18	1.710	0.620	1.020	0.191
ma.1.j	33	1.02	1.740	0.982	1.028	0.212
maP8.0.j	35	2.05	1.006	0.538	0.604	0.119
maP8.1.j	35	1.00	1.036	0.912	0.986	0.147
sma.0.j	41	1.21	1.529	0.182	0.244	0.136
sma.1.j	36	1,02	1.559	0.893	0.985	0.186
smaP8.0.j	41	1.09	0.824	0.100	0.121	0.065
smaP8.1.j	35	1.00	0.854	0.823	0.969	0.116
mult.gs	21	1.00	1.000	1.000	1.000	0.234
add.4.gs	22	1.00	0.733	0.982	1.000	0.203
ma.2.gs	21	1.02	1.770	1.000	1.216	0.253
maP8.0.gs	21	1.09(12.78)	1.012	0.537	0.733	0.141
maP8.1.gs	21	1.00(1.21)	1.042	0.844	0.990	0.179
maP8.0.gs20	28	1.09(1.39)	1.012	0.538	0.733	0.124
maP8.0.gs1000	28	1.09(2.68)	1.012	0.539	0.733	0.126

Table 9: Various quantities for Problem U-Quadratic using 4096 cores. Change factors for no msgs, data sent, flops are given per cycle.

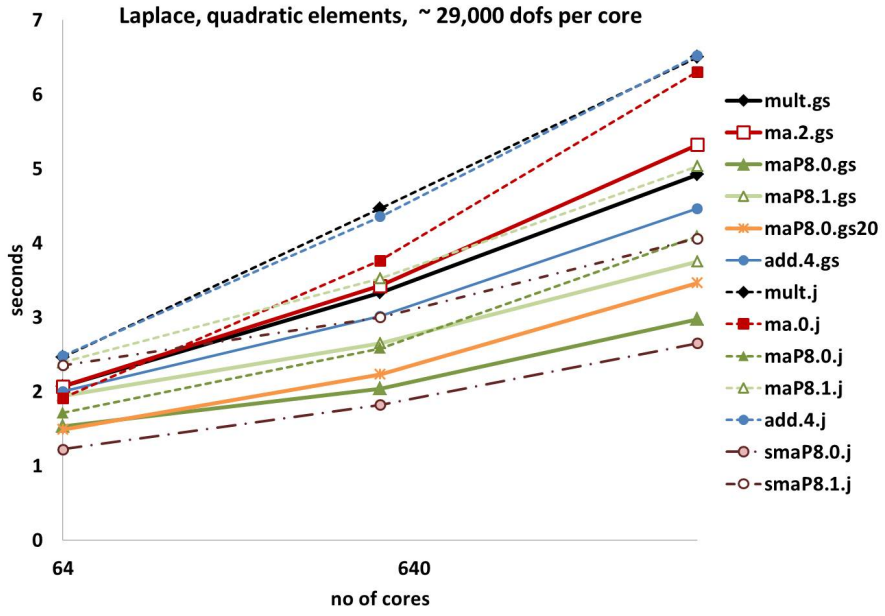


Figure 10: Solve times for Problem U-Quadratic.

Variant	iter.	memory	no. msgs	data sent	flops	cycle time
mult.j	56	1.00	1.000	1.000	1.000	0.240
add.4.j	67	1.00	0.725	0.985	1.000	0.197
ma.0.j	56	2.25	1.418	0.630	0.767	0.217
ma.1.j	56	1.03	1.448	0.966	1.002	0.248
maP8.0.j	61	2.03	0.828	0.536	0.557	0.132
maP8.1.j	60	1.00	0.860	0.904	0.976	0.175
sma.0.j	74	1.26	1.247	0.174	0.275	0.158
sma.1.j	58	1.03	1.278	0.877	0.977	0.219
smaP8.0.j	74	1.05	0.658	0.080	0.064	0.075
smaP8.1.j	58	1.00	0.690	0.815	0.984	0.150
mult.gs	29	1.00	1.000	1.000	1.000	0.279
add.4.gs	30	1.00	0.725	0.985	1.000	0.240
ma.2.gs	29	1.03	1.479	0.984	1.017	0.301
maP8.0.gs	30	1.05(7.54)	0.867	0.539	0.703	0.171
maP8.1.gs	30	1.00(1.34)	0.862	0.904	0.984	0.220
maP8.0.gs20	42	1.05(1.35)	0.831	0.538	0.703	0.138
maP8.0.gs1000	41	1.05(2.51)	0.830	0.538	0.703	0.143

Table 10: Various quantities for Problem U-Cubic using 2048 cores. Change factors for no msgs, data sent, flops are given per cycle.

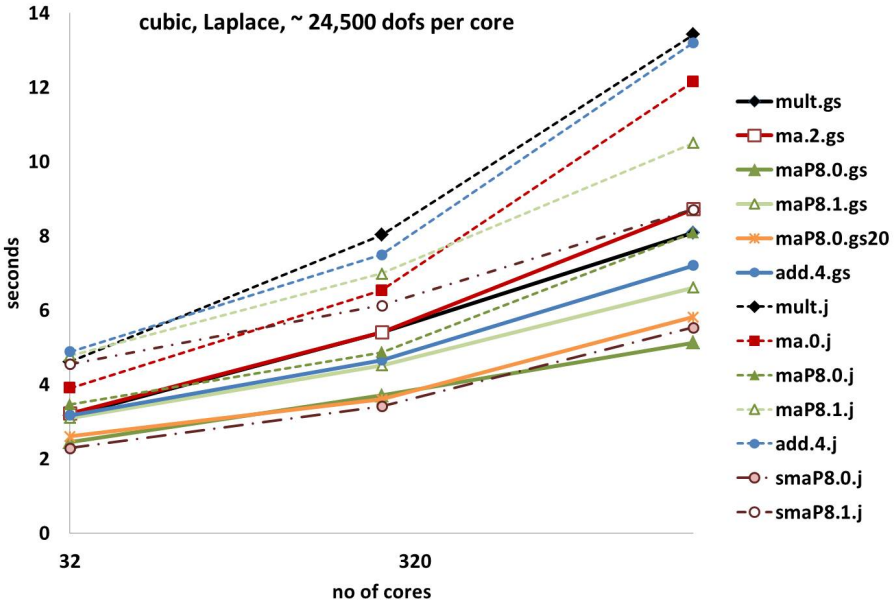


Figure 11: Solve times for Problem U-Cubic.

Variant	iter.	memory	no. msgs	data sent	flops	cycle time
mult.j	33	1.00	1.000	1.000	1.000	0.189
add.4.j	56	1.00	0.678	0.972	1.000	0.150
ma.0.j	33	2.27	1.706	0.693	0.889	0.198
ma.1.j	33	1.05	1.730	0.929	1.008	0.214
maP8.0.j	35	2.08	0.873	0.586	0.706	0.123
maP8.1.j	35	1.01	0.897	0.829	0.964	0.147
sma.0.j	43	1.35	1.524	0.281	0.430	0.148
sma.1.j	38	1.05	1.548	0.771	0.965	0.249
smaP8.0.j	43	1.17	0.691	0.173	0.247	0.068
smaP8.1.j	38	1.01	0.715	0.672	0.922	0.110
mult.gs	24	1.00	1.000	1.000	1.000	0.221
add.4.gs	26	1.00	0.678	0.972	1.000	0.181
ma.2.gs	24	1.10	1.752	0.980	1.058	0.247
maP8.0.gs	26	1.18(33.52)	0.874	0.583	0.806	0.146
maP8.1.gs	25	1.01(2.18)	0.898	0.829	0.976	0.178
maP8.0.gs20	29	1.17(1.53)	0.874	0.585	0.802	0.119
maP8.0.gs1000	29	1.17(2.25)	0.874	0.585	0.803	0.120

Table 11: Various quantities for Problem U-Jumps using 4096 cores. Change factors for no msgs, data sent, flops are given per cycle.

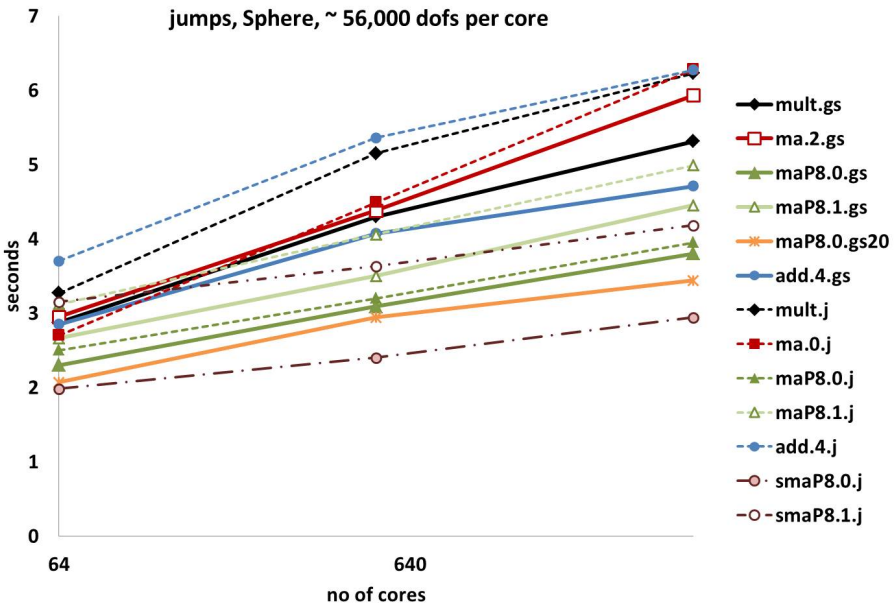


Figure 12: Solve times for Problem U-Jump.

to multiplicative multigrid, but converge significantly faster than additive multigrid. We analyzed the communication and computation cost as well as memory requirements for a V-cycle of the new methods in comparison to muliplicative AMG for ℓ_1 Jacobi and ℓ_1 Gauss-Seidel smoothing. We also presented performance results for several structured and unstructured problems. Use of Jacobi smoothing allows a very efficient implementation of the smoothing step, with a large reduction in communication per V-cycle and much improved communication-computation overlap. However, if one starts this approach at the finest level, memory requirements are increased by a factor of two, but we have observed improvements in speed up to 60 percent. Best performance was observed using the simplified approach which led to solve times improved by a factor up to 2.5 for some problems. Use of ℓ_1 Gauss-Seidel gives overall better convergence, but requires an unreasonable amount of memory during setup if the block size is too large. If the block size equals the degrees of freedom per core, it is also not possible to overlap computation and communication during smoothing. Using a smaller block size significantly reduces memory requirements, allows overlap, and leads to an efficient V-cycle. While convergence is impacted, this method still converges significantly faster than additive AMG and achieves solve times that are up to 50 percent faster than multiplicative AMG. The simplified approach has shown to lead to even faster solve times for several problems in spite of its slower convergence.

7. Acknowledgments

We thank Tzanio Kolev for providing the unstructured test problems. Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] P. Bastian, W. Hackbusch, and G. Wittum, "*Additive and Multiplicative Multi-Grid – A Comparison,*" Computing **60**(1998), pp. 345-364.

[2] A. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "*Multigrid Smoothers for Ultraparallel Computing,*" SIAM J. Sci. Comput. **33**(2011), pp. 2864-2887.

[3] A. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "*Scaling hypre's Multigrid solvers to 100,000 Cores,*" in M. Berry, K. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad and F. Saied (eds.), High Performance Scientific Computing: algorithms and Applications, Springer Verlag, 2012, pp. 261-279.

[4] A. Baker, T. Gamblin, M. Schulz and U. M. Yang, "*Challenges of Scaling Algebraic Multigrid across Modern Multicore Architectures,*" in 25th IEEE International Symposium on Parallel and Distributed Processing IPDPS 2011, Anchorage, AK, USA, 16-20 May, 2011 - Conference Proceedings, IEEE, 2011, pp. 275-286.

[5] Baker A, Schulz M, Yang UM, On the Performance of an Algebraic Multigrid Solver on Multicore Clusters, in VECPAR 2010, J. M. L. M. Palma et al. Eds. *Lecture Notes in Computer Science*, **6449**, Springer Verlag, 2011; 102-115.

[6] J. Bramble, J. Pasciak, and J. Xu, "*Parallel Multilevel Preconditioners,*" Mathematics of Computation **55** (1990), pp. 1-2.
[7] Tony F. Chan and Ray S. Tuminaro, "*Design and Implementation of Parallel Multigrid Algorithms*", in Steve F. McCormick (ed.), *Multigrid Methods: Theory, Applications, and Supercomputing*, Marcel Dekker, New York, 1988, pp. = 101–115.
[8] Tony F. Chan and Ray S. Tuminaro, "*Analysis of a Parallel Multigrid Algorithm*", in eds. J. Mandel, S. McCormick, J. E. Dendy, C. Farhat, G. Lonsdale, S. Parter, J. Ruge, K. Stüben (eds.), *Proceedings of the Fourth Copper Mountain Conference on Multigri d Methods*, SIAM, Philadelphia, 1989, p. = 66–86.
[9] H. De Sterck, U. M. Yang, and J. J. Heys, "*Reducing Complexity in Parallel Algebraic Multigrid Preconditioners,*" SIAM J. on Matrix Analysis and Applications **27** (2006), pp. 1019-1039.
[10] H. de Sterck, R. Falgout, J. Nolting, and U.M. Yang, "*Distance-Two Interpolation for Parallel Algebraic Multigrid,*" Numer. Linear Algebra Appl. **15** (2008), pp. 115-139.
[11] R. D. Falgout, J. E. Jones, and U. M. Yang, "*Pursuing Scalability for hypre's Conceptual Interfaces,*" ACM Trans. Math Softw. **31**(2005), pp. 326-350.
[12] Gahvari H, Baker A, Schulz M, Yang UM, Jordan K, Gropp W, Modeling the Performance of an Algebraic Multigrid Cycle, in *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ*, ACM, 2011; 172-181.
[13] L. Fournier and S. Lanteri, *Multiplicative and additive parallel multigrid algorithms for the acceleration of compressible flow computations on unstructured meshes*, Applied Numerical Mathematics, **36**, (2001), p. = 401–426
[14] A. Greenbaum, "*A Multigrid Method for Multiprocessors,*" Applied Mathematics and Computation **19**(1986), pp. 75-88.
[15] *hypre: High Performance Preconditioners*. http://www.llnl.gov/CASC/linear_solvers/.
[16] V. Henson, U. M. Yang, BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner, *Applied Numerical Mathematics* 41 (2002) 155–177.
[17] Kolev TV, Vassilevski PS, Parallel Auxiliary Space AMG for H(curl) Problems, *Journal of Computational Mathematics* 2009; **27**; 604-623.
[18] Lesley R. Matheson and Robert E. Tarjan, "*Parallelism in Multigrid Methods: How Much Is Too Much?*", International Journal of Parallel Programming, **24** (1996), pp. = 397–432.
[19] MFEM Finite Element Discretization Library, code.google.com/p/mfem/ .
[20] K. Stüben, Algebraic multigrid (AMG): an introduction with applications, in : U. Trottenberg, C. Oosterlee and A. Schüller, eds., *Multigrid* (Academic Press, 2000).
[21] Ray S. Tuminaro, "*A Highly Parallel Multigrid-Like Method for the Solution of the Euler Equations*", SIAM Journal on Scientific and Statistical Computing, **13** (1992), pp = 88–100.
[22] Vampir 8.0, http://www.vampir.eu .
[23] Panayot S. Vassilevski, "Multilevel Block Factorization Preconditioners, Matrix-based Analysis and Algorithms for Solving Finite Element Equations," Springer, New York, 2008. 514 p.
[24] U. M. Yang, "*On Long Range Interpolation Operators for Aggressive Coarsening,*" Numer. Linear Algebra Appl. **17** (2010), pp. 453-472.