2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

Asynchronous Multigrid Methods

Jordi Wolfson-Pou
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States of America
jwp3@gatech.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States of America
echow@cc.gatech.edu

***Abstract*—Reducing synchronization in iterative methods for solving large sparse linear systems may become one of the most important goals for such solvers on exascale computers. Research in asynchronous iterative methods has primarily considered basic iterative methods. In this paper, we examine how multigrid methods can be executed asynchronously. We present models of asynchronous additive multigrid methods, and use these models to study the convergence properties of these methods. We also introduce two parallel algorithms for implementing asynchronous additive multigrid, the global-res and local-res algorithms. These two algorithms differ in how the fine grid residual is computed, where local-res requires less computation than global-res but converges more slowly. We compare two types of asynchronous additive multigrid methods: the asynchronous fast adaptive composite grid method with smoothing (AFACx) and additive variants of the classical multiplicative method (Multadd). We implement asynchronous versions of Multadd and AFACx in OpenMP and generate the prolongation and coarse grid matrices using the BoomerAMG package. Our experimental results show that asynchronous multigrid can exhibit grid-size independent convergence and can be faster than classical multigrid in terms of solve wall-clock time. We also show that asynchronous smoothing is the best choice of smoother for our test cases, even when only one smoothing sweep is used.**

I. Introduction

Synchronization can be a bottleneck for massively parallel codes due to some parallel processes taking longer than others during phases of computation and communication (this happens on heterogeneous machines, for example). Research in asynchronous iterative methods for solving linear systems goes back to 1969 [1], studied both theoretically and in practice. However, research on these methods has mainly considered simple fixed-point iterations [1]–[7], which converge much more slowly than Krylov subspace and multigrid methods. To be clear, by "asynchronous" we are not referring to non-blocking MPI, task-based parallelism, or pipelined methods.

Multigrid methods combine a smoother with a hierarchy of coarser grids, where a smoother and a coarse grid correction are used to solve the error equations on each grid (except the coarsest grid). Multigrid methods are often used for larger problems because their convergence rate is independent of the problem size, and modern implementations scale well on massively parallel machines [8]. However, the multiplicative nature of the methods introduces many synchronization points. These may be costly for codes running on massively parallel machines, where the time it takes to synchronize may be significantly large compared to the computation time.

There have been some suggestions to use an asynchronous version of multigrid [9], [10], but only in [11] is asynchronous multigrid discussed in-depth. In [11], the authors created a "chaotic-cycle", where sawtooth-cycles (V-cycles with no pre-smoothing) are carried out asynchronously. The post-smoothing and prolongation are done asynchronously, but there is still global synchronization after a cycle is complete. In this paper, our version of asynchronous multigrid avoids all global synchronization by allowing grids to update the current approximation to the solution without ever synchronizing. We accomplish this by using additive multigrid methods, and while we are not creating a new additive method, we are creating the means by which additive methods can be executed asynchronously. We explore two additive methods: additive variants of the classical multiplicative method (Multadd) [12], and the asynchronous fast adaptive composite grid method with smoothing (AFACx) and full refinement [13]. While the literature on AFACx has discussed AFACx as an asynchronous method, this paper provides a more in-depth discussion of how additive methods can be made asynchronous.

We define two models of asynchronous multigrid, and introduce two parallel algorithms for implementing asynchronous additive multigrid, the global-res and local-res algorithms. While the amount of computation per thread is lower in the global-res algorithm than in the local-res algorithm, our experimental results show that the convergence rate of global-res is worse than local-res. We provide OpenMP results that show that asynchronous Multadd can be faster in terms of wall-clock time than the classical multiplicative multigrid method when the amount of computation per thread is reasonably small. We also show that asynchronous multigrid can exhibit grid-size independent convergence, even when the smoother is also asynchronous. Additionally, we show that using an asynchronous smoother, rather than a synchronous smoother, can reduce the wall-clock time.

II. Background

A. Classical Multiplicative Multigrid Methods

To define the classical multigrid V$(1,1)$-cycle, we first need to define the matrices used for coarse grid corrections. For grid numbers $k = 0, \ldots, \ell - 1$, where ℓ is the coarsest grid, we define:

- the two-level interpolant P_{k+1}^{k} that transfers a vector from grid $k+1$ to k. For simplicity, we will choose $(P_{k+1}^{k})^T$

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00021

as the restriction matrix that transfers a vector from grid k to $k+1$.
- the coarse grid operator $A_{k+1} = (P_{k+1}^k)^T A_k P_{k+1}^k$ at grid $k+1$.
- the smoothing iteration matrix $G_k = I - M_k^{-1} A_k$, where the smoothing matrix M_k is typically easy to invert.

We can now define the classical V(1,1)-cycle, as shown in Algorithm 1.

Algorithm 1: Multiplicative V(1,1)-Multigrid

```
1  Initialize r_0 = b
2  for t = 1, 2, ..., t_max do
3      Sequential for k = 0, ..., ℓ − 1 do
4          e_k = M_k^{-1} r_k                          ▷ pre-smoothing
5          r_{k+1} = (P_{k+1}^k)^T (r_k − A_k e_k)     ▷ restriction
6      end
7      e_ℓ = A_ℓ^{-1} r_ℓ                              ▷ exact solve on coarsest grid
8      Sequential for k = ℓ − 1 ..., 0 do
9          e_k = e_k + P_{k+1}^k e_{k+1}               ▷ coarse grid correction
10         e_k = e_k + M_k^{-1} (r_k − A_k e_k)        ▷ post-smoothing
11     end
12     x = x + e_0                                     ▷ correct solution on finest grid
13     r_0 = b − Ax                                    ▷ compute new residual
14 end
```

B. Additive Multigrid Methods

Before describing Multadd and AFACx, we define the multi-level interpolant P_k^0 for $k = 0, 1, \ldots \ell$ that transfers a vector from grid k to the finest grid. Additionally, we define $(P_k^0)^T$ as the corresponding restriction matrix. For $k = 0$, $P_0^0 = I$. In this paper, $P_k^0 = P_1^0 P_2^1 \cdots P_k^{k-1}$ and is not explicitly formed, i.e., each P_j^{j-1} for $j = 1, \ldots, k$ is applied to the vector that is being interpolated.

In additive multigrid methods, while the restriction and prolongation steps are done sequentially, smoothing on each grid can be done concurrently. This allows the corrections from each grid to be added together on the fine grid. The classical additive multigrid method is known as the BPX method [14]. One V-cycle of BPX can be written as

$$x = x + \sum_{k=0}^{\ell} P_k^0 \Lambda_k (P_k^0)^T r, \tag{1}$$

where Λ_k is the inverse of the smoothing matrix for $k = 0, \ldots, \ell - 1$ and $\Lambda_\ell = A_\ell^{-1}$.

BPX is typically used as a preconditioner because adding the corrections "over-corrects" x, resulting in a divergent solver. This over-correction occurs because the right-hand sides on the coarse grids are approximately equal, resulting in redundant corrections. In [10], BPX is modified using multi-coloring to create a convergent solver. The authors suggest that this new solver could be asynchronous, but do not precisely define what asynchronous multigrid means in this context.

1) Additive Variants of Multiplicative Multigrid (Multadd)

Multadd [12] is derived by re-writing the multiplicative method as

$$x = x + \sum_{k=0}^{\ell} \overline{P}_k^0 \Lambda_k (\overline{P}_k^0)^T r. \tag{2}$$

This method looks like BPX, but with the multi-level smoothed interpolants $\overline{P}_k^0 = \overline{P}_1^0 \cdots \overline{P}_k^{k-1}$, where the two-level smoothed interpolants are $\overline{P}_{k+1}^k = G_k P_{k+1}^k$ for $k = 0, \ldots, \ell - 1$.

If Λ_k is chosen to be the *symmetrized smoothing matrix* $\overline{M}_k^{-1} = M_k^{-T}(M_k + M_k^T - A_k)M_k^{-1}$, then Multadd is mathematically equivalent to a symmetric multiplicative V(1,1)-cycle (where G_k^T is chosen as the post-smoothing iteration matrix). If it is not a requirement for Multadd to be mathematically equivalent to the classical multiplicative multigrid method, an approximation $\overline{\Lambda}_k$ to Λ_k can also be used as the symmetrized smoother, e.g., $\overline{\Lambda}_k = D_k$. We consider this case for a hybrid smoother (see Section V).

Expressing the multiplicative method in this additive form may seem too good to be true since now each grid can be processed concurrently without sacrificing multiplicative convergence properties. However, the additive form introduces redundant computation, since grid $k+1$ must carry out the same set of prolongation and restriction steps as grid k. This suggests that Multadd would likely be slower than the multiplicative method, but if we were to make the method asynchronous, the increased computational cost might be outweighed by the gain in speed from not having to synchronize.

2) The Asynchronous Fast Adaptive Composite Grid Method

The asynchronous fast adaptive composite grid (AFACx) method [9], [13], [15], [16] is an additive multigrid method for solving PDEs on composite grids. A composite grid can be decomposed into a hierarchy of grids with different resolutions and different domain sizes. We can use AFACx as a multigrid method by thinking of the multigrid hierarchy as a hierarchy from a fully refined composite grid. There are three key steps in AFACx when computing the correction for grid k:

1) The quantity e_{k+1} is computed by smoothing on the equations $A_{k+1}e_{k+1} = r_{k+1}$, where an initial guess of zero is used and r_{k+1} is the fine grid residual restricted to grid $k+1$.
2) The quantity e_k is then computed by smoothing on the equations $A_k e_k = r_k$ using an initial guess of $P_{k+1}^k e_{k+1}$.
3) x is corrected: $x = x + P_k^k e_k - P_{k+1}^k e_{k+1}$.

The subtraction of $P_{k+1}^k e_{k+1}$ from x in the third step is what prevents an over-correction of x. This is because grids k and $k+1$ may produce approximately the same corrections, so subtracting $P_{k+1}^k e_{k+1}$ from $P_k^k e_k$ serves to remove the portion of $P_k^k e_k$ that is close in value to the correction from grid $k+1$.

Algorithm 2 shows AFACx, where V(1/1,0)-cycles are used in the inner loop (not to be confused with a V(1,1)-cycle). The 1/1 refers to using one smoothing sweep to compute e_k and one smoothing sweep to compute e_{k+1} (a V$(s_1/s_2, 0)$-cycle can also be defined). The redundant computation of $P_k^0 e_k$ and $P_{k+1}^0 e_{k+1}$ can be avoided by modifying how e_k is computed: we use an initial guess of zero and a modified right-hand side of $r_k - A_k P_{k+1}^k e_{k+1}$, as shown in lines 8 and 9 of Algorithm 2.

3) Other Additive Multigrid Methods

The first additive multigrid method was proposed in [17]. To address the over-correction issue, the corrections are done sequentially, where each correction is made to be orthogonal

Algorithm 2: V(1/1,0)-AFACx

Initialize $r = b$
for $t = 1, 2, \ldots, t_{max}$ **do**
  Sequential for $k = 0, \ldots, \ell$ **do**
    $r_k = (P_k^0)^T r$ ▷ restriction
    if $k == \ell$ **then**
      $e_k = A_k^{-1} r_k$ ▷ exact solve on coarsest grid
    else
      $e_{k+1} = M_{k+1}^{-1}(P_{k+1}^k)^T r_k$
      $e_k = M_k^{-1}(r_k - A_k P_{k+1}^k e_{k+1})$ ▷ smooth
    end
    $x = x + P_k^0 e_k$ ▷ correct solution on finest grid
  end
  $r = b - Ax$ ▷ compute new residual
end

to the new residual (the residual after the approximation is corrected) on the next finer grid. This sequential aspect, however, is not ideal for devising an asynchronous method.

Residual splitting methods [18] split the residual into a rough and smooth part using an appropriate filter. The smoother then uses the rough part of the residual, and the coarse grid correction uses the smooth part. These two corrections can then be added together without over-correcting. In [19], all grids carry out this process simultaneously. These methods converge slower than multiplicative methods, and the added cost of filtering increases the solve time.

C. Asynchronous Iterative Methods

Using a smoother to solve $Ax = b$ can be expressed as

$$x^{(t+1)} = Gx^{(t)} + f, \tag{3}$$

where $f = M^{-1}b$ and the superscript t denotes the iteration number. If we were to implement this method on a parallel computer with n processes (equal to the number of rows), each process would be responsible for *relaxing* a single row, i.e., process i would do the calculation

$$x_i^{(t+1)} = \sum_{j=1}^{n} G_{ij} x_j^{(t)} + f_i. \tag{4}$$

In the synchronous case, each process must wait for all other processes to finish computing $x^{(t+1)}$ before moving on to iteration $t+2$. Alternatively, rows can be relaxed asynchronously, where a process simply continues to iteration $t+2$ without waiting and uses the most up-to-date information to calculate $x_i^{(t+2)}$. As presented in Chapter 5 of [2], a mathematical model of an asynchronous iterative method can be written as

$$x_i^{(t+1)} = \begin{cases} \sum_{j=1}^{n} G_{ij} x_j^{(z_{ij}(t))} + f_i, & \text{if } i \in \Psi(t), \\ x_i^{(t)}, & \text{otherwise.} \end{cases} \tag{5}$$

We will refer to the asynchronous iteration number t as the *time instant*. The mapping $z_{ij}(t)$ denotes the time instant that process i reads x_j from, and $\Psi(t)$ is the set of rows that are relaxed at time instant t. The method from Equation 5 will converge if $\rho(|G|) < 1$ ($|G|$ is the element-wise absolute value of the synchronous iteration matrix). While asynchronous smoothers are not commonly used, we will show experimentally that using asynchronous smoothers can reduce the solve wall-clock time of multigrid compared to synchronous smoothers. This result is also shown in [20] for asynchronous smoothers implemented on GPUs.

III. Models of Asynchronous Multigrid Methods

In this section, we present new models of asynchronous additive multigrid methods. We emphasize that our definitions of asynchronous multigrid are different than that of asynchronous task-based processing of grids, as in [21]. The purpose of this section is not to analyze these models, but to define what asynchronous multigrid actually is, which has not been done before. In other words, the models presented in this section give us a clear picture of what is meant by asynchronous multigrid: at some time instant t, some set of grids update without any of the grids synchronizing, i.e., each grid has no information about the progress made by other grids. In the case that some grid update is delayed, this means that multiple corrections from other grids have been performed before the delayed grid has corrected once.

The first model is the *semi-asynchronous* model (semi-async),

$$x^{(t+1)} = x^{(t)} + \sum_{k \in \Psi(t)} B_k(x^{(z_k(t))}), \tag{6}$$

and the second is the *fully asynchronous* model (full-async),

$$x^{(t+1)} = x^{(t)} + \sum_{k \in \Psi(t)} B_k(x_1^{(z_{k1}(t))}, \ldots, x_n^{(z_{kn}(t))}). \tag{7}$$

We refer to these two models as the *solution-based* versions of semi-async and full-async since $x^{(t)}$ is written to and read from by all grids. If $k > 0$, B_k is a function that outputs the correction for grid k. If $k = 0$, the output of B_k corresponds to one smoothing sweep applied to the error equations. For example, for grid k in the semi-async model of Multadd, $B_k(x) = \overline{P}_k^0 \Lambda_k (\overline{P}_k^0)^T (b - Ax^{(z_k(t))})$. In these models, the computation of $B_k(x)$ is carried out synchronously by the threads belonging to grid k. However, an asynchronous smoother could also be used, i.e., we could apply Λ_k asynchronously.

There are similarities between Equations 6 and 7, and Equation 5. First, the set $\Psi(t)$ is now the set of grids correcting the solution at time instant t. Second, we now have the mappings $z_k(t)$ and $z_{ki}(t)$ for $i = 1, \ldots, n$. These two mappings are what make semi-async and full-async different from each other. For full-async, x can be corrected by one grid while a different grid is simultaneously reading x from memory. The result is that the copy of x read from memory contains elements from different time instants. For semi-async, all components of x read from memory come from the same time instant.

Alternatively, with the observation that any fixed-point iteration (Equation 3) can be expressed as

$$\begin{aligned} x &= x + M^{-1} r \\ r &= r - AM^{-1} r \end{aligned} \tag{8}$$

we can also express the semi-async and full-async models in terms of the residual:

$$r^{(t+1)} = r^{(t)} - A \sum_{k \in \Psi(t)} C_k(r^{(z_k(t))}), \quad (9)$$

and,

$$r^{(t+1)} = r^{(t)} - A \sum_{k \in \Psi(t)} C_k(r_1^{(z_{k1}(t))}, \ldots, r_n^{(z_{kn}(t))}), \quad (10)$$

where C_k is defined similarly to B_k. We refer to these two models as the *residual-based* versions of semi-async and full-async, respectively. In the case of semi-async, there is no difference between the residual-based and solution-based versions, given that, for all k and t, $z_k(t)$ is the same in both cases. However, for full-async, the solution-based and residual-based versions are different since the vectors $(r_1^{(z_{k1}(t))}, \ldots, r_n^{(z_{kn}(t))})^T$ and $b - A(x_1^{(z_{k1}(t))}, \ldots, x_n^{(z_{kn}(t))})^T$ can be different, even when $z_{k1}(t), \ldots, z_{kn}(t)$ are the same for all k and t.

To demonstrate the difference in convergence among our four models (solution-based and residual-based versions of semi-async and full-async), we simulated asynchronous multigrid by implementing Equations 6, 7, and 10 as solvers to be executed sequentially. In the simulation, grid k has an *update probability* p_k, i.e., grid k has the probability p_k of being in $\Psi(t)$ at time instant t. In our experiments, p_k is determined in advance (before we start solving $Ax = b$) by sampling from a uniform random integer distribution in the range $[\alpha, 1]$, where α is the *minimum update probability* and $1 > \alpha > 0$. As α decreases, the grids will become more "out of sync", i.e., the values of p_k will have a higher variation resulting in some grids updating more often than others.

If $k \in \Psi(t)$, the value of $z_k(t)$ ($z_{ki}(t)$ in the case of full-async) is chosen randomly by sampling from a uniform random integer distribution in the range $(\min(z_k(\tau_k), t-\delta), t]$. The time instant τ_k denotes the last time instant that grid k read from. The *maximum read delay* δ is defined as the maximum value of $t - z_k(t)$ and denotes the minimum past time instant that grid k can read from. In other words, we are assuming two things: 1) a grid cannot read older information than what has already been read ($z_k(\tau_k)$ term), and 2) even if a grid updates very slowly compared to other grids, there is still some bound on how old the information can be that is read from memory ($t - \delta$ term).

Each grid stops updating after 20 updates, and the iteration is terminated after all grids have completed 20 updates. We compare this to 20 V(1,1)-cycles of synchronous multigrid. For our test framework, we used the 27pt test set (see Section V for matrix descriptions) with mesh sizes ranging from $40 \times 40 \times 40$ to $80 \times 80 \times 80$. Weighted Jacobi was used as a smoother with a weight of .9. We used the BoomerAMG package [22] to generate the interpolation and coarse grid matrices. For our BoomerAMG options, we chose HMIS coarsening with one aggressive level, and classical modified interpolation.

Figure 1 demonstrates the effect of α on the convergence of semi-async when $\delta = 0$. The figure shows the relative residual 2-norm versus the grid length for Multadd and AFACx. Each data point is the mean relative residual 2-norm of 20 runs. Each figure shows synchronous multigrid and simulations of semi-async with different values of α. The figures show that with small values of α, convergence is slower, but the convergence is still independent of the grid length.

Figure 2 demonstrates the effect of δ on the convergence of full-async with $\alpha = .1$. Each figure shows synchronous multigrid and simulations of either the solution-based or residual-based versions of full-async. These results show that with larger values of δ, convergence is slower, but the convergence is still independent of the grid length. Additionally, the residual-based versions converge faster than the solution-based versions for large values of δ.

IV. Asynchronous Multigrid for Shared Memory

This section presents asynchronous additive multigrid methods for shared memory parallel computers. The main issue to address is the computation of the residual on the fine grid. We first describe two implementations for synchronous two-grid Multadd, and then extend these to the asynchronous case. The implementations are mathematically the same when executed synchronously. We proceed with an example. We have five threads, t_0, t_1, t_2, t_3 and t_4. The fine grid has seven points, and the coarse grid has three points. Recall that one V(1,1)-

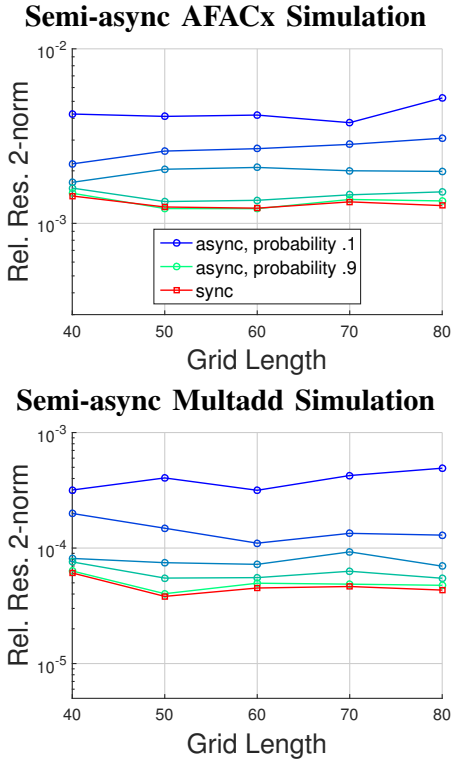

Fig. 1. Final relative residual 2-norm after 20 V-cycles versus grid length for the semi-asynchronous multigrid model (Equation 6) for AFACx and Multadd. A maximum delay of zero is used. Results are shown for five minimum update probabilities, where blue-to-green corresponds to increasing minimum update probability. The 27pt test set is used (see Section V). These results show that even with a small minimum update probability, asynchronous multigrid still exhibits grid-size independent convergence.

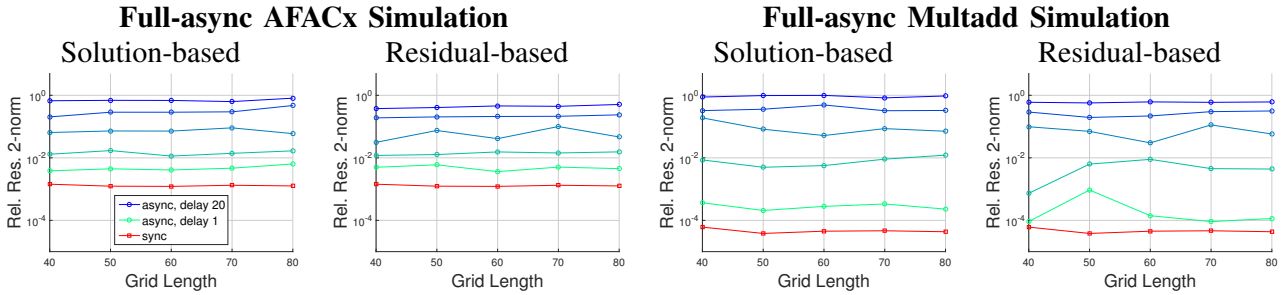

Fig. 2. Final relative residual 2-norm after 20 V-cycles versus grid length for the full-asynchronous multigrid model. The solution-based (Equation 7) and residual-based versions (Equation 10) of AFACx and Multadd are shown. A minimum update probability of .1 is used and results for five maximum delay values are shown, where blue-to-green gradient corresponds to decreasing maximum delay. The 27pt test set is used (see Section V). These results show that even with large delays, asynchronous multigrid still exhibits grid-size independent convergence.

cycle of Multadd is

$$\begin{aligned} r &= b - Ax \\ x &= x + \Lambda_0 r + \overline{P}_1^0 A_1^{-1} (\overline{P}_1^0)^T r. \end{aligned} \tag{11}$$

Threads t_0 and t_1 are responsible for computing $\Lambda_0 r$, and threads t_2, t_3 and t_4 are responsible for computing $\overline{P}_1^0 A_1^{-1} (\overline{P}_1^0)^T r$. We say that t_0 and t_1 are assigned to grid 0, and t_2, t_3 and t_4 are assigned to grid 1. In the general case, threads are distributed among the grids to balance the amount of "work", where the work for a grid is approximately the number of flops required for that grid to carry out its correction.

We present two algorithms for parallel synchronous Multadd which differ only in how r is computed:

1) **global-res:** Just like in classical multigrid, each thread would be responsible for computing some number of elements of the fine grid residual r, and r would be computed using a parallel SpMV operation using all five threads. We call this the *global-res algorithm* since, in addition to x, r is a "global" variable. Here, "global" refers to memory that can be read by all threads, while memory that is "local" to a grid refers to memory that can be read only by threads assigned to that grid. Algorithm 3 and Figure 3 show global-res for this example, where Sync() denotes the synchronization of the threads listed. In line 1 of Algorithm 3, all threads take part in computing r using a parallel SpMV operation. If we are using OpenMP, the computation of r would be parallelized using a parallel for loop with a static scheduling.

   In the if statements, only the threads assigned to a grid take part in each operation, which are also carried out with parallel loops. For example, in the case of grid 0, if we are using OpenMP, $\Lambda_0 r$ would be computed using a parallel for loop but only with the threads t_0 and t_1. In the case of grid 1, the application of $\overline{P}_1^0$, A_1^{-1}, and $(\overline{P}_1^0)^T$ to a vector are carried out by threads t_2, t_3 and t_4 (SpMV and triangular solve operations), where the three threads synchronize after each application.

   Note that both grids update x concurrently in lines 6 and 11, which creates a race condition. We will discuss later in this section how these race conditions are handled.

2) **local-res:** Only x is a global variable. Threads assigned to a grid would read x from memory and then compute a local residual, e.g., threads t_0 and t_1 would compute the local residual r^0 using a parallel for loop. We call this the *local-res algorithm*, which is shown in Algorithm 4 and in Figure 3. The two threads first read x in line 1, and then in lines 4 and 9, threads t_0 and t_1 compute r^0 and r^1, respectively, which are the local residuals. The rest of the algorithm is the same as that of global-res.

In Algorithms 3 and 4, to make these algorithms asynchronous, we simply replace all Sync$(t_0, t_1, t_2, t_3, t_4)$ operations with Sync(t_0, t_1) and Sync(t_2, t_3, t_4), i.e., we replace all global synchronizations with synchronizations of subsets of threads, where each subset is the set of threads assigned to a grid, and the union of all the subsets is the set of all threads. This means that there is some synchronization, but only among threads assigned to the same grid.

Algorithm 3: **global-res** for two-grid synchronous Multadd with five threads

```
1  r = b − Ax
2  Sync(t0, t1, t2, t3, t4)
3  if threads t0, t1 then
4      Sync(t0, t1)
5      e^0 = Λ0 r
6      Sync(t0, t1)
7      x = x + e^0
8  end
9  if threads t2, t3, t4 then
10     c = (P̄_1^0)^T r
11     Sync(t2, t3, t4)
12     d = A_1^{-1} c
13     Sync(t2, t3, t4)
14     e^1 = P̄_1^0 d
15     Sync(t2, t3, t4)
16     x = x + e^1
17 end
18 Sync(t0, t1, t2, t3, t4)
```

Algorithm 4: **local-res** for two-grid synchronous Multadd with five threads

```
1  x^0 = x^1 = x
2  Sync(t0, t1, t2, t3, t4)
3  if threads t0, t1 then
4      r^0 = b − Ax^0
5      Sync(t0, t1)
6      e^0 = Λ0 r^0
7      Sync(t0, t1)
8      x = x + e^0
9  end
10 if threads t2, t3, t4 then
11     r^1 = b − Ax^1
12     Sync(t2, t3, t4)
13     c = (P̄_1^0)^T r^1
14     Sync(t2, t3, t4)
15     d = A_1^{-1} c
16     Sync(t2, t3, t4)
17     e^1 = P̄_1^0 d
18     Sync(t2, t3, t4)
19     x = x + e^1
20 end
21 Sync(t0, t1, t2, t3, t4)
```

A problem with the global-res algorithm comes from how the residual is computed. As stated earlier, if a grid update is severely delayed, the faster grids may do many corrections

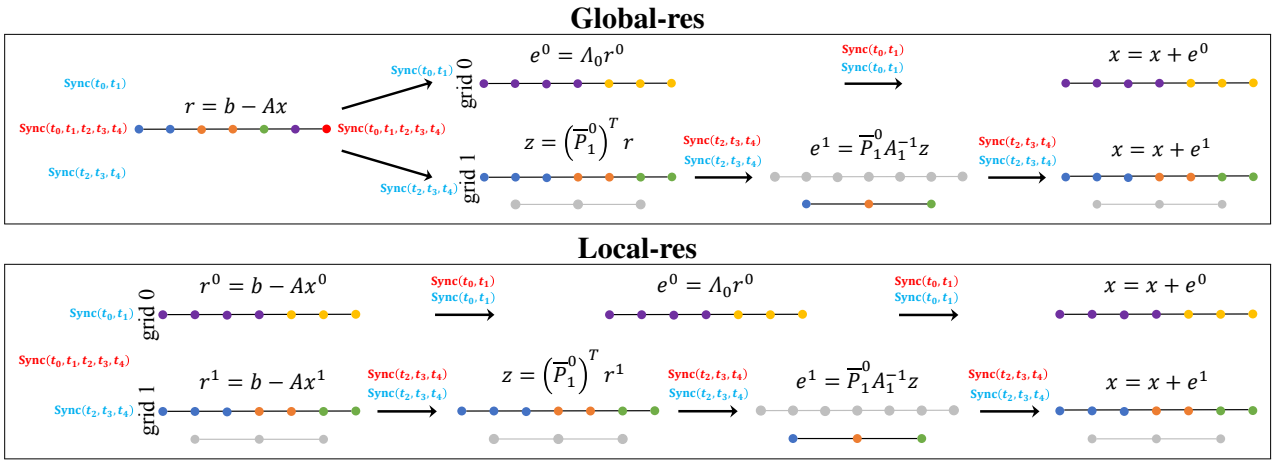

Fig. 3. Global-res and local-res partitionings for the Multadd example presented in Section IV for each step of the computation of the corrections e^0 and e^1. Arrows denote moving to the next step of the computation. Sync() denotes a synchronization point, where the list of threads passed to Sync() denotes the threads that synchronize. Blue **Sync()** denotes a synchronization for asynchronous multigrid, and red **Sync()** denotes a synchronization point for synchronous multigrid. Colored points denote points used in a calculation, where t_0 is assigned the purple points, t_1 is assigned the yellow points, t_2 is assigned the blue points, t_3 is assigned the orange points, and t_4 is assigned the green points. Gray points denote points not used in a calculation.

with a residual that has some components that are up-to-date, and other components that are very out-of-date. We will see in Section VI that this can result in asynchronous multigrid diverging or converging more slowly than synchronous multigrid. The local-res algorithm does not suffer from this problem, but requires more computation per thread.

As mentioned earlier, when the global variable x (x and r in the case of global-res) is updated, we must handle race conditions since threads assigned to different grids update x concurrently. One option is to use a mutex lock. For this option, all grids have a master thread. All threads assigned to grid k block until the master thread for grid k acquires a mutex lock. Once the lock is acquired, the variable is updated by all threads assigned to grid k using a parallel for loop. We call this option the *lock-write* option. The second option is to use an atomic fetch-and-add operation inside the parfor loop. We call this option the *atomic-write* option.

We can now write asynchronous multigrid, as shown in Algorithm 5. In the algorithm:

- The k superscript denotes a variable stored in the local memory of grid k.
- For grid k, the operations Smooth(), Prolong(), Restrict(), Read(), Ax^k, and $x + e^k$ are carried using blocking parallel for loops (threads synchronize after completing the loop), where only the threads assigned to k carry out the loops.
- x and r are global, i.e., can be accessed by any grid.
- The flag *rescomp_type* (local or global) specifies whether global-res or local-res is used.
- The Write() operation handles race conditions (explained above) when writing to a global variable.
- The **GlobalParFor** loop is executed by all threads, which is the global update of the residual for global-res. The **No Wait** denotes a non-blocking parallel for loop, which is conceptually the same as adding a "no wait" clause to an OpenMP parfor loop.
- In lines 19-26, the residual can instead be updated as $r = r - Ae$ instead of $r = b - Ax$ as outlined in Section III.

Algorithm 5: Asynchronous multigrid for grid k

```
1  Initialize r^k = r = b                          ▷ initialize local residuals
2  while grid k has not converged do               ▷ procedure for grid k
3      r^k = Restrict(r^k)
4      if k == ℓ then
5          e^k = Smooth(A_k, r^k)
6      else
7          e^k = ExactSolve(A_k, r^k)
8      end
9      e^k = Prolong(e^k)
10     x = Write(x + e^k)                          ▷ correct x
11     x^k = Read(x)                               ▷ store x to local memory
12     if rescomp_type == local then
13         r^k = b − Ax^k                          ▷ recompute local residual
14     else
15         No Wait GlobalParfor i = 1, ..., n do
16             r_i = Write(b_i − Σ_{j=1}^n a_ij x_j)  ▷ update global residual
17         end
18         r^k = Read(r)                           ▷ store r to local memory
19     end
20 end
```

In terms of the models presented in Section III, only local-res with the lock-write option can be modeled by semi-async (Equation 6). All other variations of Algorithm 5 can be modeled by full-async (Equations 7 and 10).

V. Test Framework for Experimental Results

For our numerical results, we used an Intel Xeon Phi Knights Landing (KNL) processor with 68 cores and 272 threads. We implemented synchronous multigrid (Mult) using OpenMP parallel for loops with static scheduling. For synchronous Multadd and AFACx, each grid was assigned threads in the same way as the asynchronous local-res implementation. This thread partitioning is used only to do coarse grid corrections concurrently. At the end of a single cycle, all threads synchronize and carry out an SpMV to compute the residual using an OpenMP parallel for loop. This is the same way the residual

is computed in Mult.

We experimented with four different smoothers: weighted Jacobi (ω-Jacobi), ℓ_1-Jacobi [23], hybrid Jacobi Gauss-Seidel (hybrid JGS) [23], and asynchronous Gauss-Seidel (async GS). As in ω-Jacobi, ℓ_1-Jacobi uses a diagonal smoothing matrix, where the diagonal entries are the L1 norms of the rows of A, i.e., $M_{ii} = \sum_{j=1}^{n} |a_{ij}|$. It can be shown that if A is symmetric and positive-definite, the error monotonically decreases in the A-norm when ℓ_1-Jacobi is used as a smoother.

The hybrid Jacobi Gauss-Seidel smoother can be thought of as an inexact block Jacobi method where the blocks are solved inexactly using a small number of Gauss-Seidel sweeps. In this paper, we only consider using one sweep. For parallel smoothers, the number of subdomains is equal to the number of processes or threads, making the method highly parallel. However, without proper weighting [24] or using an ℓ_1 variation of the method [23], the method can diverge if many subdomains are used. Asynchronous Gauss-Seidel is an asynchronous version of hybrid JGS. For a shared memory implementation, a thread relaxes a subset of rows (approximately n/p rows), and immediately writes the updated information to memory after each relaxation. This means that information that is read from memory could be a mix of new and old information, which is modeled in Equation 5.

We used BoomerAMG [22] to generate the prolongation and coarse grid matrices for all our multigrid methods. For Multadd, if ℓ_1-Jacobi was used as a smoother, we used the ℓ_1-Jacobi iteration matrix to construct the smoothed interpolants. For all other smoothers, we used the ω-Jacobi iteration matrix. We did this for performance reasons, i.e., we wanted to keep the smoothed interpolants sparse, even though the convergence may be slower than when using a hybrid or asynchronous smoother. For example, for a V(1,1)-cycle of Multadd with hybrid JGS, $\overline{P}_{k+1}^{k} = (I - \omega D_k^{-1} P_{k+1}^{k})$ and Λ_k is the block diagonal matrix with blocks $L_{k1}^{-1}, \ldots, L_{kp}^{-1}$, where L_{ki} is the lower triangular part of block i of A_k, for $i = 1, \ldots, p$. In our results, we only consider using one smoothing sweep since it is not clear how to do multiple sweeps with Multadd while keeping the smoothed interpolants fixed.

We used four sets of test matrices with different problem sizes within each set. Two of these sets were generated using the MFEM software package [25]:

- The three-dimensional Laplace matrices in a cube discretized using the 7-point and 27-point centered difference methods. We will refer to these two sets as 7pt and 27pt.
- The three-dimensional Laplace matrices in a sphere discretized using a NURBS mesh [26] and H^1 nodal finite elements. These matrices were generated using the MFEM package [25]. We will refer to these matrices as MFEM Laplace.
- Three-dimensional linear elasticity matrices modeling a multi-material cantilever beam using a tetrahedral mesh and H^1 nodal finite elements. These matrices were generated using the MFEM package [25]. We will refer to these matrices as MFEM Elasticity.

We used random right-hand sides with values in $[-1, 1]$.

In our implementations, we do not try to detect when the global relative residual norm $\|r\|_2/\|b\|_2$ falls below some specified tolerance τ. This would require a subset of grids to compute a norm, which is an extra delay on that grid, and the relative residual norm generally does not monotonically decrease. There are two convergence criteria we use to detect when t_{max} V-cycles have been carried out:

- **Criterion 1**: A grid immediately breaks from the main loop when it has carried out t_{max} corrections. This means that grids can finish iterating before other grids have finished. This is the same criterion used in the simulations in Section III.
- **Criterion 2**: A single master thread is in charge of making sure all grids have carried out at least t_{max} corrections. This thread then sets a flag indicating that the iteration must terminate. For a thread that is not the master, it reads this flag after finishing computing a correction. If the flag is not set, the thread computes another correction. Otherwise, it exits the main solve loop.

To find the wall-clock time required to reduce $\|r\|_2/\|b\|_2$ below some tolerance, we plot $\|r\|_2/\|b\|_2$ versus wall-clock time, saving time stamps of $\|r\|_2/\|b\|_2$ for doing a small to large number of cycles, e.g., we do $5, 10, \ldots, 100$ V-cycles, saving $\|r\|_2/\|b\|_2$ and the wall-clock time for each number of V(1,1)-cycles. When saving the wall-clock times, we do multiple runs for each number of cycles and take the mean of the wall-clock times for those runs (for asynchronous methods, we also take the mean of the relative residual 2-norms). We then find the wall-clock time corresponding to the first occurrence of $\|r\|_2/\|b\|_2 < \tau$. For all our experiments, we took the average of 20 runs and set $\tau = 10^{-9}$.

VI. Experimental Results

We first show that asynchronous multigrid methods can exhibit grid-size independent convergence. Figure 4 shows $\|r\|_2/\|b\|_2$ after 20 V(1,1)-cycles (see Section V for how a V-cycle is defined in the asynchronous case) versus the grid length for the 7pt and 27pt test sets (a grid length of 40 denotes a $40 \times 40 \times 40$ mesh) using 68 threads. Each data point is the mean $\|r\|_2/\|b\|_2$ of 20 runs. For the asynchronous methods, we used Criterion 1 for convergence detection (see Section IV). Results for ω-Jacobi and async GS smoothing are shown. For our BoomerAMG options, we chose HMIS coarsening with one aggressive level, and classical modified interpolation. If "sync" is written next to a legend entry, the method is synchronous. Otherwise, the method is asynchronous.

Figure 4 shows that all the asynchronous methods approximately achieve grid-size independent convergence, even when using async GS as the smoother. We can also see that in most cases, global-res results in a solver that converges more slowly than when using local-res. This is due to grids using fine grid residual values that are delayed. In other words, since grid k computes a correction using values of r_0 that are updated

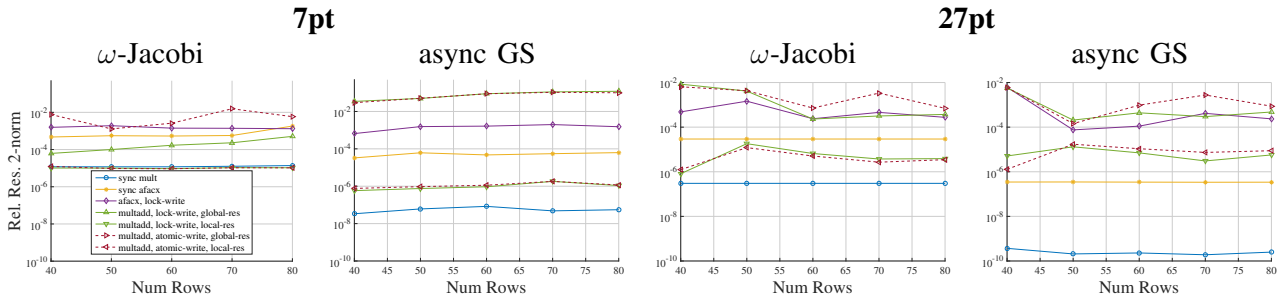

Fig. 4. Relative residual 2-norm versus number of rows for 20 V(1,1)-cycles and 68 threads. Results for the 7pt and 27pt test sets are shown. For each test set, results for two smoothers are shown. For the asynchronous methods, we used Criterion 1 as our stopping criterion (see Section V), and each data point is the mean relative residual 2-norm of 20 runs. The figures show that asynchronous multigrid methods can exhibit grid-size independent convergence.

exclusively by other grids, grid k could be using very old values if another grid update is delayed. Figure 5 shows the same experiment but with the MFEM Laplace test set and no aggressive coarsening. Multadd local-res lock-write exhibits grid-size independent convergence. AFACx (synchronous and asynchronous) and Multadd global-res did not exhibit grid-size independent convergence for this test set.

Table I shows results for four test matrices, one from each test set, and 272 threads. For the asynchronous methods, we used Criterion 2 for convergence detection (see Section IV). *Corrects* is the average number of corrections of all the grids divided by the number of grids. For each test matrix, results for four smoothers are shown. For our BoomerAMG options,

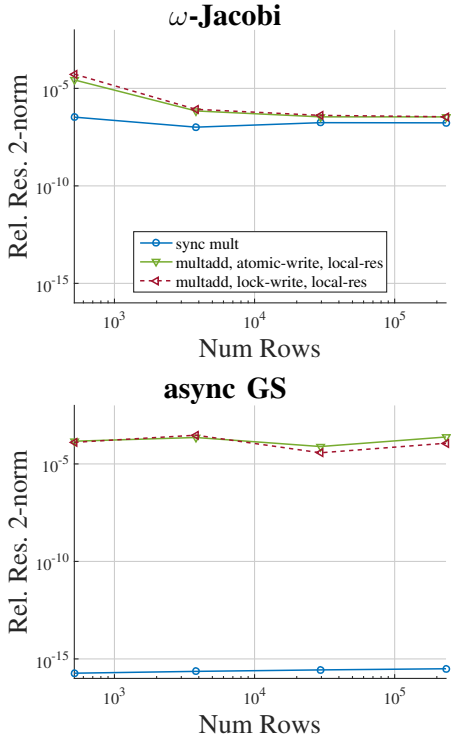

Fig. 5. Relative residual 2-norm versus number of rows for 20 V(1,1)-cycles and 68 threads. The MFEM Laplace matrix is used and results for the ω-Jacobi and async GS smoothers are shown. The figures show that asynchronous multigrid can exhibit grid-size independent convergence.

we chose HMIS coarsening with two aggressive levels, and classical modified interpolation. The r- prefix in r-Multadd denotes that Multadd was implemented using the residual-based implementation (in Section IV, see the last bullet of the explanation of Algorithm 5). These results show that, with the exception of async GS for the MFEM Elasticity matrix, asynchronous Multadd requires the lowest wall-clock time, even if it requires more computation than Mult (higher number in the Corrects column). Additionally, using atomic operations is slower than using locks, with the exception of r-Multadd for the MFEM Laplace matrix with the async GS smoother. In some cases (MFEM Laplace with ω-Jacobi smoothing, and 7pt with hybrid JGS smoothing), global-res is the best solver. In most cases, local-res is the best solver since it requires significantly fewer V-cycles to converge. Finally, using async GS smoothing always requires the lowest number of V(1,1)-cycles and least wall-clock time compared to the other smoothers.

Figure 6 shows the wall-clock time versus the number of threads for the same four matrices from Table I. The BoomerAMG options are the same options used in Table I and w-Jacobi smoothing is used. Each subfigure shows three methods: sync Mult, sync Multadd lock-write, and Multadd lock-write local-res. In all subfigures, we can see that with a low number of threads, Mult is typically the fastest since synchronization is not a large cost compared to the cost of computation. However, asynchronous Multadd is the fastest for a sufficiently large number of threads, and scales better, i.e., as the number of threads increases, the wall-clock time of asynchronous Multadd does not increase as much as that of Mult. This provides a good outlook for distributed memory, where the number of parallel processes is orders of magnitude higher, and in the case of exascale machines, the problem size per process may be quite small. We also see that synchronous Multadd scales better than Mult, demonstrating that computing corrections concurrently can be beneficial. This is because there is only global communication on the fine grid for synchronous Multadd, whereas for Mult, there is global synchronization on every grid.

VII. Conclusion

In this paper, we introduced asynchronous multigrid methods. These methods are asynchronous versions of additive multi-

Table I: Timing results for four test matrices, and for each matrix, four smoothers. 272 threads are used. For each smoother, results for all multigrid methods are shown (see Section IV for explanations of lock-write, atomic-write, local-res, and global-res). The † marker indicates that a method diverged. For each smoother, the **bolded** number indicates the lowest wall-clock time among all the methods. These results show that asynchronous Multadd is generally faster than the classical multiplicative multigrid method (Mult) in terms of wall-clock time, and async GS is the best smoother for all matrices.

7pt: 27,000 rows and 183,600 non-zero values

	ω-Jacobi, $\omega = .9$			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles
sync Mult	0.1164	75	75	0.1927	120	120	0.1009	65	65	0.0828	55	55
sync Multadd, lock-write	0.0305	75	75	0.0490	120	120	0.0405	100	100	0.0323	80	80
sync Multadd, atomic-write	0.0299	75	75	0.0465	120	120	0.0393	100	100	0.0322	80	80
sync AFACx, lock-write	0.0489	135	135	†	†	†	0.0420	115	115	0.0339	95	95
sync AFACx, atomic-write	0.0481	135	135	†	†	†	0.0418	115	115	0.0337	95	95
AFACx, lock-write	0.0429	154	110	†	†	†	0.0430	142	110	0.0349	115	90
AFACx, atomic-write	0.0575	160	120	†	†	†	0.0533	138	110	0.0466	121	95
Multadd, lock-write, global-res	0.0249	89	70	†	†	†	**0.0267**	97	75	0.0591	192	155
Multadd, lock-write, local-res	**0.0200**	73	45	**0.0326**	123	75	0.0269	97	60	**0.0203**	74	45
Multadd, atomic-write, global-res	0.0286	78	70	†	†	†	0.0351	97	85	0.0293	80	70
Multadd, atomic-write, local-res	0.0259	71	50	0.0441	123	85	0.0360	98	70	0.0310	86	60
r-Multadd, atomic-write, local-res	0.0257	69	50	0.0452	122	90	0.0359	94	70	0.0281	76	55

27pt: 27,000 rows and 681,472 non-zero values

	ω-Jacobi, $\omega = .9$			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles
sync Mult	0.0939	65	65	0.1553	105	105	0.0795	55	55	0.0581	40	40
sync Multadd, lock-write	0.0259	65	65	0.0414	105	105	0.0349	90	90	0.0281	70	70
sync Multadd, atomic-write	0.0250	65	65	0.0400	105	105	0.0355	90	90	0.0254	65	65
sync AFACx, lock-write	0.0451	120	120	†	†	†	0.0383	100	100	0.0282	75	75
sync AFACx, atomic-write	0.0429	120	120	†	†	†	0.0380	100	100	0.0274	75	75
AFACx, lock-write	0.0420	120	85	†	†	†	0.0418	110	85	0.0324	85	65
AFACx, atomic-write	0.0465	112	85	†	†	†	0.0464	108	85	0.0385	90	70
Multadd, lock-write, global-res	0.0254	79	65	†	†	†	0.0321	119	95	0.0481	150	125
Multadd, lock-write, local-res	**0.0206**	58	40	**0.0304**	93	60	**0.0280**	85	55	**0.0231**	65	45
Multadd, atomic-write, global-res	0.0339	98	95	†	†	†	0.0357	105	100	0.0342	99	95
Multadd, atomic-write, local-res	0.0223	56	40	0.0336	89	60	0.0308	81	55	0.0253	65	45
r-Multadd, atomic-write, local-res	0.0254	62	45	0.0362	89	65	0.0391	92	70	0.0282	69	50

MFEM Laplace: 29,521 rows and 781,297 non-zero values

	ω-Jacobi, $\omega = .5$			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles
sync Mult	0.2404	150	150	0.2473	155	155	†	†	†	0.0924	60	60
sync Multadd, lock-write	0.0924	150	150	0.0964	155	155	0.0847	140	140	0.0588	95	95
sync Multadd, atomic-write	0.0909	150	150	0.0949	155	155	0.0845	140	140	0.0586	95	95
sync AFACx, lock-write	0.1316	295	295	†	†	†	†	†	†	0.0572	100	100
sync AFACx, atomic-write	0.1314	295	295	†	†	†	†	†	†	0.0563	100	100
AFACx, lock-write	0.1442	300	235	†	†	†	†	†	†	0.0730	135	120
AFACx, atomic-write	0.1532	296	230	†	†	†	†	†	†	0.0751	127	115
Multadd, lock-write, global-res	**0.0737**	189	160	†	†	†	0.0677	177	145	0.0652	166	140
Multadd, lock-write, local-res	0.0782	148	110	**0.0818**	154	115	**0.0636**	127	90	0.0513	94	70
Multadd, atomic-write, global-res	0.0788	172	160	†	†	†	0.0721	159	145	0.0691	147	140
Multadd, atomic-write, local-res	0.0836	149	115	0.0899	158	120	0.0732	135	100	0.0564	97	75
r-Multadd, atomic-write, local-res	0.0790	145	110	0.0845	153	115	0.0644	122	90	**0.0512**	93	70

MFEM Elasticity: 37,281 rows and 251,617 non-zero values

	ω-Jacobi, $\omega = .5$			ℓ_1-Jacobi			hybrid JGS			async GS		
method	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles	time	corrects	V-cycles
sync Mult	0.3425	190	190	0.3352	190	190	0.1736	100	100	0.1465	85	85
sync Multadd, lock-write	0.1367	190	190	0.1361	190	190	0.1134	165	165	0.0902	125	125
sync Multadd, atomic-write	0.1337	190	190	0.1346	190	190	0.1119	165	165	**0.0888**	125	125
sync AFACx, lock-write	0.2301	385	385	†	†	†	0.1150	195	195	0.1109	170	170
sync AFACx, atomic-write	0.2269	385	385	†	†	†	0.1134	195	195	0.1107	170	170
AFACx, lock-write	0.2103	404	310	†	†	†	†	†	†	0.1603	268	235
AFACx, atomic-write	0.2378	405	315	†	†	†	†	†	†	0.2006	301	260
Multadd, lock-write, global-res	†	†	†	†	†	†	†	†	†	†	†	†
Multadd, lock-write, local-res	**0.1098**	192	145	**0.1099**	195	145	**0.0934**	171	125	0.0904	152	115
Multadd, atomic-write, global-res	†	†	†	†	†	†	†	†	†	†	†	†
Multadd, atomic-write, local-res	0.1266	201	160	0.1268	202	160	0.1174	192	150	0.1008	156	125
r-Multadd, atomic-write, local-res	0.1177	193	155	0.1185	195	155	0.1014	169	135	0.0927	150	120

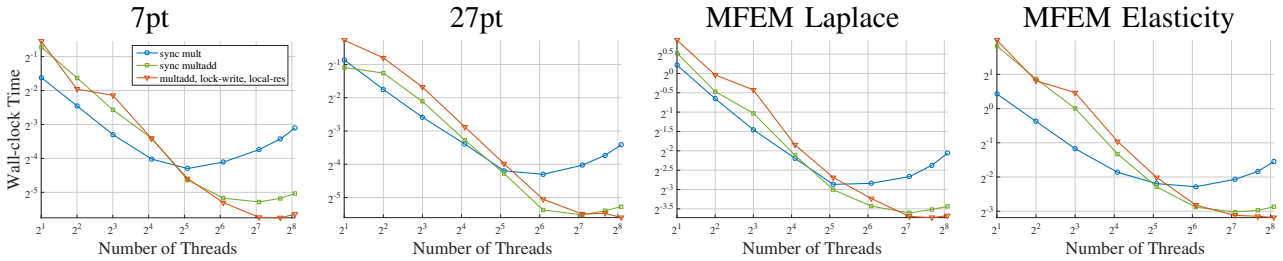

Fig. 6. Wall-clock time versus number of threads for the 7pt, 27pt, MFEM Laplace, and MFEM Elasticity matrices (see Table I) are shown with ω-Jacobi smoothing. The BoomerAMG options are the same as that of Table I. Each data point is the mean $\|r\|_2/\|b\|_2$ of 20 runs. The figures show that asynchronous multigrid is faster than synchronous multigrid for a sufficiently large number of threads, and typically scales better.

grid methods, specifically, the AFACx and Multadd methods. Although we have used the familiar term "V-cycle" in these methods to mean one set of corrections from every grid in the multigrid hierarchy, there is no concept of a cycle in asynchronous additive multigrid methods: corrections from all grids are performed simultaneously and do not wait for each other. Our models and experiments show that grid-independent convergence can be retained in this asynchronous setting. However, in our simulations and experimental tests, the number of corrections from each grid is approximately balanced. It is possible to show that if the number of corrections is not balanced (e.g., far more corrections from some grids compared to others), then grid-independent convergence is lost.

We showed that asynchronous Multadd can be faster (in terms of solver wall-clock time) than the classical synchronous multiplicative method when the problem size per thread is sufficiently small, which suggests that asynchronous multigrid would be the method of choice for massively parallel machines. Additionally, we showed that an asynchronous smoother is the best choice in smoother, even when using just one smoothing sweep. Looking towards distributed memory parallelism, we believe that the global-res approach is the most natural way to implement a distributed asynchronous multigrid method since we do not have to compute multiple fine grid residuals.

VIII. Acknowledgments

This work was supported by the U.S. Department of Energy under Award Number DE-SC-0012538 and used the resources of the National Energy Research Scientific Computing Center under Contract No. DE-AC02-05CH11231.

References

[1] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199–222, 1969.

[2] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel Iterative Algorithms: From Sequential to Grid Computing*. Chapman & Hall/CRC, 2007.

[3] A. Frommer and D. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, no. 12, pp. 201–216, 2000.

[4] Z. Peng, Y. Xu, M. Yan, and W. Yin, "ARock: An algorithmic framework for asynchronous parallel coordinate updates," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. 2851–2879, 2016.

[5] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *International Journal on High Performance Computing Applications*, vol. 28, no. 1, pp. 97–111, 2014.

[6] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.

[7] J. Wolfson-Pou and E. Chow, "Convergence models and surprising results for the asynchronous Jacobi method," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 940–949, 2018.

[8] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's multigrid solvers to 100,000 cores," *High-Performance Scientific Computing: Algorithms and Applications*, pp. 261–279, 2012.

[9] L. Hart and S. McCormick, "Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Basic ideas," *Parallel Computing*, vol. 12, no. 2, pp. 131–144, 1989.

[10] X. C. Tai and P. Tseng, "Convergence rate analysis of an asynchronous space decomposition method for convex minimization," *Mathematics of Computation*, vol. 71, no. 239, pp. 1105–1135, 2002.

[11] J. Hawkes, G. Vaz, A. Phillips, C. Klaij, S. Cox, and S. Turnock, "Chaotic multigrid methods for the solution of elliptic equations," *Computer Physics Communications*, vol. 237, pp. 26–36, 2019.

[12] P. S. Vassilevski and U. M. Yang, "Reducing communication in algebraic multigrid using additive variants," *Numerical Linear Algebra with Applications*, vol. 21, no. 2, pp. 275–296, 2014.

[13] D. Quinlan, "Adaptive mesh refinement for distributed parallel architectures," Ph.D. dissertation, University of Coloardo Denver, 1993.

[14] J. H. Bramble, J. E. Pasciak, and J. Xu, "Parallel multilevel preconditioners," *Mathematics of Computation*, vol. 55, no. 191, pp. 131–144, 1990.

[15] B. Lee, S. McCormick, B. Philip, and D. Quinlan, "Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations," *SIAM Journal on Numerical Analysis*, vol. 42, no. 1, pp. 130–152, 2004.

[16] ——, "Asynchronous fast adaptive composite-grid methods: Numerical results," *SIAM Journal on Scientific Computing*, vol. 25, no. 2, pp. 682–700, 2003.

[17] A. Greenbaum, "A multigrid method for multiprocessors," *Applied Mathematics and Computation*, vol. 19, no. 1-4, pp. 75–88, 1986.

[18] T. F. Chan and R. S. Tuminaro, "Design and implementation of parallel multigrid algorithms," *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, pp. 101–115, 1987.

[19] D. Gannon and J. V. Rosendale, "On the structure of parallelism in a highly concurrent PDE solver," *Journal of Parallel and Distributed Computing*, vol. 3, no. 1, pp. 106–135, 1986.

[20] H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline, "Block-asynchronous multigrid smoothers for GPU-accelerated systems," *Proceedings of the International Conference on Computational Science (ICCS)*, vol. 9, pp. 7–16, 2012.

[21] A. AlOnazi, G. S. Markomanolis, and D. Keyes, "Asynchronous task-based parallelization of algebraic multigrid," *Proceedings of the Platform for Advanced Scientific Computing Conference*, no. 5, pp. 1–11, 2017.

[22] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[23] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid smoothers for ultraparallel computing," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2864–2887, 2011.

[24] U. M. Yang, "On the use of relaxation parameters in hybrid smoothers," *Numerical Linear Algebra with Applications*, vol. 11, no. 23, pp. 155–172, 2004.

[25] "MFEM: Modular finite element methods library," mfem.org.

[26] R. Sevilla, "NURBS: Enhanced finite element method (NEFEM)," Ph.D. dissertation, Polytechnic University of Catalonia, 2009.