LLNL-JRNL-812088

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

A New Class of AMG Interpolation Methods Based on Matrix-Matrix Multiplications

R. Li, B. Sjogreen, U. M. Yang

June 26, 2020

SIAM Journal of Scientific Computing

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A NEW CLASS OF AMG INTERPOLATION METHODS BASED ON MATRIX-MATRIX MULTIPLICATIONS

RUIPENG LI*, BJORN SJOGREEN*, AND ULRIKE MEIER YANG*

Abstract. A new class of distance-two interpolation methods for algebraic multigrid (AMG) that can be formulated in terms of sparse matrix-matrix multiplications is presented and analyzed. Compared with current similar methods, the proposed algorithms exhibit improved efficiency and portability to various computing platforms, since they allow to easily exploit existing high-performance sparse matrix kernels. The new interpolation methods have been implemented in hypre [15], a widely-used parallel multigrid solver library. With the proposed interpolations, the overall time of hypre's BoomerAMG setup can be considerably reduced, while sustaining equivalent, sometimes improved, convergence rates. Numerical results for a variety of test problems on parallel machines are presented that support the superiority of the proposed interpolation operators over the existing ones in hypre.

Key words. AMG interpolation; sparse matrix computation; parallel and distributed computing

1. Introduction.

The problem of solving linear systems with large sparse matrices often arises in many fields of science and engineering when computing the numerical solution of partial differential equations (PDEs). Algebraic multigrid (AMG) [6, 29, 31] has been an efficient scalable parallel solver or preconditioner for such systems due to its numerical scalability and good coarse-grain parallelism [4]. However the emergence of recent high performance computers, which achieve their largest performance potential through the use of accelerators, such as graphics processing units (GPUs), presents some serious challenges to an efficient AMG implementation. GPUs require a high level of small grain parallelism, which is hard to achieve with unstructured sparse matrices, generally based on a compressed sparse row (CSR) data structure. In addition, best performance is achieved for large matrices and vectors, however the hierarchy of AMG requires dealing with decreasing matrix sizes on coarser levels. Nevertheless, despite these issues, much progress has been made on porting AMG and other sparse solvers on GPUs, see, e.g., [5, 8, 9, 16, 21, 22, 26, 28, 30, 33].

AMG consists of a setup and a solve phase. In the setup phase, for each level, a coarsening algorithm is applied to generate the variables for the next level, a prolongation operator P and a restriction operator R, often defined as $R = P^T$, and the coarse grid operator RAP are generated. During the solve phase, the error is smoothed with a few sweeps of a generally simple iterative method such as Jacobi or Gauss-Seidel, and R and P are used to move between levels. For more details see [14]. When choosing highly parallel smoothers [3] based on matrix-vector operations, such as Jacobi or polynomial smoothers, fairly good performance on GPUs can be achieved for the AMG solve cycle. It is more difficult to port AMG setup phase. The coarse grid operator is obtained by performing two sparse matrix-matrix operations or a triple matrix product, which has also been very challenging. There has however been recent progress on the development of efficient sparse matrix-matrix multiplications [10, 13, 17, 23, 25] on GPUs. Commonly used coarsening schemes such as HMIS [12] that lead to good converging methods possess larger grain parallelism, but the individual processes contain sequential portions, not suitable for GPUs. However, PMIS [12] is a highly parallel coarsening scheme and amenable for porting to GPUs. It is a modification of a par-

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P. O. Box 808, L-561, Livermore, CA 94551 ({li50,sjogreen2,yang11}@llnl.gov). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-812088).

allel maximal independent set algorithm by Luby [24]. To achieve good convergence, it often needs to be combined with distance-two interpolation, such as the extended and the extended+i approaches [11]. Current implementations of these methods, as can be found in hypre [15], are not directly appropriate for GPUs. While they are parallel, i.e. each row of the interpolation matrix can be computed independently, the computation of the interpolation weights requires many conditional statements and parsing of branches that inhibit efficient execution on accelerators.

We propose here a new class of interpolation operators, which we will call "MM-interpolation operators". They are similar to extended and extended+i interpolation, but can be computed on GPUs more efficiently, since they are composed of basic sparse matrix and vector kernels, including sparse matrix-matrix (MM) multiplications. While our original goal was to develop interpolation operators suitable for GPUs, it turns out that the new methods also perform better on CPUs. We investigate their convergence, performance and implementation differences. We additionally propose several new long range MM-interpolation operators [34] that can be used in combination with more aggressive coarsening, which can have additional performance benefits by decreasing operator and memory complexities. We present numerical results for a variety of problems on a parallel computer utilizing its CPUs.

We adopt the following notations from [7]. The cardinality of a set s is denoted by $|s|$. The number of unknowns of the linear system is n. The index set $\{1, 2, \ldots, n\}$ is partitioned into fine and coarse points, F and C, by coarsening algorithms, and $n_f = |F|$ and $n_c = |C|$. The (i, j)-entry of a matrix A is denoted by a_{ij} or $[A]_{ij}$. By N_i we denote the neighborhood of i, i.e., $N_i = \{j \mid a_{ij} \neq 0, j \neq i\}$, which can be divided into N_i^s and N_i^w consisting of the points that strongly and weakly influence i respectively. By definition, x_i is strongly influenced by x_j if $-a_{ij} \geq \theta \max_{k \neq i}\{-a_{ik}\}$. Also, we define $F_i^s = N_i^s \cap F$ and $C_i^s = N_i^s \cap C$.

2. History of extended and extended+i interpolation. Interpolation formulae for AMG are generally based on the characteristic of the smoothness of an error vector x, i.e.,

$$a_{ii}x_i + \sum_{j \in F_i^s} a_{ij}x_j + \sum_{k \in C_i^s} a_{ik}x_k + \sum_{m \in N_i^w} a_{im}x_m \approx 0. \tag{2.1}$$

Since smooth error varies slowly in the direction of strong connections, the error x_j for F-points $j \in F_i^s$ can be expressed by the errors associated with C_i^s as

$$x_j \approx \sum_{l \in C_i^s} \frac{a_{jl}}{\sum_{m \in C_i^s} a_{jm}} x_l. \tag{2.2}$$

This is also called "collapsing the stencils" [14]. Dividing by the sum of connections to strong C-points in (2.2) allows to interpolate constants exactly. Since the weak couplings with N_i^w are less important, we simply replace x_m with x_i. Therefore, this allows the following definition of the so-called "classical" interpolation [29, 32]:

$$w_{ik} = -\frac{a_{ik} + \displaystyle\sum_{j \in F_i^s} \frac{a_{ij}a_{jk}}{\sum_{l \in C_i^s} a_{jl}}}{a_{ii} + \sum_{m \in N_i^w} a_{im}}, \quad k \in C_i^s. \tag{2.3}$$

Due to a condition that strongly connected F-points require a common C-point, which was guaranteed in the original AMG coarsening algorithm [29], and since many of the

targeted matrices coming from PDEs are M-matrices, where off-diagonal elements are of the opposite sign as the diagonal elements, the sum $\sum_{l\in C_i^s} a_{jl}$ in the denominator generally does not vanish. However if A is not an M-matrix, it is possible that, assuming a positive diagonal, a large positive off-diagonal element can lead to a cancellation. Note that even if the matrix A at the finest level is an M-matrix, this might not be the case for matrices on the coarser levels. In order to avoid this situation, (2.3) was modified by using $\bar{a}_{jk}$ and $\bar{a}_{jl}$ defined as

$$\bar{a}_{jm} = \begin{cases} 0 & \text{if } \operatorname{sign}(a_{jm}) = \operatorname{sign}(a_{jj}) \\ a_{jm} & \text{otherwise.} \end{cases}, \quad m = k, l \tag{2.4}$$

in the place of a_{jk} and a_{jl} respectively.

The classical Ruge-Stüben coarsening [29], which uses a second pass to enforce that two fine points have a common coarse point, can lead to large complexities and unsuitability for massively parallel computing, so PMIS and HMIS coarsening [12] were introduced to address such problems. Specifically, the second pass was dropped in HMIS and a highly parallel approach based on Luby's method [19, 24] was used in PMIS. It was shown that using the classical interpolation (2.3) in combination with PMIS often leads to poor convergence [11]. As a remedy, interpolation operators that can reach additional C-points at a larger range can yield better convergence, such as the extended interpolation [11], for which an augmented interpolatory set

$$\hat{C}_i = \bigcup_{j\in\{\{i\}\cup F_i^s\}} C_j^s, \tag{2.5}$$

was defined. Replacing C_i^s with $\hat{C}_i$ in (2.3) leads to the extended interpolation formula

$$w_{ik} = -\frac{a_{ik} + \sum_{j\in F_i^s} \frac{a_{ij}\bar{a}_{jk}}{\sum_{l\in\hat{C}_i} \bar{a}_{jl}}}{a_{ii} + \sum_{m\in\{N_i^w\setminus\hat{C}_i\}} a_{im}}, \quad k \in \hat{C}_i\ . \tag{2.6}$$

It contains an additional change: since it is now possible for weak connections from i to $\hat{C}_i$ to be included in the numerator, weak connections that end up being included should no longer be added to the diagonal, to correctly interpolate constants.

Another variant that is referred to as the "extended+i" interpolation, includes an extra connection between a point in F_i^s and i itself, which, specifically, is given by

$$w_{ik} = -\frac{a_{ik} + \sum_{j\in F_i^s} \frac{a_{ij}\bar{a}_{jk}}{\sum_{l\in\{\hat{C}_i\cup\{i\}\}} \bar{a}_{jl}}}{a_{ii} + \sum_{m\in\{N_i^w\setminus\hat{C}_i\}} a_{im} + \delta_i}, \quad \delta_i = \sum_{j\in F_i^s} \frac{a_{ij}a_{ji}}{\sum_{l\in\{\hat{C}_i\cup\{i\}\}} \bar{a}_{jl}}, \quad k \in \hat{C}_i\ . \tag{2.7}$$

This approach can yield better convergence than (2.6) for certain cases [11], which will be also shown in Section 8. Once the interpolation operators are built, truncation is typically deployed to control complexity. A common strategy of truncation is dropping entries with small magnitudes and rescaling the remaining interpolation weights.

Implementing extended+i interpolation can be fairly complex. First, one needs to determine the interpolatory set for each F-point. Next, the computation of the numerical values requires checking whether each connection a_{jk} is in the interpolatory set and is weak or strong, which leads to a large number of if-statements on various

branches. While good coarse-grain parallelism can be achieved once the communication pattern is established, the algorithms exhibit little fine-grain parallelism needed for efficient use of modern many-core architectures. Park et al. [27] implemented a threaded version of the extended+i interpolation in hypre, which required sophisticated hash routines to achieve reasonable speedups. A more in-depth analysis of the current implementations is described in Section 6.

3. The class of MM interpolation operators. To achieve a fine-grain parallel implementation of extended and extended+i interpolation, we decided to approach their definition from a different angle. Instead of focusing on the individual weights we took a more holistic view of the interpolation operator and came up with a formula that is based on sparse MM multiplications. An obvious advantage of this approach is that it simplifies the implementation by utilizing existing optimized sparse kernels on various computing platforms and thus increases portability. As we will see in this section, the new interpolation operators can be presented in a compact form that consists of MM multiplications and simple vector operations. Before we present this formulation, we relate the new weights to the weights of the extended interpolation. Replacing C_i^s in (2.2) with C_j^s and substituting it into (2.1), a new interpolation formula can be written as

$$w_{ik} = -\frac{\hat{a}_{ik} + \displaystyle\sum_{j \in F_i^s} \frac{a_{ij}\hat{a}_{jk}}{\sum_{l \in C_j^s} a_{jl}}}{a_{ii} + \sum_{m \in N_i^w} a_{im}}, \quad k \in \hat{C}_i, \tag{3.1}$$

where

$$\hat{a}_{mk} = \begin{cases} a_{mk} & k \in C_m^s \\ 0 & k \notin C_m^s \end{cases}, \quad m = i, j. \tag{3.2}$$

The main difference in (3.1), compared with (2.6), is that all coefficients in the numerator are associated with strong connections only, and therefore all weak connections a_{im}, $m \in N_i^w$ are now included in the denominator. Moreover, we no longer require the modification in (2.4) that prevents cancellation, since according to the definition of the strength of connections (SoC), coefficients with the same sign as the diagonal element will be eliminated as weak connections. These changes simplify the evaluation of the interpolation weights and, as we will see later, also allow to express the whole interpolation operator in a clean multiplication formula. The modification to the extended+i interpolation (2.7) is similar.

In the next sections, we use the following notations: $\text{diag}(v)$ denotes the diagonal matrix with vector v as the diagonal, and $\text{diag}(A)$ is the vector that consists of the diagonal of matrix A. Thus, $\text{diag}(\text{diag}(A))$ is the diagonal matrix that consist of the diagonal of A. Let $e_f \in \mathbb{R}^{n_f}$ and $e_c \in \mathbb{R}^{n_c}$ be the vectors of all ones, and e_i the i-th canonical basis vector. We suppose A is partitioned per the F/C splitting as the 2-by-2 block structure

$$\begin{pmatrix} A_{FF} & A_{FC} \\ A_{CF} & A_{CC} \end{pmatrix}, \tag{3.3}$$

and also $A = D + A^s + A^w$, where $D = \text{diag}(\text{diag}(A))$, and A^s and A^w contain the off-diagonal strong and the weak connections according to the SoC respectively. Using the F/C splitting we can present the interpolation operator P in the form $P = \left(\begin{smallmatrix} W \\ I \end{smallmatrix}\right)$, where $W \in \mathbb{R}^{n_f \times n_c}$ contains the interpolation weights.

3.1. "MM-ext" interpolation. We can now define the MM form of the new extended interpolation formula (3.1), to which we refer as the MM-ext interpolation. Using D_{FF}, A^s_{FF} and A^s_{FC} as the submatrices conforming to (3.3), it follows that the matrix W can be computed as the following multiplication

$$W = -\left[(D_{FF} + D_\gamma)^{-1}(A^s_{FF} + D_\beta)\right]\left[D_\beta^{-1} A^s_{FC}\right] \equiv -\tilde{A}^s_{FF}\tilde{A}^s_{FC}, \tag{3.4}$$

where

$$D_\beta = \operatorname{diag}(A^s_{FC} e_c), \tag{3.5}$$

$$D_\gamma = \operatorname{diag}(A^w_{FF} e_f + A^w_{FC} e_c), \tag{3.6}$$

i.e. D_β contains the row sums of A^s_{FC} and D_γ has the row sums of A^w_{FF} and A^w_{FC}. More specifically, each element of $\tilde{A}^s_{FF}$ and $\tilde{A}^s_{FC}$ can be obtained as

$$[\tilde{A}^s_{FF}]_{ij} = \begin{cases} \alpha_i^{-1}[D_\beta]_{ii}, & i = j \\ \alpha_i^{-1}[A^s_{FF}]_{ij}, & i \neq j \end{cases}, \quad \text{with} \quad \alpha_i = [D_{FF}]_{ii} + [D_\gamma]_{ii}\ , \tag{3.7}$$

$$[\tilde{A}^s_{FC}]_{ij} = [D_\beta]^{-1}_{ii}[A^s_{FC}]_{ij}\ . \tag{3.8}$$

Notice that (3.8) requires $[D_\beta]_{ii} \neq 0$ if $i \in F^s_k$ for some $k \in F$, which usually can be ensured by coarsening algorithms since i and k should share at least one C-point in their interpolatory sets. However, if this is not the case, we can simply fix it by replacing $[D_\beta]^{-1}_{ii}$ with 0 and add a_{ki} to $[D_\gamma]_{kk}$. Lastly, we remark here that D_{FF}, D_β, D_γ as well as the diagonal scaling from the left $D_\beta^{-1}A^s_{FC}$ in (3.4) can all be determined locally without communicating with other processes when A is distributed row-wise as in the parallel CSR format in hypre's AMG implementation BoomerAMG [18].

3.2. "MM-ext+i" interpolation. We discuss the MM multiplication form of the new extended+i interpolation, which we call the MM-ext+i interpolation. It is based on the formula (2.7) using the same modification used to get (3.1) from (2.6). As before, the formula of MM-ext+i changes by adding i into the interpolatory set of j along with C^s_j. As in MM-ext, all the weak connections of i are added to the diagonal and only strong connections are included in the remainder of the formula. Figure 3.1 illustrates the computation pattern of a row of W corresponding to $i \in F$ in both MM-ext and MM-ext+i, and the difference in MM-ext+i by the inclusion of the point i. The most significant difference in computing this interpolation as opposed

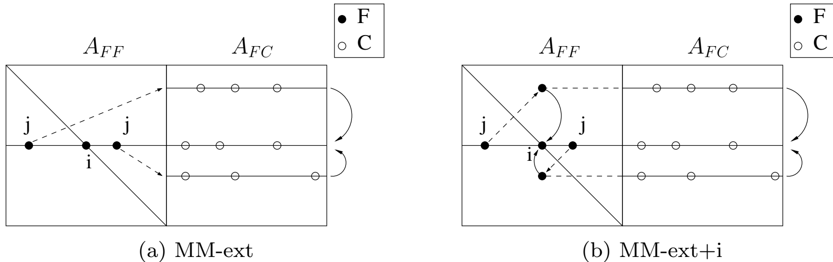


FIG. 3.1. *Extended and Extended+i interpolations in MM multiplication forms*

to MM-ext is that the scaling matrix D_β in (3.4) is no longer constant for all the F-points, but depends on each i, which we now denote by $D_{\beta(i)}$, for $i \in F$. As a result,

the diagonal scaling cannot be easily absorbed inside $\tilde{A}^s_{FC}$ as in (3.4), whereas we scale (from the right) each row i of A^s_{FF} with (a different) $D^{-1}_{\beta^{(i)}}$ instead. Therefore, it follows that the matrix W of the MM-ext+i interpolation takes the form

$$W = -\left[(D_{FF} + D_\gamma + D_\theta)^{-1}(\hat{A}^s_{FF} + I)\right] A^s_{FC} \equiv -\tilde{A}^s_{FF} A^s_{FC}, \tag{3.9}$$

where for $i \in F$

$$e_i^\mathsf{T} \hat{A}^s_{FF} = e_i^\mathsf{T} A^s_{FF} D^{-1}_{\beta^{(i)}}, \tag{3.10}$$

$$D_{\beta^{(i)}} = \mathrm{diag}(A^s_{FF} e_i) + D_\beta, \tag{3.11}$$

the terms $e_i^\mathsf{T} A^s_{FF}$ and $A^s_{FF} e_i$ are the i-th row and column of A^s_{FF} respectively, and

$$D_\theta = \mathrm{diag}(\mathrm{diag}(\hat{A}^s_{FF} A^s_{FF})). \tag{3.12}$$

We note here that to compute (3.12) only the diagonal of $\hat{A}^s_{FF} A^s_{FF}$ is needed, i.e. one should not compute the whole product. Due to scaling from the right in (3.10), communications for the off-process entries of $D_{\beta^{(i)}}$ are needed, in a distributed code. The invariant part, namely D_β, that does not depend on i, can be computed and communicated once, with the same communication pattern as in a sparse matrix-by-vector with A^s_{FF}. For determining the other part that varies with i, i.e, $A^s_{FF} e_i$, it is important to realize that we do not need to gather the entire i-th column of A^s_{FF} on each process. Only the elements corresponding to F^s_i are needed, since they are just needed to scale $e_i^\mathsf{T} A^s_{FF}$. Hence, it is required to retrieve the off-process rows of A^s_{FF} that correspond to F^s_i, which has the same communication pattern that is needed in performing the MM multiplication $(A^s_{FF})^2$. Furthermore, it is not difficult to see that D_θ can be computed in a similar way without requiring more communication. More details on the parallel implementation will be given in Section 5.

3.3. "MM-ext+e" interpolation. The increased complexity of MM-ext+i over MM-ext, particularly the additional requirement of communicating the extra matrix rows, are caused by the fact that the interpolatory set of j includes i, i.e.,

$$x_j \approx \sum_{l \in C^s_j \cup \{i\}} \frac{a_{jl}}{\sum_{m \in C^s_j \cup \{i\}} a_{jm}} x_l \ . \tag{3.13}$$

We investigate a simplified approach by replacing a_{ji} with an estimate μ_j by averaging the connections in F^s_j, i.e.,

$$x_j \approx \frac{\sum_{l \in C^s_j} a_{jl} x_l + \mu_j x_i}{\sum_{l \in C^s_j} a_{jl} + \mu_j} \quad \text{with} \quad \mu_j = \sum_{m \in F^s_j} a_{jm} |F^s_j|^{-1} \ , \tag{3.14}$$

where $|F^s_j|$ denotes the number of points in F^s_j. Apparently, the coefficients in (3.14) do not vary with i for different j, and approximating a_{ji} with μ_j makes sense, particularly for isotropic problems. With D_β and D_γ defined as before, it is straightforward to derive the expression of the W matrix as

$$W = -\left[(D_{FF} + D_\gamma + D_\tau)^{-1} (A^s_{FF} + D_\lambda)\right] \left[D^{-1}_\lambda A^s_{FC}\right] \equiv -\tilde{A}^s_{FF} \tilde{A}^s_{FC}, \tag{3.15}$$

with

$$[D_\lambda]_{ii} = [D_\beta]_{ii} + \mu_i, \tag{3.16}$$

$$[D_\tau]_{ii} = \textstyle\sum_{j \in F^s_i} a_{ij} \mu_j [D_\lambda]^{-1}_{jj}. \tag{3.17}$$

We remark here that, compared with MM-ext+i, there is no diagonal scaling from the right or varying with different $i \in F$ anymore; the matrix D_τ plays the same role as D_θ for including the contribution from i itself, but is much easier to compute; and lastly D_λ can now be determined locally while D_τ requires communicating the value of $\mu_j [D_\lambda]_{jj}^{-1}$ for off-process index j with neighboring processes. This eliminates the communication of complete matrix rows needed for MM-ext+i. We refer to this interpolation as the MM-ext+e interpolation, since the connection to point i has been replaced by an estimate.

4. Convergence comparisons. In this section, we compare the convergence of the new interpolation operators with the original extended and extended+i interpolation for a variety of problems. We consider first a few two-dimensional problems, which include two Poisson problems, one with a five-point stencil and a second one with a nine-point stencil on a 1000×1000 grid, as well as two problems with anisotropies of 0.001 rotated by 45 and 60 degrees on a 512×512 grid. Since our final goal is the implementation on GPUs, we choose a highly parallel coarsening scheme and smoother, i.e. PMIS coarsening and Jacobi smoothing with a weight of 0.85. For the anisotropy rotated by 60 degrees, which is a hard problem for AMG, we use two sweeps of Jacobi with a weight of 0.5. We use a zero initial guess and a random right-hand side, and stop convergence when the relative residual is smaller than 10^{-8}. We use truncation to at most 4 coefficients per row for the interpolation operator, which is the default in hypre. We record the operator complexities (C_{op}), which are defined by the sum of the numbers of nonzeros of all the A operators in the AMG hierarchy divided by the number of nonzeros of A, the number of iterations for AMG as a solver and the setup times. For the anisotropy rotated by 60 degrees, we also record the number of conjugate gradient (CG) iterations when using AMG as the preconditioner. We denote the original extended and extended+i interpolations by “ext” and “ext+i” respectively.

As reported in Tables 4.1 and 4.2, we see overall similar operator complexities and setup times with some slight improvements for the new interpolation routines for the 9-point problem. Overall, we achieve similar convergence for the corresponding interpolation routines, except for MM-ext when applied to the anisotropy rotated by 60 degrees.

Table 4.1

AMG for the 5-point and the 9-point 2D Poisson problem on a 1000×1000 square with random right-hand side using different interpolation operators.

Method	5-point			9-point		
	C_{op}	#its	time	C_{op}	#its	time
ext	2.42	29	1.28	1.53	18	1.57
ext+i	2.40	24	1.40	1.52	18	1.75
MM-ext	2.42	29	1.26	1.53	19	1.37
MM-ext+i	2.40	24	1.29	1.53	19	1.43
MM-ext+e	2.40	24	1.27	1.52	18	1.40

Next, we consider several three-dimensional problems on a $80 \times 80 \times 80$ grid: a 7-point and a 27-point Poisson problem, and a problem with jumps of 6 orders of magnitude. In Table 4.3, we see overall similar convergence and operator complexities, however the setup times for the MM-ext* operators are clearly lower than their ext* counterparts, particularly for the 27-point problem, where they take only about half the amount of time as the original prolongation operators. We further investigate the

Table 4.2

AMG for a problem with by 45 degrees and by 60 degrees rotated anisotropies on a 512×512 square mesh using different interpolation operators. The number of iterations of AMG preconditioned CG is shown in parentheses.

Method	45 degrees			60 degrees		
	C_{op}	#its	time	C_{op}	#its	time
ext	2.06	41	0.27	2.49	285 (42)	0.46
ext+i	2.06	25	0.27	2.63	168 (33)	0.50
MM-ext	2.06	41	0.28	2.39	444 (52)	0.44
MM-ext+i	2.06	25	0.29	2.57	149 (31)	0.47
MM-ext+e	2.06	25	0.29	2.57	158 (31)	0.48

reasons for this in Section 6.

Table 4.3

AMG for the 7-point and the 27-point 3D Poisson problem, and a problem with jumps on a $80 \times 80 \times 80$ *cube with a random right-hand side using different interpolation operators.*

Method	7-point			27-point			Jumps		
	C_{op}	#its	time	C_{op}	#its	time	C_{op}	#its	time
ext	2.80	23	1.49	1.21	15	3.20	2.92	47	1.50
ext+i	2.74	20	1.63	1.21	15	3.51	2.86	46	1.63
MM-ext	2.85	23	1.18	1.21	15	1.61	2.94	45	1.18
MM-ext+i	2.77	21	1.25	1.21	15	1.82	2.89	45	1.24
MM-ext+e	2.77	21	1.20	1.21	15	1.67	2.89	47	1.21

Finally, we include a few results to evaluate scalability and different types of parallelism of AMG setup with the proposed interpolation algorithms when used as a preconditioner of CG. For these experiments we used a right-hand side of all ones. One of the advantages in computing the MM-ext* interpolation operators is the large amount of fine-grain parallelism that allows straightforward threading, which is much simpler than previous efforts in [27]. The numerical experiments were conducted on a Linux workstation with a 12-core CPU, where the code was compiled by the `mpicc` compiler with optimization level -O2 and OpenMP (with hypre's configure options, `--with-openmp --enable-hopscotch --enable-persistent`). In Tables 4.4 and 4.5, we report the AMG setup times and the total times for the sequential runs, the runs using 12 MPI tasks and 1 thread for each task (labelled as 'p-n'), and the runs with 1 MPI task and 12 OpenMP threads (labelled as 't-n'), for solving the 7pt and the 27pt problems on $n \times n \times n$ grids.

As before, we achieved improved setup times for the MM-ext* interpolation operators, where, in particular, the 27-point problems showed the best improvements. This is the case for all types of experiments: sequential, distributed and threaded parallelism. Also, as in the previous experiments, we see equivalent convergence, in some cases even slightly improved convergence. Since the original extended interpolation in hypre has not been threaded, we do not present timings for this case.

5. Parallel implementation. All the aforementioned interpolation algorithms have been implemented in hypre. In this section, we discuss some implementation details, especially, with the focus on the efficient implementations for distributed memory environments. In hypre, distributed sparse matrices are stored in the parallel CSR format (ParCSR), for which each process owns a chunk of consecutive rows

TABLE 4.4
AMG setup times with the different interpolation operators considered for 3D 7pt and 27pt Poisson problems of size n^3

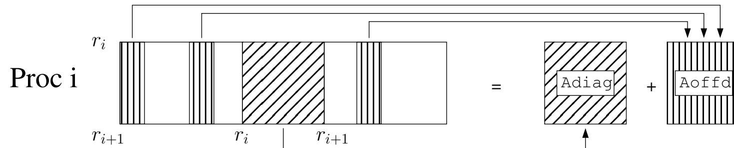
	7pt					27pt				
n	ext	ext+i	MM-ext	MM-ext+i	MM-ext+e	ext	ext+i	MM-ext	MM-ext+i	MM-ext+e
40	0.25	0.27	**0.21**	0.22	**0.21**	0.45	0.49	**0.25**	0.27	0.26
80	1.48	1.63	**1.19**	1.24	1.20	3.06	3.44	**1.51**	1.70	1.54
120	4.90	5.40	3.85	4.03	**3.84**	10.55	11.83	**5.40**	6.17	5.53
p-120	0.84	0.91	**0.69**	0.76	**0.69**	1.68	1.79	**0.91**	1.07	0.95
p-160	2.03	2.08	**1.53**	1.62	1.55	3.70	4.15	**2.09**	2.55	2.10
t-120	—	0.98	**0.82**	0.87	0.86	—	1.91	**1.14**	1.28	1.16
t-160	—	2.37	**1.92**	2.04	1.99	—	4.57	**2.69**	2.94	2.76

TABLE 4.5
Total times and number of iterations (in parentheses) for AMG-PCG with various interpolation operators for a 3D 7pt and 27pt diffusion problem of size n^3

	7pt					27pt				
n	ext	ext+i	MM-ext	MM-ext+i	MM-ext+e	ext	ext+i	MM-ext	MM-ext+i	MM-ext+e
40	0.33	0.35	**0.29**	0.30	**0.29**	0.53	0.58	**0.35**	0.36	0.36
	(12)	(12)	(12)	(12)	(12)	(9)	(9)	(10)	(9)	(10)
80	2.18	2.32	1.94	1.94	**1.89**	3.86	4.25	2.41	2.64	**2.33**
	(14)	(13)	(14)	(13)	(13)	(11)	(11)	(11)	(11)	(11)
120	7.49	7.92	6.66	6.58	**6.22**	13.53	14.82	**8.39**	9.17	8.53
	(14)	(14)	(15)	(14)	(13)	(12)	(12)	(12)	(12)	(12)
p-120	1.68	1.74	1.59	1.56	**1.47**	2.71	2.90	**1.94**	2.09	1.97
	(14)	(14)	(15)	(14)	(13)	(12)	(13)	(12)	(12)	(12)
p-160	4.16	4.20	3.68	3.61	**3.54**	6.35	7.02	**4.74**	5.20	4.76
	(15)	(15)	(15)	(14)	(14)	(13)	(14)	(13)	(13)	(13)
t-120	—	1.85	1.77	1.74	**1.67**	—	2.94	**2.18**	2.32	2.20
	—	(14)	(15)	(14)	(13)	—	(12)	(12)	(12)	(12)
t-160	—	4.63	4.21	4.15	**4.10**	—	7.45	**5.36**	5.61	5.44
	—	(15)	(15)	(14)	(14)	—	(14)	(13)	(13)	(13)

that are further split into the "on-process" and the "off-process" CSR matrix blocks. A clear advantage of this splitting is that typically in parallel matrix operations, the computations involved with the on-process block can be overlapped with the communications. Figure 5.1 illustrates the storage of the local rows, and the splitting into the "diag" and the "offd" matrices. Similarly, vectors are distributed conformingly to the sparse matrix and saved in the parallel vector (ParVec) format.

FIG. 5.1. *The storage of local rows of a ParCSR matrix*



The implementation of the MM-ext interpolation is presented in Algorithm 1, where at step 4 we utilize the existing distributed MM multiplication routine. The MM-ext+i method is presented in Algorithm 2. Notice that an extra communication step is performed at step 3. The required entries in the i-th column of A^s_{FF} and the i-th entry of $\mathrm{diag}(\hat{A}^s_{FF} A^s_{FF})$ are computed at step 4, as in (3.11) and (3.12). A (probably) simpler alternative to implement this step is by computing the transpose of A^s_{FF} in advance, which, however, apparently requires more memory and communication.

Algorithm 1 The MM-ext interpolation

Input: coefficient matrix A, C/F splitting and SoC matrix S
Output: interpolation matrix P

1: Extract $\mathrm{diag}(D_{FF})$ in ParVec, and A^s_{FF} and A^s_{FC} in ParCSR based on S
2: Compute $\mathrm{diag}(D_\beta)$ in (3.5) and $\mathrm{diag}(D_\gamma)$ in (3.6), stored in ParVec
3: Diagonally scale A^s_{FF} and A^s_{FC} for $\tilde{A}^s_{FF}$ and $\tilde{A}^s_{FC}$ as in (3.4)
4: Compute $W = -\tilde{A}^s_{FF} \tilde{A}^s_{FC}$
5: Allocate $P = \binom{W}{I}$ and copy W into P

Algorithm 2 The MM-ext+i interpolation

Input: coefficient matrix A, C/F splitting and SoC matrix S
Output: interpolation matrix P

1: Extract $\mathrm{diag}(D_{FF})$ in ParVec, and A^s_{FF} and A^s_{FC} in ParCSR based on S
2: Compute $\mathrm{diag}(D_\beta)$ in (3.5) and $\mathrm{diag}(D_\gamma)$ in (3.6), stored in ParVec
3: Communicate for the external $\mathrm{diag}(D_\beta)$ and the external rows of A^s_{FF} in a CSR matrix X_{ext}, corresponding to the off-process column indices of the local A^s_{FF}.
4: Scale each row i of A^s_{FF} as in (3.10): for each $[A^s_{FF}]_{ij} \neq 0$, go to row j, search for $[A^s_{FF}]_{ji}$ in either the diagonal part of A^s_{FF} or in X_{ext}, depending on whether j is on- or off-process, and calculate $[\hat{A}^s_{FF}]_{ij} = \big([A^s_{FF}]_{ji} + [D_\beta]_{jj}\big)^{-1}[A^s_{FF}]_{ij}$. Accumulate $[\hat{A}^s_{FF}]_{ij}[A^s_{FF}]_{ji}$ in $[D_\theta]_{ii}$ per (3.12).
5: Diagonally scale $\hat{A}^s_{FF}$ for $\tilde{A}^s_{FF}$ as in (3.9)
6: Compute $W = -\tilde{A}^s_{FF} \tilde{A}^s_{FC}$
7: Allocate $P = \binom{W}{I}$ and copy W into P

The implementation of the MM-ext+e algorithm should be straightforward at this stage, so we omit the details for brevity.

6. Analyzing implementation of extended and MM-ext interpolation. In this section, we discuss the differences in the implementations between the original extended (ext) and the proposed MM-ext interpolations to get a better understanding of the differences in timings we observed in the Section 4. To simplify the discussion, we focus on the sequential version, where we already see a big difference in timings. We will estimate the number of floating point operations, specifically additions, multiplications and divisions for both versions. We will also count the number of if-statements or checks as we call them here, since the implementation of the original interpolation operators required many such branch statements, which are problematic when porting the code to GPUs and can also prevent compilers from performing

optimizations. We will use the following notations:

S	strength matrix contains strong connections in A, no diagonal elements
n	number of rows of A
m	average number of nonzeros per row in A without the diagonal element
$nnz(M)$	number of nonzeros of matrix M
σ	strength ratio defined as $nnz(S)/(nnz(A) - n)$
$\tilde{\sigma}$	strength ratio when some weak connections are included; $\sigma \leq \tilde{\sigma} \leq 1$
n_F	number of fine points.
n_C	number of coarse points and equivalent to $n - n_F$.
ρ	fine point ratio, defined as n_F/n, coarse point ratio is $1 - \rho$.

We will start by discussing storage requirements for both versions. Matrices are stored in the CSR format using an integer array 'I' of length $n + 1$, which marks the beginning for each row within the array 'J' for the column indices and the data array. We will assume double precision for the data and 32-bit integers for the integer arrays, thus multiplying the numbers for real arrays by two. Note that hypre also assumes that the first element in each row is the diagonal element, allowing for easy access to the diagonal.

Both versions require the original matrix A and the strength matrix S, which does not have a data array and omits the diagonal elements, an integer array of length n to mark whether a point is an F- or a C-point (`CFmarker`), an integer array of length n to map the original column indices to new column indices (`CFMap`) required for P or A^s_{FF} and A^s_{FC} and the final P-matrix. Extended interpolation additionally requires an integer array `PMarker` of length n used to mark strong fine connections or positions within P when generating the interpolation data structure and when populating it. We need about $(3m+1)n$ 32-bit units for A, $(\sigma m+1)n$ for S, and $3nnz(P)+n$ units for P. This leads to a total of $[(3 + \sigma)m + 6]n + 3nnz(P)$ units for the extended interpolation.

The MM-ext interpolation requires additionally the matrices A^s_{FF}, A^s_{FC}, W, an integer array of length ρn to mark positions when generating W and arrays D_β and D_γ of length ρn. Note that D_γ can be combined with D_{FF} to save storage. Note that in our implementation and analysis, the diagonal is included in A^s_{FF}, which is different from the definition for A^s_{FF} that we use in other sections of this paper. A^s_{FF} requires about $(3\sigma\rho m + 1)\rho n$ units, A^s_{FC} about $[3\sigma(1 - \rho)m + 1]\rho n$ units, and W about $3[nnz(P) - (1 - \rho)n] + \rho n$ units. Combining all these numbers leads to a total of $[(3+\sigma)m + 9\rho + 3\sigma\rho + 4]n + 6nnz(P)$ units for the MM-ext interpolation. Clearly, MM-ext requires more memory than extended interpolation.

The implementation of the extended interpolation in hypre consists of basically two phases: in Phase 1, the structure of P is determined and created as well as a map of current column indices to new coarse column indices for P, whereas in Phase 2, the interpolation weights are evaluated and the structure of P is populated.

Phase 1 loops through `CFmarker` to determine if a point i is coarse or fine, requiring a total of n checks over the course of the phase. If i is coarse, the counter for $nnz(P)$ is increased and `CFMap` is updated. If i is one of the ρn F-points, the process in Figure 6.1 is performed for matrix S and $c = \sigma m$. Phase 1 requires about $n + \rho n(1 + \sigma\rho m)(2 - \rho)\sigma m$ checks.

In Phase 2, the column indices and weights are inserted into P one row at a time. Phase 2 loops through `CFmarker` to check if i is fine or coarse. If i is a coarse point, the weight is set to one and the column index is inserted. If i is a fine point, two tasks

FIG. 6.1. *Typical code portion in implementation of extended interpolation.* `Mi` *denotes the i-the row of a matrix M and c the average number of coefficients per row. Note that this code portion performs approximately $(1+\rho c)(2-\rho)c$ checks.*

```
1 for j=Mi[i],...,Mi[i+1]   /* c coefficients */
2   if (Mj[j] is a C-point)   /* c checks */
3     if (PMarker[Mj[j]] < RowStart) /* (1-ρ)c checks */
4       ...
5     else /* ρc times */
6       for k=Mi[j],...,Mi[j+1]
7         if (Mj[k] is a C-point) /* c checks */
8           if (PMarker[Mj[k]] < RowStart) /* (1-ρ)c checks */
9             ...
```

are performed: first, inserting the column indices, second, computing and inserting the weights into the i-th row of P. While performing the first task, a marker array (`PMarker`) is generated to mark strong fine points, indicated as a negative number, or in case of a C-point to mark its position within the data and column indices arrays of P. This information is used to efficiently access locations, to initialize weights with zero, and to set column indices. The position information is also used to accumulate values when computing the weights during the second task. The first task uses the code in Figure 6.1 parsing S as in Phase 1. Therefore, this portion of the code across all rows requires the same amount of checks as in Phase 1, not counting checks needed for computing and inserting the weights, which is described next.

To perform the second task for row i, which is only performed when i is an F-point, it is necessary to parse A. While this portion of the code is structured similarly to Figure 6.1, it is slightly different. Note that $c=m$ here, since A is parsed skipping the diagonal element. The code does not include Line 3, but instead checks in Line 5 whether `Mj[j]` is a strong fine point requiring ρm checks. Before parsing A in Line 6, a check is made to determine the sign of the diagonal. Line 7 checks if this is a relevant C-point. There are $(1-\rho)\tilde{\sigma}m$ of them, since they can also include weakly connected C-points as we are parsing A. Line 8 checks the sign to avoid cancellation of row sums by large elements of the same sign as the diagonal element. The portion between Lines 5 and 9 performs $1+m+(1-\rho)\tilde{\sigma}m$ checks. A code portion is performed afterwards including a check that insures a sum is not equal zero and a portion similar to Lines 6 to 8. This leads to a total of $\rho n[m+\rho m+\sigma\rho m[2+2m+2(1-\rho)\tilde{\sigma}m]$ checks for the second task in Phase 2 across all F-points.

We now estimate the floating point operations. Referring to Figure 6.1, if `Mj[j]` is a coarse point, a data value is added to the interpolation weight using the location given by `PMarker`, requiring $(1-\rho)\tilde{\sigma}m$ adds. Note that, since we are parsing A and not S, this coefficient can also be a weak coarse connection. If `PMarker[Mj[j]]` is negative, `Mj[j]` is a strong fine connection. We then parse the j-th row of A to accumulate the row sum of the coarse connections k of point j. If k is coarse and of the opposite sign as the diagonal, a_{ij} is added to the diagonal (that is $(1-\rho)\tilde{\sigma}m$ adds). If the sum is larger than zero, which we assume to generally be the case for this analysis, the coefficient a_{ij} is divided by the sum and stored in a variable. Then the j-th row of A is parsed again, and the product of a_{jk} with the stored value is added to the appropriate position in P, requiring $(1-\rho)\tilde{\sigma}m$ adds and multiplications. If j is neither coarse nor a strong fine neighbor (about $[1-\tilde{\sigma}+\rho(\tilde{\sigma}-\sigma)]m$ coefficients), a_{ij} is treated as a weak connection and added to the diagonal. Finally every weight

in row P_i is divided by the value accumulated in the diagonal.

Since there are a total of ρn fine points, the generation of the weights requires about $\rho n[(1-\rho)\tilde{\sigma} m+2\sigma\rho m(1-\rho)\tilde{\sigma} m+(1-\sigma+\rho(\tilde{\sigma}-\sigma))m]$ additions, $\rho n\sigma\tilde{\sigma}\rho m(1-\rho)$ multiplications and $\rho n\sigma\rho m + nnz(P) - (1-\rho)n$ divisions. Combining the checks of Phase 1 and both parts of Phase 2 leads to $2n + 2\rho n(1+\sigma\rho m)(2-\rho)\sigma m + \rho n[m + \rho m + \sigma\rho m[2 + 2m + 2(1-\rho)\tilde{\sigma} m]$ checks.

To analyze the new interpolation we will refer to the steps outlined in Algorithm 1. In Step 1, A^s_{FF} and A^s_{FC} are generated. Generating mappings for both fine and coarse indices to new column indices requires n checks. Determining the number of coefficients for each matrix requires parsing S to determine the number of coefficients for each matrix.This requires $n + \sigma\rho mn$ checks. To populate the matrices requires parsing S again checking for coarse connections, while also parsing A. Now additional checks are required to skip weak connections in A leading to a total of $n+\rho mn$ checks. In Step 2, adding weak connections to the diagonal requires n checks, and merging W into P in Step 5 requires another n checks. Step 4, the multiplication of $\tilde{A}^s_{FF}$, which has ρn rows and on average $\rho\sigma m + 1$ nonzeros per row, with $\tilde{A}^s_{FC}$, which has on average $\sigma(1-\rho)m$ nonzeros per row, consists of 2 phases: the generation of the structure of W followed by its population. It uses a `PMarker` array similarly as described for the extended interpolation to mark positions and avoid over-counting of indices in W, requiring a total of $2\sigma\rho(1-\rho)(\sigma\rho m+1)mn$ checks. To avoid dividing by zeros requires an additional $2\rho n$ checks. Combining this leads to a total of $5n + 2\rho n + [1 + \sigma + 2\sigma(1-\rho)(\sigma\rho m + 1)]\rho mn$ checks.

Now we estimate the number of floating point operations. In Step 2, generating D_β requires $\sigma\rho(1-\rho)mn$ additions, and computing $D_\gamma + D_{FF}$ by determining the row sums of A and subtracting the row sums of A^s_{FF} and D_β requires $\rho n(2 + m + \sigma\rho m)$ additions (or subtractions). In Step 3, generating $\tilde{A}^s_{FF}$ requires ρn divisions and $\sigma\rho^2 mn$ multiplications, and computing $\tilde{A}^s_{FC}$ requires another ρn divisions and $\sigma\rho(1-\rho)mn$ multiplications. Finally, we estimate Step 4 to take about $\rho\sigma(\sigma\rho m+1)(1-\rho)mn$ multiplications and additions. Adding these numbers leads to $2\rho n + (2+\sigma)\rho mn + \sigma^2\rho^2(1-\rho)m^2 n$ additions, $\sigma\rho mn[1+(1-\rho)(\sigma\rho m+1)]$ multiplications and $2\rho n$ divisions.

The results for both interpolation algorithms are combined in Table 6.1. We generally arranged the larger components first and lined them up for better comparisons. As such we consider portions containing m^2 to be generally more significant, specifically for larger stencils. Comparing checks between both algorithms, we see that the number of checks are significantly larger for the extended interpolation. This is not surprising, since the generation of matrices containing only strong coefficients eliminates many checks. We also see that the number of additions is larger for the extended interpolation. Since it is possible for the MM-ext interpolation to generate the row sums of only strong coefficients ahead of time and store them, many additions can be avoided. Since in the extended interpolation weak coefficients can be included in the sums it is not possible to generate them ahead of time. The number of multiplications appears fairly similar with some additional multiplications since divisions are avoided in MM-ext by storing inverses. This approach leads to more divisions for the extended interpolation.

We measured the individual times for performing the two interpolation operators on the first two levels of the 7pt and the 27pt problem on a $80 \times 80 \times 80$ grid using optimization level -O2. We present the parameters and times for the individual routines, which are measured in seconds in Table 6.2. The upper portion of Table 6.3 contains the estimates we received inserting the parameters into the formulae in Table

Table 6.1
Estimates of number of checks, floating point operations and memory in extended and MM-ext interpolation

	extended interpolation	MM-ext interpolation
Checks	$2\rho^2m^2n\sigma[\sigma(2-\rho)+\tilde{\sigma}(1-\rho)+1]+\rho mn(1+4\sigma)+2n$	$2\rho^2m^2n\sigma^2(1-\rho)+\rho mn[1+3\sigma-2\sigma\rho)]+(5+2\rho)n$
Adds	$2\sigma\tilde{\sigma}(1-\rho)\rho^2m^2n+\rho mn(1-\sigma\rho)$	$\sigma^2(1-\rho)\rho^2m^2n+\rho mn(1-\sigma\rho+2\sigma)+2\rho n$
Mults	$\sigma\tilde{\sigma}(1-\rho)\rho^2m^2n$	$\sigma^2(1-\rho)\rho^2m^2n+\sigma\rho mn(2-\rho)$
Divs	$\sigma\rho^2mn+nnz(P)-(1-\rho)n$	$2\rho n$
Memory	$[(3+\sigma)m+6]n+3nnz(P)$	$[(3+\sigma)m+4+9\rho+3\sigma\rho]n+6nnz(P)$

6.1. We see that the extended interpolation requires significantly more checks, about four times as many for the '7pt, lvl 1' problem, and seventeen times as many for the '27pt, lvl 1' problem, but approximately similar total amounts of floating point operations. The MM-ext interpolation operator for the latter problem can be generated about three times as fast as the extended interpolation operator. Note that while the extended interpolation has more divisions, replacing those with multiplications does not change the times significantly. It is harder to compare the operators on level 2, since we do not have the exact value for $\tilde{\sigma}$, we just present an interval here for the lowest (no weak connections included) and highest (all weak connections included) estimated value. The actual value is expected to be somewhere in between.

To get additional understanding of the differences, we used the performance tool `perf` [2] to measure individual events for the two routines. We specifically list numbers for instructions, cycles, cache-references, cache-misses, page faults, memory stores and branch instructions in Table 6.3. Since several runs of `perf` led to variations of about 1-3 percent for the first 3 matrices and up to 5 percent for the smallest matrix, we list here the average numbers of three runs. While these numbers are not accurate, they provide additional information. Generally, the MM-ext interpolation has significantly more page faults and cache misses than extended interpolation, however lower (often much lower) numbers of instructions, cycles and branch instructions. The latter explains its better times. Comparing the number of instructions with the times, we see similar proportions.

Table 6.2
Parameters for 2 levels of the 7pt and 27pt problems

	n	m	ρ	σ	$\tilde{\sigma}$	$nnz(P)$	time ext	time MM-ext
7pt, lvl 1	512,000	5.9	0.69	1.0	1.0	$5.0n$	0.15	0.13
7pt, lvl 2	159,443	26.9	0.84	0.42	1.0 (0.42)	$8.7n$	0.28	0.10
27pt, lvl 1	512,000	25.3	0.92	1.0	1.0	$9.0n$	1.86	0.57
27pt, lvl 2	41,999	56.7	0.88	0.34	1.0 (0.34)	$14.6n$	0.20	0.05

Table 6.3

Estimates for checks, and floating point operations using Table 6.1 and performance events using performance tool `perf` *for 2 levels of the 7pt and 27pt problems*

	7pt, lvl 1		7pt, lvl 2		27pt, lvl 1		27pt, lvl 2	
	ext	MM-ext	ext	MM-ext	ext	MM-ext	ext	MM-ext
Checks	$109.2n$	$27.3n$	$769.0n$ ($729.2n$)	$110.4n$	$2458.8n$	$143.8n$	$2660.5n$ ($2526.4n$)	$280.8n$
Adds	$11.5n$	$15.9n$	$83.2n$ ($43.4n$)	$49.7n$	$88.5n$	$93.6n$	$238.1n$ ($104.0n$)	$105.2n$
Mults	$5.1n$	$10.5n$	$34.3n$ ($14.4n$)	$25.4n$	$43.3n$	$68.5n$	$101.6n$ ($34.5n$)	$53.5n$
Divs	$7.5n$	$1.4n$	$16.5n$	$1.7n$	$30.3n$	$1.8n$	$29.4n$	$1.8n$
Memory	$44.6n$	$65.9n$	$124.1n$	$156.8n$	$134.2n$	$170.2n$	$239.2n$	$289.8n$
instruct.	$942n$	$882n$	$5613n$	$2111n$	$18{,}228n$	$4{,}544n$	$19{,}080n$	$4{,}403n$
cycles	$958n$	$637n$	$6{,}996n$	$1{,}976n$	$14{,}689n$	$3{,}010n$	$19{,}008n$	$3{,}510n$
cache ref	$9.8n$	$14.0n$	$32.7n$	$29.7n$	$70.1n$	$42.5n$	$118.1n$	$48.8n$
cache mis	$1.81n$	$5.61n$	$5.37n$	$11.07n$	$8.21n$	$24.09n$	$4.86n$	$6.45n$
pg. faults	804	5633	836	4814	2694	5907	1559	2600
mem-stores	$58.0n$	$87.7n$	$129.2n$	$154.3n$	$225.3n$	$266.8n$	$254.4n$	$255.9n$
branch inst	$207.9n$	$147.2n$	$1397.7n$	$372.8n$	$4{,}784.9n$	$835.2n$	$4900.8n$	$855.1n$

7. Two-stage MM-interpolation. The extended interpolation operators can have larger memory requirements than desired even when using truncation. Therefore, efforts have been made to coarsen even more aggressively to lower the overall operator complexities of the AMG hierarchy. Aggressive coarsening typically needs to be combined with longer range interpolation algorithms, such as multi-pass interpolation [31], to deal with the fact that C-points are now often more than distance-two apart. However, multi-pass interpolation is based on direct interpolation, which generally leads to worse convergence than distance-two interpolation. In [34], a two-stage interpolation operator was introduced that is based on distance-two interpolation and has shown to lead to better convergence than multi-pass interpolation. However, one drawback of two-stage interpolation is its expensive setup. Since the class of MM-interpolation operators has shown significantly improved setup times, we want to investigate here whether this approach has the same effect on a MM formulation of two-stage interpolation operators.

The basic idea of aggressive coarsening is to perform a second C/F splitting of the C-points obtained by the first coarsening phase. The final C/F splitting can be represented by the following three sets:

$$\begin{aligned} F^1 &: \text{ the original } F\text{-points, i.e., } F^1 \equiv F, \\ F^2 &: \text{ the original } C\text{-points that become } F\text{-points at the 2nd coarsening,} \\ C^2 &: \text{ the final } C\text{-points after the second coarsening,} \end{aligned} \tag{7.1}$$

where the final F-set is $F^1 \cup F^2$ and the final C-set equals C^2. As the name suggests, two-stage interpolation consists of two stages. During the first stage, a distance-two interpolation operator, such as extended or the extended+i interpolation, is applied using the original F- and C-sets, i.e, F^1 and $F^2 \cup C^2$. This generates the first-stage interpolation matrix

$$P_1 = \begin{pmatrix} W_1 & W_0 \\ I & \\ & I \end{pmatrix}, \tag{7.2}$$

where $W_1 \in \mathbb{R}^{n_f \times |F^2|}$ and $W_0 \in \mathbb{R}^{n_f \times |C^2|}$ contain the interpolation weights from F^1 to F^2 and C^2 respectively. During the second stage, the weight matrix $W_2 \in \mathbb{R}^{|F^2| \times |C^2|}$ is computed for the now fine points in F^2 with the interpolatory set C^2, leading to the second-stage interpolation matrix

$$P_2 = \begin{pmatrix} W_2 \\ I \end{pmatrix}. \tag{7.3}$$

The final interpolation operator can be generated by multiplying the two matrices:

$$P = P_1 P_2 = \begin{pmatrix} W_0 + W_1 W_2 \\ W_2 \\ I \end{pmatrix}. \tag{7.4}$$

7.1. Two-stage MM-ext interpolation. According to the C/F splitting (7.1) in the two stages, A is reordered and partitioned into the 3×3 block form:

$$\left(\begin{array}{c|cc} A_{F^1F^1} & A_{F^1F^2} & A_{F^1C^2} \\ \hline A_{F^2F^1} & A_{F^2F^2} & A_{F^2C^2} \\ A_{C^2F^1} & A_{C^2F^2} & A_{C^2C^2} \end{array}\right), \tag{7.5}$$

and once again split into its diagonal, strong and weak parts, i.e., $A = D + A^s + A^w$. During the first stage (7.2), we apply the MM-ext interpolation using the first block row, which leads to the following matrix

$$\begin{aligned} [W_1, W_0] &= -(D_{F^1F^1} + D_\gamma)^{-1}(A^s_{F^1F^1} + D_\beta)[D_\beta^{-1} A^s_{F^1F^2}, D_\beta^{-1} A^s_{FC^2}] && (7.6) \\ &\equiv -\tilde{A}^s_{F^1F^1}[\tilde{A}^s_{F^1F^2}, \tilde{A}^s_{F^1C^2}], && (7.7) \end{aligned}$$

where D_γ and D_β are the same as in (3.5) and (3.6) with $A_{FC} = [A_{F^1F^2}, A_{F^1C^2}]$.

For the second stage, it is easy to see that we need to consider the matrix blocks,

$$\left(\begin{array}{cc|c} * & * & A_{F^1C^2} \\ A_{F^2F^1} & A_{F^2F^2} & A_{F^2C^2} \end{array}\right). \tag{7.8}$$

We use the following definitions to express (7.3),

$$\begin{aligned} D_{\beta_k} &= \operatorname{diag}(A^s_{F^kC^2} e_{c_2}), \quad k = 1, 2 && (7.9) \\ D_{\gamma_2} &= \operatorname{diag}(A^w_{F^2F^1} e_f + A^w_{F^2F^2} e_{f_2} + A^w_{F^2C^2} e_{c_2}), && (7.10) \end{aligned}$$

where $e_{f_2} \in \mathbb{R}^{|F^2|}$ and $e_{c_2} \in \mathbb{R}^{|C^2|}$ denote vectors of all ones, and therefore

$$\begin{aligned} W_2 &= -(D_{F^2F^2} + D_{\gamma_2})^{-1}\big(A^s_{F^2F^1} D_{\beta_1}^{-1} A^s_{F^1C^2} + (A^s_{F^2F^2} + D_{\beta_2}) D_{\beta_2}^{-1} A^s_{F^2C^2}\big) && (7.11) \\ &\equiv -\tilde{A}^s_{F^2F^1} \tilde{A}^s_{F^1C^2} - \tilde{A}^s_{F^2F^2} \tilde{A}^s_{F^2C^2}, && (7.12) \end{aligned}$$

with

$$\begin{aligned} \tilde{A}^s_{F^2F^1} &= (D_{F^2F^2} + D_{\gamma_2})^{-1} A^s_{F^2F^1}, && (7.13) \\ \tilde{A}^s_{F^2F^2} &= (D_{F^2F^2} + D_{\gamma_2})^{-1}(A^s_{F^2F^2} + D_{\beta_2}), && (7.14) \\ \tilde{A}^s_{F^kC^2} &= D_{\beta_k}^{-1} A^s_{F^kC^2}, \quad k = 1, 2. && (7.15) \end{aligned}$$

Notice that it is possible that D_{β_1} and D_{β_2} are singular, so that their inverses are not defined. The fix mentioned at the end of Section 3.1 can also be applied, i.e. the

rows of $A^s_{F^1C^2}$ and $A^s_{F^2C^2}$ corresponding to the zero diagonal entries of D_{β_1} and D_{β_2} are scaled by 0. Moreover, the associated elements of $A^s_{F^2F^1}$ and $A^s_{F^2F^2}$ should be added to D_{γ_2} to correctly interpolate constants. Note that W_2 can be computed as the multiplication of the matrix containing $\tilde{A}_{F^2F^k}$ with the matrix containing $\tilde{A}_{F^kC^2}$ for $k = 1, 2$ using the stucture in (7.8). The final interpolation matrix is obtained by substituting W_0, W_1 and W_2 into (7.4).

7.2. Two-stage MM-ext+i and MM-ext+e interpolation. The two-stage MM-ext+i and MM-ext+e interpolation operators can be derived in a similar way. For the sake of completeness, we give below their expressions. The first-stage MM-ext+i interpolation operator is

$$[W_1, W_0] = -(D_{F^1F^1} + D_\gamma + D_\theta)^{-1}(\hat{A}^s_{F^1F^1} + I)[A^s_{F^1F^2}, A^s_{F^1C^2}], \tag{7.16}$$

where D_θ and $\hat{A}^s_{F^1F^1}$ are the same as in (3.10) and (3.12). The second-stage interpolation is given by

$$W_2 = -(D_{F^2F^2} + D_{\gamma_2} + D_{\theta_2})^{-1}\big(\hat{A}^s_{F^2F^1}A^s_{F^1C^2} + (\hat{A}^s_{F^2F^2} + I)A^s_{F^2C^2}\big), \tag{7.17}$$

where, for $k = 1, 2$, the matrix $\hat{A}^s_{F^2F^k}$ is computed by scaling $A^s_{F^2F^k}$ with

$$D_{\beta_k^{(i)}} = \mathrm{diag}(A^s_{F^kF^2}e_i + A^s_{F^kC^2}e_{c_2}), \tag{7.18}$$

in the same way as (3.10), and $D_{\theta_2} = \mathrm{diag}(d_1 + d_2)$ with $d_k = \mathrm{diag}(\hat{A}^s_{F^2F^k}A^s_{F^kF^2})$.

Finally, we describe the two-stage MM-ext+e. For the first stage, we have

$$[W_1, W_0] = -(D_{F^1F^1} + D_\gamma + D_\tau)^{-1}(A^s_{F^1F^1} + D_\lambda)D_\lambda^{-1}[A^s_{F^1F^2}, A^s_{F^1C^2}], \tag{7.19}$$

with D_λ and D_τ the same as defined in (3.16) and (3.17) respectively. For the second stage, we define the diagonal matrix D_{μ_k}, for $k = 1, 2$, as follows

$$[D_{\mu_k}]_{ii} = s_i^{-1}\textstyle\sum_{l=1}^2\sum_j\,[A^s_{F^kF^l}]_{ij}, \quad i = 1, 2, \ldots, |F^k|, \tag{7.20}$$

where s_i denotes combined number of nonzeros of the i-th rows of $A^s_{F^kF^1}$ and $A^s_{F^kF^2}$. Thus, the second-stage interpolation weight matrix can be written as

$$W_2 = -(D_{F^2F^2} + D_{\gamma_2} + D_{\tau_2})^{-1}\big(A^s_{F^2F^1}D_{\lambda_1}^{-1}A^s_{F^1C^2} + (A^s_{F^2F^2} + D_{\lambda_2})D_{\lambda_2}^{-1}A^s_{F^2C^2}\big),$$

where, for $k = 1, 2$, $D_{\lambda_k} = D_{\beta_k} + D_{\mu_k}$, and D_{τ_2} is defined similarly as D_τ by

$$[D_{\tau_2}]_{ii} = \textstyle\sum_{k=1}^2\sum_j\,[A_{F^2F^k}]_{ij}[D_{\mu_k}]_{jj}[D_{\lambda_k}]^{-1}_{jj}. \tag{7.21}$$

Note that, differently than two-stage MM-ext, two-stage MM-ext+i requires $A_{F^1F^2}$ and MM-ext+e requires the use of $A_{F^1F^1}$ and $A_{F^1F^2}$ in their second stages, leading to additional computation cost. While two-stage MM-ext+i needs to extract columns from $A_{F^1F^2}$ requiring the expensive communication of rows of matrices from neighboring processes, two-stage MM-ext+e can generate D_{μ_k} locally.

8. Numerical Results. The experiments were conducted on a Linux cluster at Lawrence Livermore National Laboratory with 62 nodes, each of which is equipped with a 40-core IBM Power9 CPU. The code was written in C, included in hypre's BoomerAMG and compiled by IBM `mpixlc` compiler with Spectrum MPI. We compared the new MM-interpolation operators with extended (denoted by ext) and extended+i interpolation (denoted by ext+i) for various structured and unstructured

diffusion problems, using AMG as a preconditioner for CG. We also tested the use of one level of aggressive coarsening in AMG with the 5 different settings that are listed and described in Table 8.1. The method 2sMei is only partially implemented, i.e. the first pass uses MM-ext+i but the second stage uses ext+i, i.e. the routine that is also used for 2sei, since we did not expect to gain anything using the new formulation due to its much larger communication requirements compared with 2sMe and 2sMee.

Table 8.1
Experiment settings for the interpolation algorithms with 1-level aggressive coarsening

mp	multi-pass interpolation on the 1st level, ext+i on the remaining levels
2sei	2-stage ext+i on the 1st level, MM-ext+i on the remaining levels
2sMe	2-stage MM-ext on the 1st level, MM-ext on the remaining levels
2sMei	*partial* 2-stage MM-ext+i on the 1st level, MM-ext+i on the remainings
2sMee	2-stage MM-ext+e on the 1st level, MM-ext+e on the remaining levels

8.1. Structured diffusion problems. We first present the results for a variety of structured model problems that include

7pt/27pt	3D Poisson problems on a cube with 7- and 27-point stencils,
sysLap	System of Poisson equations with 3 variables per mesh point,
Jumps	3D 7-point Poisson problem with jumps in coefficients of up to 10^6,
Aniso3D	3-D Poisson problem with anistropy of 0.001 in the y-direction,
Aniso2D45°	2-D Poisson problem with anisotropy of 0.001 and 45° rotations,
Aniso2D60°	2-D Poisson problem with anisotropy of 0.001 and 60° rotations.

All runs were executed on 16 compute nodes with a total of 640 cores. For a set of larger problems, we used 8 million degrees of freedom per node, leading to the global problem size of 128 million for the 3-D scalar PDE problems, and 384 million for the problem from the system of PDE. For a set of smaller problems, we used 1 million degree of freedoms per node, and 3 million for sysLap. The Aniso2D problem has 8 million unknowns per node, for which we did not use aggressive coarsening, since it significantly deteriorates convergence. We used PMIS coarsening and the weighted Jacobi smoother with the following weights ω for different problems.

	7pt/27pt	sysLap	Jumps	Jumps w/ (2sei, 2sMe)	Aniso 3D	Aniso 2D45°	Aniso 2D60°
ω	0.85	0.5	0.85	0.8	0.85	0.85	0.7

The results for the structured diffusion problems are reported in Tables 8.2, 8.3 and 8.4, where AMG setup, solve and total times and the number of iterations are presented. The best results are highlighted in boldface. We can clearly see that for most cases the AMG setup time is considerably shortened when using MM-interpolation operators compared with extended and extended+i interpolation. The required number of iterations is comparable or even reduced in some cases. Therefore, the total time is generally improved. Among the MM-interpolation methods, the setup time of MM-ext and MM-ext+e is typically shorter than that of MM-ext+i, which is consistent with the discussion in Section 3, since they do not require the communication for retrieving the external matrix rows. The results when using aggressive coarsening are similar, i.e., the total times for the new interpolation operators are typically better than 2sei. While generally AMG setup with multi-pass interpolation is still the

TABLE 8.2

Results for various interpolation operators for the set of larger structured problems. All timings are in seconds. The best results are highlighted in boldface.

		no aggressive coarsening					1 level aggressive coarsening				
		ext	ext+i	MM-ext	MM-ext+i	MM-ext+e	mp	2sei	2sMe	2sMei	2sMee
7pt	setup	1.45	1.55	**1.01**	1.06	**1.01**	**0.66**	0.99	0.77	0.80	0.75
	solve	1.00	1.18	1.12	1.00	**0.93**	1.22	0.87	0.97	**0.85**	0.86
	total	2.45	2.73	2.13	2.06	**1.94**	1.88	1.86	1.74	1.65	**1.61**
	iter	18	22	19	18	17	30	22	25	22	22
27pt	setup	3.09	3.25	1.54	1.67	**1.49**	**0.99**	3.06	1.45	1.68	1.49
	solve	1.35	1.53	1.23	**1.22**	1.24	1.69	**1.35**	1.39	1.41	1.39
	total	4.44	4.78	2.77	2.89	**2.73**	**2.68**	4.41	2.84	3.09	2.88
	iter	18	21	16	16	16	26	21	21	21	21
sysLap	setup	8.18	8.45	5.99	6.06	**5.85**	**3.34**	4.56	3.66	3.64	3.51
	solve	12.19	13.72	12.65	11.38	**11.37**	11.21	9.03	10.25	**8.85**	8.87
	total	20.37	22.17	18.64	17.44	**17.36**	14.55	13.59	13.91	12.49	**12.38**
	iter	39	45	40	37	37	64	48	52	47	47
Jumps	setup	1.53	1.77	1.11	1.18	**1.06**	**0.67**	1.19	0.85	0.87	0.85
	solve	2.32	2.04	2.24	**1.66**	1.72	2.18	1.77	2.10	**1.58**	1.60
	total	3.85	3.81	3.35	2.84	**2.78**	2.85	2.96	2.95	**2.45**	**2.45**
	iter	39	35	38	29	30	56	45	52	42	43

fastest, however using the MM-strategy has brought two-stage interpolation setup times within reach and made them a very good alternative option with better convergence and often better total times. Finally, notice that the convergence of MM-ext and 2sMe is generally worse compared with the interpolation algorithms that include connections to i or their estimates, particularly, for harder problems with jumps and rotated anisotropies.

8.2. Unstructured diffusion problems. We next consider two diffusion problems with 3-D unstructured FEM meshes, depicted in Figure 8.1, which were discretized with finite element package MFEM [1]. The sphere-in-sphere problem uses an unstructured tetrahedral mesh on a spherical domain that has a spherical hollow in the middle, see Figure 8.1a for an illustration. The larger problem has $136,501,953$ degrees of freedom, while the smaller problem has $17,143,777$. The second unstructured problem is posed on a crooked pipe, see [20] for a detailed description of the problem, which uses an unstructured hexahedral mesh as shown in Figure 8.1b. This problem is very challenging due to the dense layer of highly stretched elements in the neighborhood of the material interface, which are added to resolve the diffusion layer (zoom shown in 8.1c). The larger problem has $59,604,993$ degrees of freedom, and the smaller problem has $7,544,257$. This problem requires a better smoother than Jacobi, so symmetric l_1-Gauss-Seidel was used.

The results for the unstructured problems are reported in Table 8.5. The operator complexity (o.c.) is roughly the same among all the interpolation methods, so the memory cost of the AMG hierarchy is similar. With aggressive coarsening, the operator complexity of "mp", which is based on the simple direct interpolation, is lower than the complexities of the two-stage approaches. Similar as the structured problems, the MM-interpolation algorithms show improvements in both setup and solve

TABLE 8.3

Results for various interpolation operators for the set of smaller structured problems. All timings are in seconds. The best results are highlighted in boldface.

		no aggressive coarsening					1 level aggressive coarsening				
		ext	ext+i	MM-ext	MM-ext+i	MM-ext+e	mp	2sei	2sMe	2sMei	2sMee
7pt	setup	0.29	0.30	**0.18**	**0.18**	**0.18**	**0.13**	0.28	0.15	0.15	0.15
	solve	**0.13**	**0.13**	**0.13**	**0.13**	**0.13**	0.11	0.10	0.14	0.12	**0.09**
	total	0.42	0.42	**0.31**	**0.31**	0.32	**0.24**	0.38	0.29	0.27	**0.24**
	iter	16	17	16	15	15	26	20	23	20	20
27pt	setup	0.48	0.52	**0.25**	0.30	0.27	**0.19**	0.48	0.27	0.31	0.29
	solve	0.22	0.24	0.18	0.18	**0.17**	**0.24**	0.30	0.28	0.26	0.23
	total	0.70	0.76	**0.43**	0.48	0.44	**0.43**	0.78	0.56	0.57	0.52
	iter	15	16	14	14	14	24	21	21	21	21
sysLap	setup	1.62	1.62	0.94	**0.92**	**0.92**	0.90	1.20	0.66	0.70	**0.64**
	solve	1.86	1.79	1.93	1.75	**1.69**	2.06	1.63	1.94	**1.67**	1.71
	total	3.47	3.41	2.87	2.67	**2.61**	2.96	2.83	2.60	2.37	**2.35**
	iter	35	35	35	32	32	57	43	47	43	43
Jumps	setup	0.33	0.30	0.18	0.19	**0.18**	**0.13**	0.25	0.14	0.15	0.15
	solve	0.24	**0.23**	0.31	0.24	0.24	0.28	**0.23**	0.28	**0.23**	**0.23**
	total	0.57	0.53	0.49	0.43	**0.42**	0.41	0.49	0.42	**0.38**	**0.38**
	iter	36	33	39	32	33	51	38	47	38	39

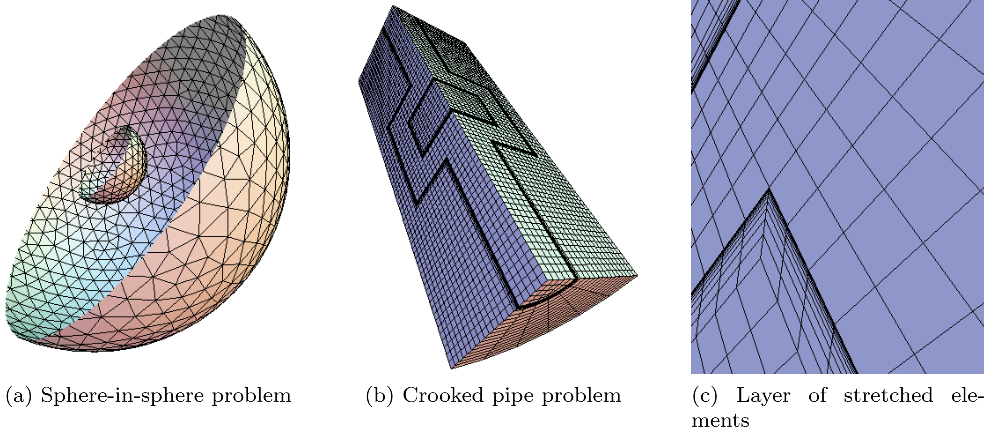
TABLE 8.4

Results for the 3-D anisotropic and the 2-D rotated anisotropic problems. All timings are in seconds. Aggressive coarsening was not used for the 2-D problems. The best results are highlighted in boldface.

		no aggressive coarsening					1 level aggressive coarsening				
		ext	ext+i	MM-ext	MM-ext+i	MM-ext+e	mp	2sei	2sMe	2sMei	2sMee
Aniso3D	setup	1.25	1.28	**0.96**	**0.96**	**0.96**	**0.60**	0.90	0.69	0.70	0.71
	solve	1.01	**0.91**	1.02	0.93	0.92	1.19	**1.05**	1.23	1.18	1.17
	total	2.26	2.19	1.98	1.99	**1.88**	**1.79**	1.95	1.92	1.88	1.88
	iter	16	14	16	14	14	29	25	27	26	26
Aniso2D45°	setup	0.40	0.40	0.40	**0.39**	**0.39**	—	—	—	—	—
	solve	1.01	0.69	1.05	0.77	**0.73**	—	—	—	—	—
	total	1.41	**1.09**	1.45	1.16	1.12	—	—	—	—	—
	iter	22	15	23	16	16	—	—	—	—	—
Aniso2D60°	setup	0.70	0.71	**0.58**	0.65	0.65	—	—	—	—	—
	solve	3.29	2.76	4.17	2.64	**2.61**	—	—	—	—	—
	total	3.99	3.47	4.75	3.29	**3.26**	—	—	—	—	—
	iter	71	57	92	54	55	—	—	—	—	—

times. The MM-ext+i and MM-ext+e algorithms perform about equally well and outperform the other methods without aggressive coarsening. When using aggressive coarsening, 2sMei and 2sMee generally outperform the other two-stage methods.

9. Conclusion. We introduced a new class of interpolation operators for algebraic multigrid that are based on sparse matrix-matrix multiplications. We inves-



(a) Sphere-in-sphere problem (b) Crooked pipe problem (c) Layer of stretched elements

Fig. 8.1. *The mesh for the sphere-in-sphere problem, and the crooked pipe problem where a dense layer of highly stretched elements has been added to the neighborhood of the material interface.*

Table 8.5
Results for the unstructured diffusion problems. The best results are highlighted in boldface.

		no aggressive coarsening					1 level aggressive coarsening				
		ext	ext+i	MM-ext	MM-ext+i	MM-ext+e	mp	2sei	2sMe	2sMei	2sMee
Sphere small	setup	0.52	0.47	**0.29**	**0.29**	0.30	**0.17**	0.39	0.20	0.20	0.19
	solve	**0.14**	0.16	0.16	**0.14**	**0.14**	0.15	0.13	0.15	**0.13**	0.14
	total	0.66	0.63	0.45	**0.43**	0.44	**0.32**	0.52	0.35	0.33	0.33
	iter	13	14	14	12	12	22	18	20	18	19
	o.c.	2.11	2.07	2.15	2.09	2.10	1.12	1.17	1.20	1.17	1.17
Sphere large	setup	3.16	3.23	2.31	2.30	**2.26**	**0.92**	1.74	1.13	1.21	1.12
	solve	1.25	1.22	1.18	**1.08**	1.10	1.25	1.05	1.21	1.05	**1.04**
	total	4.41	4.45	3.49	3.38	**3.36**	2.17	2.79	2.34	2.26	**2.16**
	iter	16	16	15	14	14	25	20	23	20	20
	o.c.	2.14	2.11	2.19	2.13	2.14	1.13	1.18	1.22	1.18	1.18
Pipe small	setup	0.63	0.53	0.39	**0.37**	0.39	**0.24**	0.38	0.29	0.30	0.28
	solve	2.24	2.29	2.35	**1.95**	1.97	1.68	1.90	1.78	1.62	**1.51**
	total	2.87	2.82	2.74	**2.32**	2.36	1.92	2.28	2.07	1.92	**1.79**
	iter	144	150	154	127	128	243	251	237	234	222
	o.c.	1.90	1.88	1.88	1.90	1.91	1.08	1.11	1.12	1.11	1.11
Pipe large	setup	2.67	2.71	**1.56**	1.71	1.63	**0.67**	1.73	0.91	1.08	0.94
	solve	16.32	15.46	16.73	**14.21**	14.46	14.64	14.98	15.36	14.16	**13.94**
	total	18.99	18.17	18.29	**15.92**	16.09	15.31	16.71	16.27	15.24	**14.88**
	iter	183	177	193	160	163	315	306	308	292	283
	o.c.	1.90	1.89	1.87	1.89	1.89	1.08	1.11	1.13	1.11	1.12

tigated their convergence for various problems and analysed their performance in comparison to current similar interpolation operators. We found that the new interpolation operators in general showed similar convergence as their counterparts. While the original goal was to find new formulations for AMG interpolatiom operators that

are more suitable for GPUs than the current ones, it turned out that the new methods also show better performance on CPUs. Analysing their implementation showed that the new methods need more memory, but significantly less if statements and branching operations, and have overall better fine-grain parallelism. We also investigated versions of the new methods in the context of two-stage interpolation operators for more aggressive coarsening schemes and found that they overcame previous large setup times making two-stage interpolation more competitive with multi-pass interpolation. While hypre v.2.1.9 includes GPU-implementations of MM-ext and MM-ext+i interpolation, there are future plans to add additional GPU-enabled MM-interpolation operators in hypre and to evaluate their performance.

Acknowledgments. We would like to thank Chak Shing Lee for the help with setting up the problems in Section 8.2, and acknowledge the fruitful discussions with Rob Falgout which prompted the development of the MM-ext+e interpolation in Section 3.3.

REFERENCES

[1] *MFEM: Modular finite element methods library.* `mfem.org`.

[2] *perf: Linux profiling with performance counters.* `https://perf.wiki.kernel.org/index.php/Main_Page`.

[3] A. Baker, R. Falgout, T. Kolev, and U. M. Yang, *Multigrid smoothers for ultraparallel computing*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2864–2887.

[4] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, *Scaling Hypre's Multigrid Solvers to 100,000 Cores*, Springer London, London, 2012, pp. 261–279.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, ACM Trans. Graph., 22 (2003), pp. 917–924.

[6] A. Brandt, S. McCormick, and J. Ruge, in Sparsity and Its Applications, Evans, ed., Cambridge University Press, Cambridge, 1984, ch. Algebraic multigrid (AMG) for sparse matrix equations.

[7] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*, Society for Industrial and Applied Mathematics, second ed., 2000.

[8] L. Buatois, G. Caumon, and B. Lévy, *Concurrent number cruncher: a GPU implementation of a general sparse linear solver*, International Journal of Parallel, Emergent and Distributed Systems, 24 (2009), pp. 205–223.

[9] M. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, *Solving lattice QCD systems of equations using mixed precision solvers on GPUs*, Computer Physics Communications, 181 (2010), pp. 1517 – 1528.

[10] S. Dalton, L. Olson, and N. Bell, *Optimizing sparse matrix—matrix multiplication for the gpu*, ACM Trans. Math. Softw., 41 (2015).

[11] H. De Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, *Distance-two interpolation for parallel algebraic multigrid*, Numerical Linear Algebra with Applications, 15 (2008), pp. 115–139.

[12] H. De Sterck, U. M. Yang, and J. J. Heys, *Reducing complexity in parallel algebraic multigrid preconditioners*, SIAM Journal on Matrix Analysis and Applications, 27 (2006), pp. 1019–1039.

[13] M. Deveci, C. Trott, and S. Rajamanickam, *Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures*, Parallel Computing, 78 (2018), pp. 33 – 46.

[14] R. D. Falgout, *An introduction to algebraic multigrid*, Computing in Science Engineering, 8 (2006), pp. 24–33.

[15] R. D. Falgout and U. M. Yang, *hypre: A library of high performance preconditioners*, in Computational Science — ICCS 2002, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, eds., Berlin, Heidelberg, 2002, Springer Berlin Heidelberg, pp. 632–641.

[16] R. Gandham, K. Esler, and Y. Zhang, *A GPU accelerated aggregation algebraic multigrid method*, Computers & Mathematics with Applications, 68 (2014), pp. 1151 – 1160.

[17] F. Gremse, K. Küpper, and U. Naumann, *Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures*, SIAM Journal on Scientific Computing,

40 (2018), pp. C429–C449.

[18] V. E. Henson and U. M. Yang, *Boomeramg: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155 – 177. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss.

[19] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[20] T. V. Kolev and P. S. Vassilevski, *Parallel auxiliary space amg solver for $H(div)$ problems*, SIAM Journal on Scientific Computing, 34 (2012), pp. A3079–A3098.

[21] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, The Journal of Supercomputing, 63 (2013), pp. 443–466.

[22] R. Li, Y. Xi, L. Erlandson, and Y. Saad, *The eigenvalues slicing library (evsl): Algorithms, implementation, and software*, SIAM Journal on Scientific Computing, 41 (2019), pp. C393–C415.

[23] W. Liu and B. Vinter, *An efficient gpu general sparse matrix-matrix multiplication for irregular data*, in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 370–381.

[24] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM Journal on Computing, 15 (1986), pp. 1036–1053.

[25] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, *High-performance sparse matrix-matrix products on intel knl and multicore architectures*, in Proceedings of the 47th International Conference on Parallel Processing Companion, ICPP '18, New York, NY, USA, 2018, Association for Computing Machinery.

[26] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM Journal on Scientific Computing, 37 (2015), pp. S602–S626.

[27] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere, and P. Dubey, *High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems*, in SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–12.

[28] C. Richter, S. Schp̈s, and M. Clemens, *GPU acceleration of algebraic multigrid preconditioners for discrete elliptic field problems*, IEEE Transactions on Magnetics, 50 (2014), pp. 461–464.

[29] J. W. Ruge and K. Stüben, *Multigrid methods*, in Multigrid Methods, S. F. McCormick, ed., Society for Industrial and Applied Mathematics, 1987, ch. 4. Algebraic Multigrid, pp. 73–130.

[30] P. Sao, R. Vuduc, and X. S. Li, *A Distributed CPU-GPU Sparse Direct Solver*, Springer International Publishing, Cham, 2014, pp. 487–498.

[31] K. Stüben, in Multigrid, U. Trottenberg and A. Schuller, eds., Academic Press, Inc., USA, 2000, ch. Algebraic multigrid (AMG): an introduction with applications.

[32] U. Trottenberg and A. Schuller, *Multigrid*, Academic Press, Inc., USA, 2000.

[33] M. Wang, H. Klie, M. Parashar, and H. Sudan, *Solving Sparse Linear Systems on NVIDIA Tesla GPUs*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 864–873.

[34] U. M. Yang, *On long-range interpolation operators for aggressive coarsening*, Numerical Linear Algebra with Applications, 17 (2010), pp. 453–472.