Parallel Time Integration with Multigrid Reduction for a Compressible Fluid Dynamics Application

R.D. Falgout[†], A. Katz[§], Tz.V. Kolev[†], J.B. Schroder[†‖],
A.M. Wissink[‡], U.M. Yang[†]

October 28, 2014

Abstract

We describe our efforts to speed up the computational fluid dynamics (CFD) application code Strand2D via the parallel multigrid reduction in time (MGRIT) software library XBraid. The need for parallel-in-time approaches, such as MGRIT, is being driven by current trends in computer architectures where performance improvements are coming from greater parallelism, not faster clock speeds. This leads to a bottleneck for sequential time integration methods, such as those used in Strand2D, because they lack parallelism in the time dimension. Thus, the ability to apply parallel-in-time approaches to CFD codes is of interest, and MGRIT is particularly well suited given its non-intrusiveness, which only requires users to wrap existing time stepping codes in the XBraid framework. The contributions of this paper are the description of the nonlinear version of MGRIT and the corresponding software implementation XBraid. We also discuss the steps needed to use XBraid with Strand2D and present the corresponding results for unsteady laminar flow over a cylinder. These results demonstrate a significant speedup when sufficient resources are available.
Keywords: parallel-in-time, multigrid reduction in time, MGRIT, XBraid, compressible Navier-Stokes, Navier-Stokes, strand grids

1 Introduction

Since single-core clock speeds have stagnated, computer engineers are achieving higher speeds by designing high performance computer architectures with significantly increased parallelism with millions of cores. Although most computational fluid dynamics (CFD) codes are able to fully exploit parallelism in the spatial dimensions, this limits how many processors they can effectively exploit. Moreover, stagnant clock speeds imply that the time per time step is also stagnant, if spatial parallelism has already been fully exploited. Thus, for unsteady CFD applications, such as rotorcraft or turbomachinery, where the number of timesteps required to reach a desired state generally grows proportionately with the spatial problem size, future simulation wall clock times will grow. Additionally, it is unclear how these CFD codes will take advantage of massively parallel machines, if they only support parallelism in space. Thus for such applications it is advantageous to find additional parallelism in time to maximize performance on these massively parallel architectures.

The question then arises if it is possible to parallelize the time dimension. There have been various efforts to develop such methods which date back to as far as 1964 [25, 13, 8, 32, 16, 14, 12, 3, 5, 10]. One of the currently better known algorithms that parallelizes in time is parareal [20], which has been extensively investigated [23, 22, 24, 28, 6]. It can be interpreted as a two-level multigrid method [11]. However, since

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551. 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-663416)

[‡]U.S. Army Aviation Development Directorate - AFDD, Aviation Missile Research Development and Engineering Command (AMRDEC), Moffett Field, CA 94035-1000

[§]Assistant Professor, Utah State University, Dept. of Mechanical and Aerospace Engineering, Logan, Utah 84322

[‖]Corresponding author: schroder2@llnl.gov

the coarse level solve is sequential, use of parallelism is limited to the finest level. It is therefore of interest to investigate multilevel algorithms in time. There are various multigrid methods which parallelize in both space and time [15, 29, 33]. However, using such methods in an existing application code that is generally already highly parallel requires major rewriting of often extremely complex codes and is therefore not desirable.

In this work, we explore a multigrid reduction in time (MGRIT) [9] approach that parallelizes in the time dimension and is non-intrusive. We describe for the first time the nonlinear version of MGRIT and its corresponding software implementation XBraid [1]. The XBraid library reflects the non-intrusiveness of MGRIT and enables users to apply the approach to their existing application code by writing only a small amount of code. We demonstrate this non-intrusiveness by applying XBraid to the Strand2D CFD code [19], a compressible Reynolds-Averaged Navier-Stokes (RANS) code that operates on 2-dimensional strand grids. Last, we present results for unsteady flow over a cylinder that show a significant wall clock speedup over the original code when sufficient parallel processors are available.

The paper is organized as follows. Section 2 describes the multigrid in time algorithm. Section 3 gives an overview of the Strand2D code. Section 4 discusses how to apply XBraid to Strand2D. Numerical results are described in Section 5, and we conclude in Section 6.

2 Multigrid in Time

Consider a system of ordinary differential equations of the form

$$u'(t) = f(t, u(t)), \quad u(0) = g_0, \quad t \in [0, T]. \tag{1}$$

Let $t_i = i\delta t, i = 0, 1, ..., N$ be a temporal mesh with spacing $\delta t = T/N$, and u_i be an approximation to $u(t_i)$. A general one-step time discretization is now given by

$$u_0 = g_0 \tag{2}$$
$$u_i = \Phi_i(u_{i-1}) + g_i, \quad i = 1, 2, ..., N. \tag{3}$$

A traditional time stepping scheme solves this system sequentially by first solving for $i = 1$, followed by $i = 2$ and so on. For linear time propagators $\{\Phi_i\}$, this can also be expressed as applying a direct solver (a forward solve) to the following system:

$$A(\mathbf{u}) = \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_N \end{pmatrix} = \mathbf{g}. \tag{4}$$

This $O(N)$ process is optimal, but sequential. We achieve parallelism in time by replacing the sequential solve with an optimal multigrid iterative method. Our particular approach is called MGRIT [9] and is based on applying multigrid reduction (MGR) [21] in time. The method coarsens in the time dimension with factor $m > 1$ to yield a coarse (time) grid with $N_\Delta = N/m$ points and time step $\Delta T = m\delta t$. The corresponding coarse grid problem,

$$A_\Delta(\mathbf{u}_\Delta) = \begin{pmatrix} I & & & \\ -\Phi_{\Delta,1} & I & & \\ & \ddots & \ddots & \\ & & -\Phi_{\Delta,N_\Delta} & I \end{pmatrix} \begin{pmatrix} u_{\Delta,0} \\ u_{\Delta,1} \\ \vdots \\ u_{\Delta,N_\Delta} \end{pmatrix} = \begin{pmatrix} g_{\Delta,0} \\ g_{\Delta,1} \\ \vdots \\ g_{\Delta,N_\Delta} \end{pmatrix} = \mathbf{g}_\Delta, \tag{5}$$

is obtained by defining coarse grid propagators $\{\Phi_{\Delta,i}\}$ which are at least as cheap to apply as the fine scale propagators $\{\Phi_i\}$. The coarse time grid induces a partition of the fine grid into *C-points* (associated with coarse grid points) and *F-points*, as visualized in Figure 1.

Every multigrid algorithm requires a relaxation method and an approach to transfer values between grids. Our relaxation scheme alternates between so-called F-relaxation and C-relaxation as illustrated

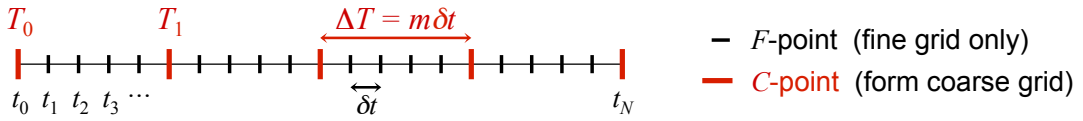

Figure 1: Fine and coarse time grids with a coarsening factor m

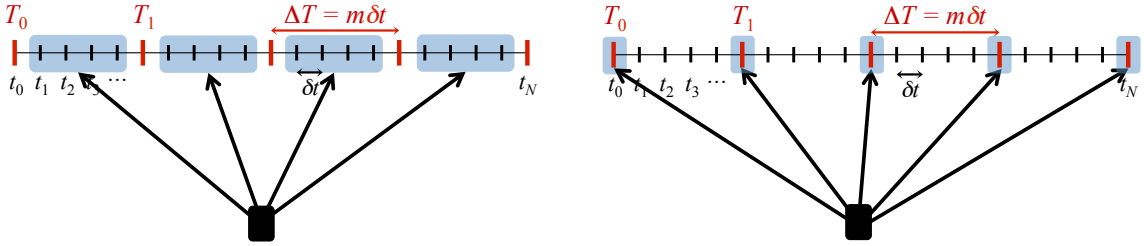

Figure 2: F-relaxation (left) updates all F-point intervals in parallel from the current values at C-points. C-relaxation (right) updates all C-points from the neighboring F-point values.

in Figure 2. F-relaxation updates the F-point values $\{u_j\}$ on interval (T_i, T_{i+1}) by simply propagating the C-point value u_{mi} across the interval using the time propagators $\{\Phi_j\}$ in sequence. While this is a sequential process, each F-point interval update is independent from the others and can be computed in parallel. Similarly, C-relaxation updates the C-point value u_{mi} based on the F-point value u_{mi-1} and these updates can also be computed in parallel. Of particular interest is FCF-relaxation, defined as F-relaxation followed by C-relaxation followed by F-relaxation. This approach to relaxation can be thought of as line relaxation in space. Simple injection is used to transfer values between grids. Note that our combination of injection and F-relaxation together can also be interpreted as harmonic interpolation [9], but it is easier to implement as an injection algorithm.

In general, f is a nonlinear function, so we apply the Full Approximation Storage (FAS) method [4, 30], which is a nonlinear version of multigrid. The two-grid FAS algorithm proceeds as follows:

1) Apply FCF-relaxation to $A(\mathbf{u}) = \mathbf{g}$.

2) Restrict the fine grid approximation and its residual to the coarse grid

$$u_{\Delta,i} \leftarrow u_{mi}, \quad r_{\Delta,i} \leftarrow g_{mi} - A(\mathbf{u})_{mi} \quad \text{for} \quad i = 0, ..., N_\Delta.$$

3) Solve $A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$.

4) Compute the coarse grid error approximation: $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$.

5) Add the error to the values of $\mathbf{u}$ at the C-points: $u_{mi} = u_{mi} + e_{\Delta,i}$.

6) Correct $\mathbf{u}$ by applying an F-relaxation sweep.

The method can be turned into a multilevel algorithm by applying this procedure recursively to the system in Step 3. A variety of standard multigrid cycling strategies may be applied, including V-cycles and F-cycles as illustrated in Figure 3. We note that this is the first description of the nonlinear version of MGRIT.

A critical facet of this multigrid reduction method is that the user need only define Φ_i, i.e., the method is fairly nonintrusive. The user can continue to use an existing time stepping code, provided it is wrapped appropriately. See Section 4 for a description of how this is accomplished in our MGRIT library XBraid. More detailed information on the MGRIT algorithm can be found in [9].

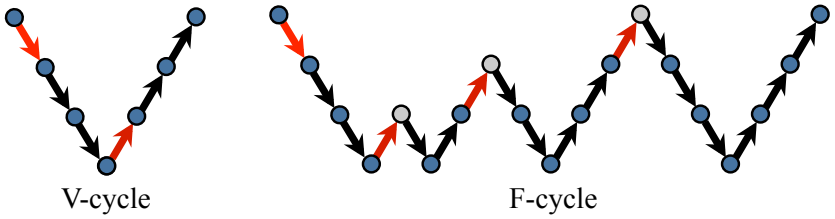

Figure 3: Illustration of a V- and an F-cycle.

3 Strand2D

Strand2D solves the unsteady Reynolds-averaged Navier-Stokes (RANS) equations in three dimensions. Turbulence closure is accomplished with the Spalart-Allmaras (SA) model[27]. The RANS-SA equations may be expressed as

$$\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} - \frac{\partial F_j^v}{\partial x_j} = S, \tag{6}$$

where the conserved variables, Q, inviscid fluxes, F_j, viscous fluxes, F_j^v, and source term, S, are defined as

$$Q = \begin{pmatrix} \rho \\ \rho u_i \\ \rho e \\ \rho\tilde{\nu} \end{pmatrix}, \quad F_j = \begin{pmatrix} \rho u_j \\ \rho u_i u_j + p\delta_{ij} \\ \rho h u_j \\ \rho\tilde{\nu} u_j \end{pmatrix}, \quad F_j^v = \begin{pmatrix} 0 \\ \sigma_{ij} \\ \sigma_{ij} u_i - q_j \\ \frac{\eta}{\sigma}\frac{\partial\tilde{\nu}}{\partial x_j} \end{pmatrix}, \quad S = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \mathcal{P} - \mathcal{D} + C_{b2}\rho\frac{\partial\tilde{\nu}}{\partial x_k}\frac{\partial\tilde{\nu}}{\partial x_k} \end{pmatrix}. \tag{7}$$

Here, ρ is the density, u_i is the Cartesian velocity vector, e is the total energy per unit mass, $\tilde{\nu}$ is the turbulence working variable, p is the pressure, h is the total enthalpy per unit mass, σ_{ij} is the deviatoric stress tensor, q_j is the heat flux vector, and η/σ is the turbulent diffusion coefficient. The turbulent source term consists of a production term, $\mathcal{P}$, and a destruction term $\mathcal{D}$. The stress tensor is defined as

$$\sigma_{ij} = 2(\mu + \mu_T)s_{ij}, \tag{8}$$

where μ is the dynamic viscosity, μ_T is the turbulent viscosity, and s_{ij} is the rate of strain tensor, defined as

$$s_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{1}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}. \tag{9}$$

The heat flux vector is obtained with Fourier's Law,

$$q_j = -C_p\left(\frac{\mu}{Pr} + \frac{\mu_T}{Pr_T}\right)\frac{\partial T}{\partial x_j}, \tag{10}$$

where C_p is the specific heat, Pr is the Prandtl number, Pr_T is the turbulent Prandtl number, and T is the temperature. The ideal gas equation of state, $p = \rho RT$ is used to close the equations.

The spatial discretization is based on a cell-centered approach where the primary unknowns are located at the centroid of the prisms formed by adjacent strands. The solver accommodates both quadrilateral and triangular prisms depending on the surface topology. However, control volumes are composed entirely of triangular facets by triangulating any non-planar quadrilateral faces. This is important for second-order accuracy on general prismatic grids with no assumption of underlying smoothness [17]. Linear reconstruction is employed to obtain second-order accuracy through first obtaining consistent nodal values of the conserved variables from surrounding cell-center values. A projection method is used to obtain these nodal values via least squares interpolation in a regression plane through the three-dimensional stencil of cells surrounding a strand [17]. Once the nodal values have been obtained, a Green-Gauss surface integration procedure is performed to obtain cell gradients in each control volume.

Inviscid fluxes rely on a reconstruction upwind formula for the numerical flux based on the approximate Riemann solver of Roe [26],

$$\hat{\mathcal{F}} = \frac{1}{2}\left(\mathcal{F}(Q_R) + \mathcal{F}(Q_L)\right) - \frac{1}{2}\left|A(Q_R, Q_L)\right|\left(Q_R - Q_L\right), \tag{11}$$

where $\mathcal{F} = F_j n_j$ is the directed flux at a face with normal n_j, and $A = \partial\mathcal{F}/\partial Q$ is the directed flux Jacobian. The viscous terms are computed using values of Q and ∇Q determined at each face,

$$\mathcal{F}^v = \mathcal{F}^v(Q_f, \nabla Q_f), \tag{12}$$

where f refers to the face reconstructed values. These face values are easily obtained once nodal values have been reconstructed using the projection method described above. This method is similar to the node averaging schemes outlined by Diskin, et al. [7]. Both the inviscid and viscous discretization methods described herein have been verified to be second-order accurate for arbitrary prismatic meshes under a variety of conditions using the method of manufactured solutions [17].

The standard SA model is used when the turbulent working variable is positive. Details of the positive model, including the well-known definitions of the production and destruction terms, may be found in the original work by Spalart and Allmaras [27]. Modifications to the model to accommodate negative values of the turbulence working variable have been suggested recently by Allmaras [2] and are employed in this work. In the case of negative values of $\tilde{\nu}$, the following turbulence equation replaces the standard model:

$$\frac{\partial\tilde{\nu}}{\partial t} + u_j\frac{\partial\tilde{\nu}}{\partial x_j} = C_{b1}(1 - C_{t3})\Omega\tilde{\nu} + C_{w1}\left(\frac{\tilde{\nu}}{d}\right)^2 + \frac{1}{\sigma}\left[\frac{\partial}{\partial x_j}\left((\nu + \tilde{\nu}f_n)\frac{\partial\tilde{\nu}}{\partial x_j}\right) + C_{b2}\frac{\partial\tilde{\nu}}{\partial x_k}\frac{\partial\tilde{\nu}}{\partial x_k}\right], \tag{13}$$

where,

$$f_n = \frac{C_{n1} + \chi^3}{C_{n1} - \chi^3}, \quad (C_{n1} = 16).$$

Here, Ω is the vorticity magnitude, d is the distance to the nearest wall, and $\chi = \tilde{\nu}/\nu$ is the ratio of the turbulent working variable to the kinematic viscosity of the fluid. All other constants in Equation 13 take the values found in the standard model.

The result of the spatial discretization of the viscous and inviscid fluxes is a coupled set of non-linear equations. In this work, we adopt a pseudo-time framework to march the steady or unsteady discretized equations to steady-state,

$$V\frac{\partial Q}{\partial\tau_k} + R(Q) = 0. \tag{14}$$

Here, V is the cell volume, and τ_k is the pseudo-time variable. The residual, $R(Q)$, contains the inviscid and viscous flux balances at each cell based on the cell-center discretization schemes described above. In order to reach a pseudo-steady state using an implicit scheme, the residual is linearized, leading to the following linear system to be solved at each pseudo-time step:

$$\left[\frac{V}{\Delta\tau_k}I + \frac{\partial R^k}{\partial Q}\right]\left(Q^{k+1} - Q^k\right) = -R(Q^k). \tag{15}$$

Here, $\partial R^k/\partial Q$ is the Jacobian of the residual. The linear system in Equation 14 in general is large and sparse, rendering direct inversion impractical. Iterative line Gauss-Seidel (GS) methods are employed to solve this system, where contributions along strands are collected to form a tridiagonal system. To facilitate the line GS iterations and to increase robustness, we introduce an additional "linear time" variable, τ_l,

$$V\frac{\partial Q}{\partial\tau_l} + \left[\frac{V}{\Delta\tau_k}I + \frac{\partial R^k}{\partial Q}\right]\left(Q^{k+1} - Q^k\right) = -R(Q^k). \tag{16}$$

The linear time is introduced to improve the diagonal dominance of the line GS procedure in order to increase robustness. Rearranging Equation 16 in terms of solution updates in linear time results in

$$\left[\left(\frac{1}{\Delta\tau_k} + \frac{1}{\Delta\tau_l}\right)VI + \frac{\partial R^k}{\partial Q}\right]\left(Q^{l+1} - Q^l\right) = -R(Q^k) - \left[\frac{V}{\Delta\tau_k}I + \frac{\partial R^k}{\partial Q}\right]\left(Q^l - Q^k\right). \tag{17}$$

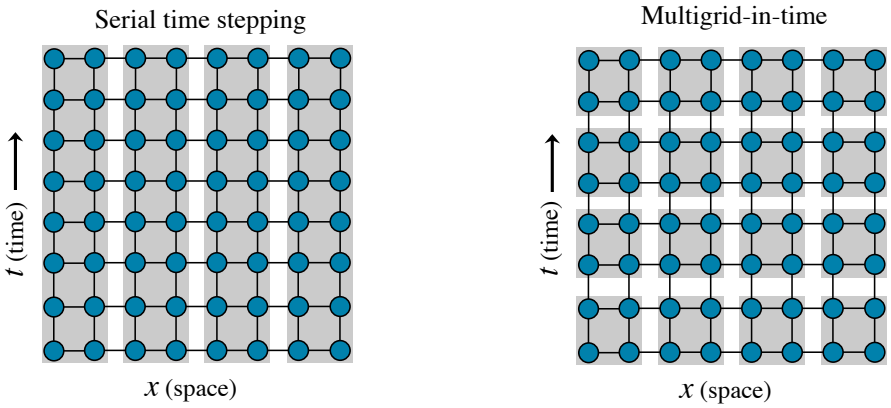

Figure 4: Parallel decomposition using sequential time stepping (left) and parallelizing in time also (right).

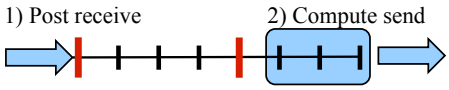

Figure 5: Overlapping communication and computation.

Upon convergence of the linear iterations in l, the linear system of Equation 15 is satisfied. At that point, the next pseudo-time step in k proceeds. When the pseudo-time iterations converge, then the residual equation $R(Q)$ is satisfied for a given physical time station. All Jacobian terms in this work are first order and retain only nearest neighbor contributions. Further details of the implicit solution method may be found in earlier references[18].

4 Use of XBraid with Strand2D

The XBraid library is an implementation of the algorithm described in Section 2. It only parallelizes in time and leaves any spatial parallelization to the user. Figure 4 shows the parallel decomposition of a sequential time stepping algorithm on the left. Here one proceeds sequentially in time and only one time step is stored. On the right is the decomposition for multigrid in time. Parallelism is significantly increased, but now several time steps need to be stored, requiring more memory. Multigrid in time is more expensive than sequential time stepping in terms of increased memory cost as well as total operation count. Therefore it is expected that the method will be slower if only a few processors are available. However due to its increased parallelism, when more processors are used, it is able to utilize them to the fullest and eventually reaches a crossover point, beyond which ever larger speedups are realized. We note that the description of XBraid is a specific contribution of this paper.

XBraid employs two strategies to address the increased memory costs. First, one need not solve the whole problem at once. Instead, the problem can be divided it into several time slabs that would fit in the available memory. Second, XBraid only stores the *C-points* (see Figure 1). Since coarsening in time is typically aggressive (e.g., a factor of 5, 10 or 20), this greatly reduces the number of time points in storage.

In the implementation of XBraid, efforts have been made to overlap communication and computation. The main computational kernel of XBraid is one relaxation sweep touching all the CF intervals. At the start of a relaxation sweep if a process owns several C-points including their intervals, it first posts a non-blocking receive on its left-most C-point. It then carries out F-relaxation in each interval, starting with the right-most interval to send the data to the neighboring process as soon as possible. This is illustrated in Figure 5.

As mentioned in Section 2, XBraid is a non-intrusive code with a flexible framework, i.e. users write simple wrappers for their time stepping code to provide XBraid with the time stepping routine Φ_i. The

user needs to define two data structures: `Vector` and `App`. `Vector` defines a state vector at a certain time value, but can also contain additional information related to the vector, such as mesh information that is needed to evolve the vector to the next time step. The strand2D wrapper for XBraid's Vector object is given as follows:

```
class BraidVector
{
public:
   // Empty Constructor
   BraidVector() {  }

   // State vector
   Array3D<double> vec;
};
```

`App` is available to every wrapper function, either as a parameter in C, or as the parent object of the wrapper function in C++. `App` needs to hold everything the user needs to evaluate a time step. In the case of Strand2D it can be defined as follows, where `StrandBraidApp` inherits from the provided abstract base class `BraidApp`.

```
class StrandBraidApp : public BraidApp
{
public:
   // Abstract base class BraidApp already defines
   //       tstart                          global start time
   //       tstop                           global stop time
   //       ntime                           number of time steps
   //       comm_t                          temporal communicator
   string   in_file;                    // File holding Strand2D input values
   double   dt;                         // Finest-level time step size
   int      noutput;                    // Print solution every noutput steps
   int      buff_size;                  // Size in bytes of the send/recv xbraid buffer
   int      state_vec_size;             // Size in bytes of the state vector
   int      nSurfNode, nStrandNode, nq; // Dimensions of the Strand2D state vector
   BraidVector *q_init;                 // Initial condition for state vector
   Strand2dFCManager *manager;          // Strand2D manager drives simulation
                                        // and takes time steps
};
```

The user needs to also write several wrapper routines, which are member functions of the StrandBraidApp class. The core user routine `Phi` defines how to advance the vector $\mathbf{u}$ from time `tstart` to `tstop`. Using the terminology of Chapter 2 it needs to define how to generate u_i from $\Phi_i(u_{i-1})$. Generally this code already exists in an application and needs to only be wrapped. For Strand2D, `Phi` can be defined as follows. Note that the status information about current time values is available through `pstatus` and that because `Phi` is a member function of `StrandBraidApp`, it has access to all of that object's data members.

```
virtual int Phi(braid_Vector    _u,
                BraidPhiStatus  &pstatus)
{
   // Initialize, using pstatus to obtain state information
   BraidVector *u = (BraidVector*) _u;
   double tstart, tstop;
   pstatus.GetTstartTstop(&tstart, &tstop);
```

```
   double dt = tstop - tstart;
   int step = round(tstop / dt);

   // Reset strand with new time step size and state vector
   manager->resetQ(u->vec);
   manager->resetDtUnsteady(dt);

   // Carry out one time step (via many pseudo steps)
   int  nPseudoSteps  = manager->getNPseudoSteps();
   bool converged     = false;
   for (int pseudoStep=0; pseudoStep<nPseudoSteps; pseudoStep++)
   {
      manager->takePseudoStep(step, pseudoStep, converged);
      if (converged)
         break;
   }

   // Save state vector from Strand2D
   manager->getQ(u->vec);

   // No temporal refinement
   pstatus.SetRFactor(1);

   return 0;
}
```

The user should also define the following additional wrapper routines:

- `Init` defines how to initialize a vector at a specified time.
- `Clone` defines how to clone a vector into a new vector.
- `Free` defines how to free a vector.
- `Sum` defines how to sum two vectors.
- `Dot` defines a dot product of two vectors.
- `Write` defines how to write a vector at time t to a user defined output (screen, file, etc...).
- `Bufsize, BufPack, BufUnpack` define how to pack and unpack an MPI buffer containing one `BraidVector`. No user-written MPI calls are needed.
- `Coarsen, Restrict` are optional routines that allow to additionally define how to coarsen in space while coarsening in time.

Overall, applying XBraid to Strand2D, which has about 13,500 lines, required adding 129 lines to Strand2D, 20 of which were needed to facilitate file output in parallel. The remaining lines were needed to enable restarting of the code at a new time for a new state vector. Many application codes already allow restarting of the codes, and in this case the application of XBraid requires less changes to the code. The XBraid wrapper code required writing about 475 lines, most of which included command line parsing. The core functionality of the wrapper code is much shorter.

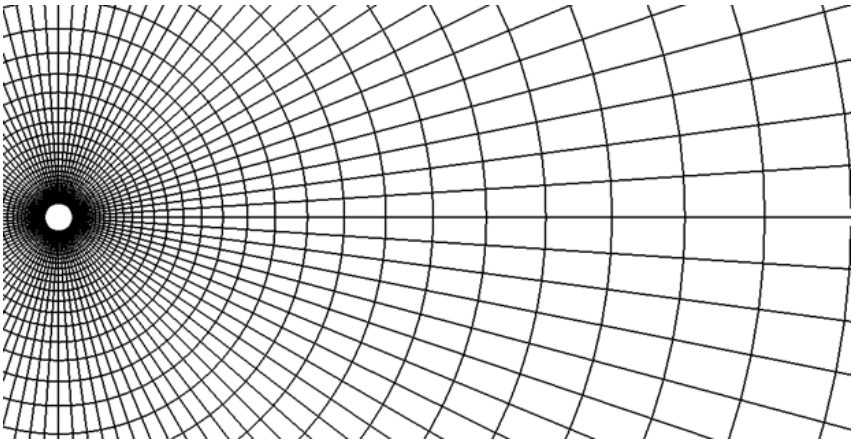

Figure 6: 96×32 strand grid.

Number of time steps	Number of processes	Iterations XBraid	Run Time XBraid	Run Time Serial	Speedup
1280	256	19	58 min	37 min	0.64
5120	1024	15	62 min	163 min	2.63
20480	4096	14	87 min	655 min	7.53

Table 1: Weak scaling study.

5 Numerical Results

The new parallel in time version of Strand2D, enabled by XBraid, is evaluated for computing the unsteady flow over a cylinder at $M = 0.2$ and $Re = 100$. We use an implicit time stepping scheme, backward Euler, and a 3rd order finite differencing scheme on strand grids. The CFD grid used for the calculations, pictured in Figure 6, contains 32 points in the normal direction and 96 points in the circumferential direction. Figure 7 shows snapshots of the velocity at different time steps exhibiting unsteady vortex shedding. The Strouhal number is a measure of the vortex shedding frequency and is used to validate the code accuracy.

In XBraid, we coarsen by a factor 5 and use F-cycles with FCF-relaxation. While we originally used a relative residual stopping criteria of 10^{-8} when solving the system (4), we found that sufficient accuracy could be achieved using a larger factor of 10^{-5}, which also led to shorter run times. The salient information such as the Strouhal number is already well-resolved at this level of accuracy. The maximum coarse time-grid size is 50 time steps. After the coarse time-grid reaches this size, temporal coarsening stops.

To evaluate the convergence of the method, we perform a weak scaling study. The final time is fixed at 2.56s and then refined in time to get problems of varying size. We consider three cases, using 1280, 5120 and 20480 time steps. We choose the 1280 time step case because it barely resolves the unsteady behavior. The corresponding time step sizes are 0.0005s, 0.000125s, and 0.00003125s. Figure 8 gives snapshots for the $\mathbf{u}$ velocity magnitude at the final time step for the 5120 time step case at different XBraid iterations. While during the initial iterations the solution at t_{final} is inaccurate, by iteration 13, the solution has been resolved at t_{final}. Note that XBraid converges to the same solution as the original code Strand2D running standard sequential time-stepping to within the specified tolerance. Consequently, one achieves exactly the same Strouhal numbers that Strand2D would achieve.

Table 1 shows our weak scaling results. The reported speedups are computed by dividing the runtime of the original sequential Strand2D code by the runtime obtained using XBraid wrapped around Strand2D. As previously mentioned for a small problem, sequential time stepping is faster than multigrid in time,

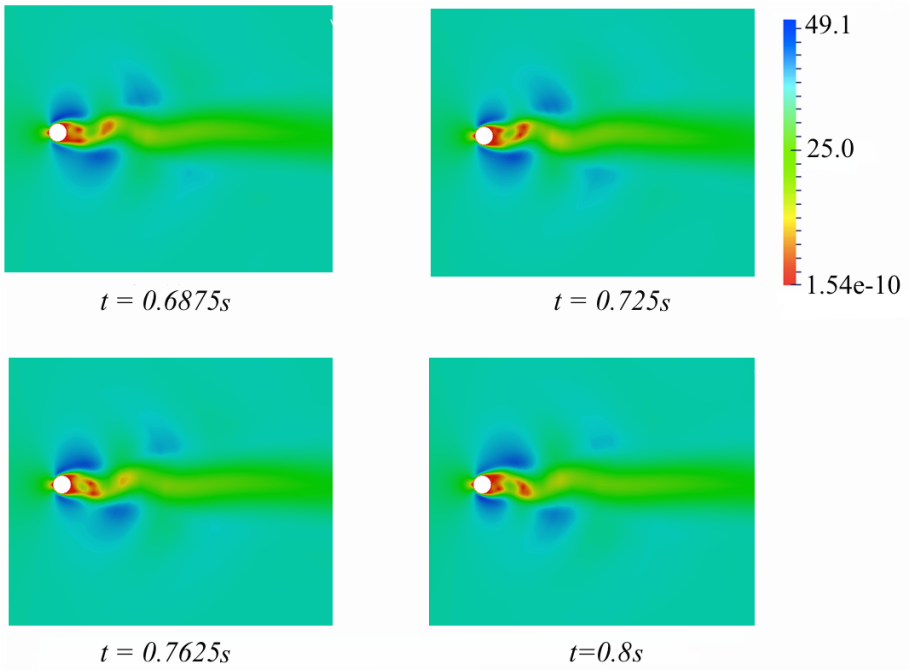

Figure 7: Snapshots of the velocity at various time steps.

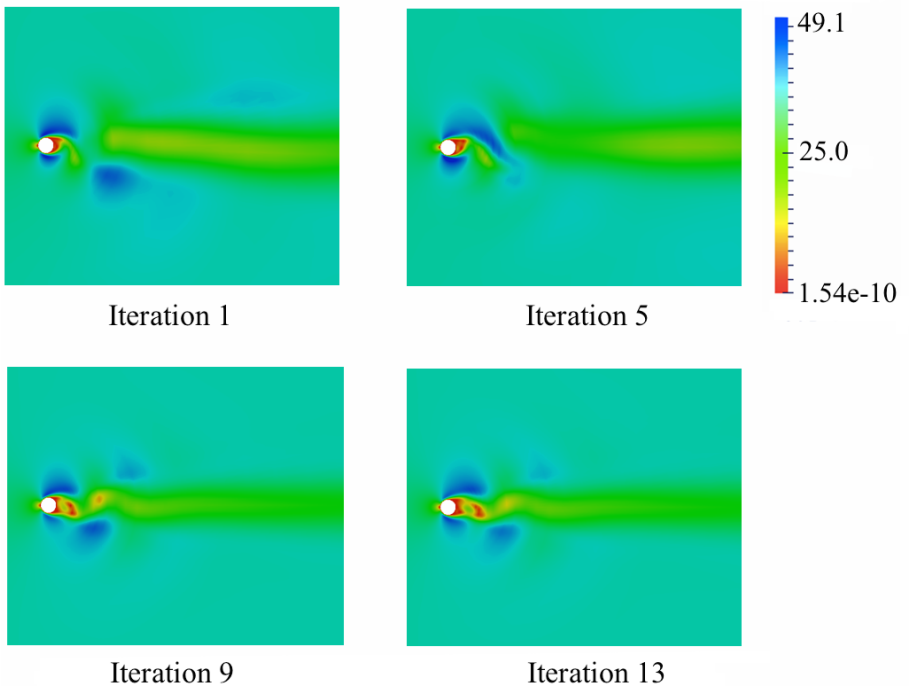

Figure 8: Snapshots of the velocity magnitude from different XBraid iterations at final time.

however already for the second case we achieve a speedup over the sequential code and observe even better speedup for the largest case. While our algorithm requires significantly more resources, it is almost 8 times faster, and can be solved in 87 minutes, a time that would have been impossible to achieve for this problem with the original code.

Since we also want to test the effect of a larger t_{final} on the convergence, we next hold the time step δt constant and quadruple the number of timesteps for some of our test cases, leading to $t_{final} = 10.24s$ instead of $t_{final} = 2.56s$. The first case of 1280 time steps and 19 iterations now takes 58 iterations, and the 5280 time step case of 15 iterations now takes 45 iterations. For a domain that is 4 times larger, we see an increase in iterations by about a factor of 3. In other words, XBraid is able to leverage coarse time grids to converge more quickly than the factor of 4 increase in the time domain would suggest.

The first and second cases have run times of 227 minutes and 268 minutes, respectively. This new runtime of 227 minutes is roughly predicted by the runtime in Table 1 for 5120 time steps, i.e., $(58/15)62 \approx 239$. Similarly, the new runtime of 268 minutes is roughly predicted by $(45/14)87 \approx 279$. This indicates that the time per iteration of XBraid is similar, regardless of t_{final}. This is due to the fact that the underlying Strand2D code has a fixed nonlinear time step cost (25 pseudotime steps are always taken for each nonlinear time step). The speedup for these two cases is 0.71 and 2.38, respectively. It is critical to note that similar speedups to Table 1 can be realized with this longer time domain, but with the requirement that roughly 4 times as many processors be used.

While for this particular problem the chosen t_{final} is sufficient to capture the correct Strouhal number, we nonetheless believe that having a more constant XBraid iteration count, regardless of t_{final} is desirable. However, the problem domain increases with larger t_{final} and the algorithm takes longer to resolve the new physical behavior associated with the larger time domain. Nonetheless, slowing the growth in iterations as t_{final} increases is a topic of ongoing research. In particular, we are interested in leveraging the periodic part of the solution to better inform the initial guess at later time steps.

5.1 Parallel Scaling

For the scaling study, we consider 5 data layouts, 80 time points (tpts) per core, 40 tpts per core, 20 tpts per core, 10 tpts per core and 5 tpts per core, for 5 problem sizes, 1280, 2560, 5120, 10240 and 20480 time steps. This generates 25 data points that can be used for both a weak and a strong scaling study. This is depicted by the log-log plot in Figure 9, where the strong scaling lines are dotted and correspond to a fixed global problem size. The weak scaling lines are solid and correspond to a fixed problem size (in time points or "tpts") per core. Overall, the weak scaling improves with more time points per core (i.e., more computation relative to communication). For the worst case at 5 time points per core, the time to solution increases about 50% from 256 to 4096 cores and for the best case at 80 time points per core, the time to solution actually decreases when going from 16 to 256 cores.

The next log-log plot in Figure 10 depicts the same information, but in terms of speedup relative to sequential single processor runs of Strand2D. The "Perfect Slope" line depicts the slope of perfect weak scaling and this line makes it apparent that our approach scales well in a weak sense. The Strouhal numbers ("St") represent the shedding frequency and are a quantity of interest for this problem. They are shown to argue that the larger problem sizes are useful because the Strouhal number does not show convergence until the second largest problem size and the careful user may even run the largest problem size to be certain that the Strouhal number has converged [1].

The crossover point where XBraid begins to offer a speedup over sequential time stepping is between 64 and 128 cores for both 40 and 80 time points per core. If one estimates the actual crossover point, it is less than 100 cores for the 80 time points per core case. The maximum speedup is a factor of 7.53 at 4096 cores and 5 time points per core. While for small problems, sequential time stepping is faster than multigrid in time, these experiments show that even modest resources on today's machines allow multigrid in time to show a benefit. Additionally, Figure 10 makes it apparent that this approach can scale to fit the availability of resources, allowing for a much greater degree of parallelism. We also note

[1] As mentioned, XBraid coupled to Strand2D solves the same problem, to within a tolerance, that sequential Strand2D solves. Thus, this Strouhal value of 0.141 corresponds to the top-right entry of 0.141 in Table 4 of [19].

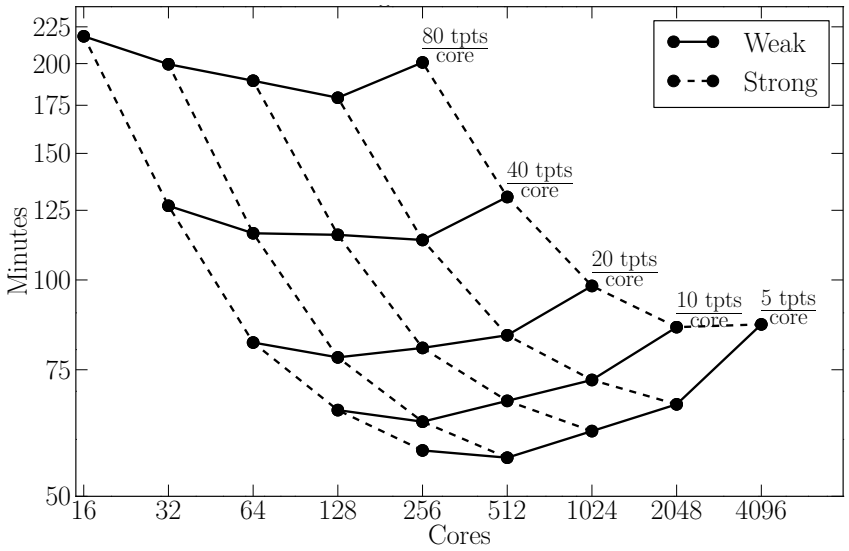

Figure 9: Strong-weak scaling study, showing actual run times of XBraid. The strong scaling lines are dotted and correspond to a fixed global problem size, while the weak scaling lines are solid and correspond to a fixed problem size (in time points or "tpts") per core.

that further optimizations to the code would allow for even better speedups, e.g., constructing coarse spatial meshes for the coarse time-grids.

6 Conclusions and Future Work

Since standard CFD codes only parallelize in the spatial dimensions, the motivation of this work is to investigate both parallelism in the time-domain and how to effectively utilize massively parallel computers for unsteady CFD applications. To this end, we presented for the first time the nonlinear version of the parallel-in-time method MGRIT and the corresponding software implementation XBraid. We also applied MGRIT to an unsteady RANS CFD application by integrating the XBraid software library with the Strand2D unsteady RANS solver [19].

Parallel in time capability was added to Strand2D with relative ease. It was simply plugged into XBraid with approximately 129 new lines added to Strand2D and 475 wrapper lines needed to accomplish the integration (see Section 4 for more details). The integrated code was tested for a model problem to compute the unsteady vortex shedding over a circular cylinder. The original Strand2D code had no parallelism (because it is two-dimensional, there is little parallel performance gain from decomposing in space). However, the MGRIT-enabled version was run up to 4096 processors, with a speedup factor of 7.53 over the original sequential time stepping code. The computed answers are identical to within the specified tolerance.

This work revealed that it is possible to introduce time-parallelism to an existing CFD code, and achieve reasonable parallel speedups with little coding effort. Future work should extend the current effort by applying MGRIT to 3D unsteady production CFD codes [31] which already have a parallel spatial decomposition. The resulting combined time-space parallel decomposition is expected to enable the effective application of such CFD codes to massively parallel computer systems.

Acknowledgements

Material presented in this paper is a product of the CREATE-AV Element of the Computational Research and Engineering for Acquisition Tools and Environments (CREATE) Program sponsored by the U.S.

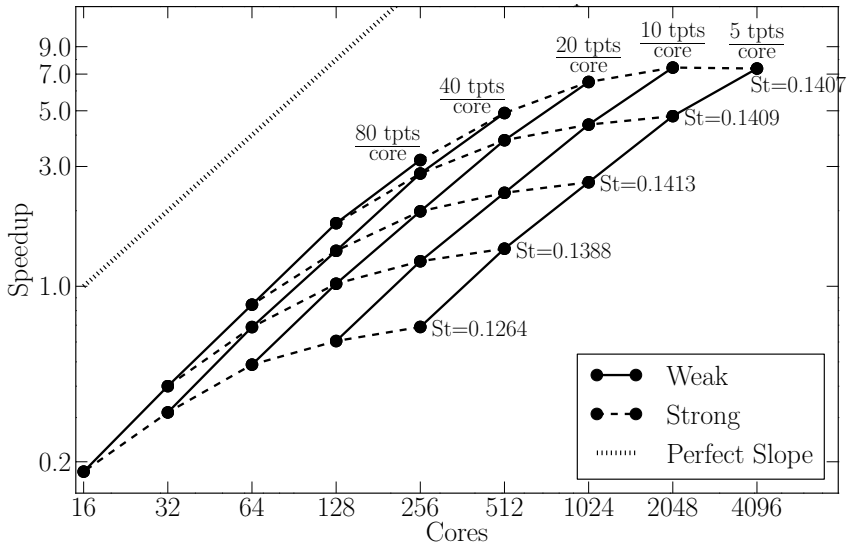

Figure 10: Strong-weak scaling study, showing speedup of XBraid versus serial Strand2D. For each of the five problem sizes, the Strouhal number (St) is printed next to that strong scaling line.

Department of Defense HPC Modernization Program Office.

We thank Christopher Atwood and the Department of Defense High Performance Computing Modernization Program (HPCMP) for their generous support of this work.

References

[1] XBraid: Parallel multigrid in time. `http://llnl.gov/casc/xbraid`. 2

[2] Stephen R. Allmaras, J. T. Forrester, and Phillipe R. Spalart. Modifications and clarifications for the implementation of the spalart-allmaras turbulence model. In *ICCFD-1902, 7th International Conference on Computational Fluid Dynamics (ICCFD7)*, July 2012. 5

[3] Pierluigi Amodio and Luigi Brugnano. Parallel solution in time of ODEs: some achievements and perspectives. *Appl. Numer. Math.*, 59(3–4):424–435, 2009. 1

[4] A. Brandt. Multi-level adaptive computations in fluid dynamics, 1979. Technical Report AIAA-79-1455, AIAA, Williamsburg, VA. 3

[5] Andrew J. Christlieb, Colin B. Macdonald, and Benjamin W. Ong. Parallel high-order integrators. *SIAM J. Sci. Comput.*, 32(2):818–835, 2010. 1

[6] Xiaoying Dai and Yvon Maday. Stable parareal in time method for first- and second-order hyperbolic systems. *SIAM J. Sci. Comput.*, 35(1):A52–A78, 2013. 1

[7] Boris Diskin, Jay Thomas, Eric Nielsen, and H. Nishikawa. Comparison of node-centered and cell-centered unstructured finite-volume discretizations. part 1: Viscous fluxes. In *AIAA-2009-0597, 47th AIAA Aerospace Sciences Meeting*, Jan 2009. 5

[8] D. J. Evans and B. B. Sanugi. A parallel Runge-Kutta integration method. *Parallel Computing*, 11:245–251, 1989. 1

[9] R .D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, and J. B. Schroder. Parallel time integration with multigrid. *SIAM J. Sci. Comput.*, submitted. 2, 3

[10] Martin J. Gander and Stefan Güttel. PARAEXP: A parallel integrator for linear initial-value problems. *SIAM J. Sci. Comput.*, to appear. 1

[11] Martin J. Gander and Stefan Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM J. Sci. Comput.*, 29(2):556–578, 2007. 1

[12] Izaskun Garrido, Barry Lee, Gunnar E. Fladmark, and Magne S. Espedal. Convergent iterative schemes for time parallelization. *Math. Comp.*, 75(255):1403–1428, 2006. 1

[13] C. W. Gear. Parallel methods for ordinary differential equations. *Calcolo*, 25(1–2):1–20, 1988. 1

[14] G. Horton and R. Knirsch. A space-time multigrid method for parabolic partial differential equations. *Parallel Computing*, 18:21–29, 1992. 1

[15] G. Horton and S. Vandewalle. A space-time multigrid method for parabolic partial differential equations. *SIAM J. Sci. Comput.*, 16(4):848–864, 1995. 2

[16] Kenneth R. Jackson. A survey of parallel numerical methods for initial value problems for ordinary differential equations. *IEEE Trans. Magnetics*, 27(5):3792–3797, 1991. 1

[17] Aaron Katz and Venke Sankaran. Discretization methodology for high aspect ratio prismatic grids. 2011. AIAA-2011-3378, 20th Computational Fluid Dynamics Conference, American Institute of Aeronautics and Astronautics. 4, 5

[18] Aaron Katz and Andrew Wissink. Efficient solution methods for strand grid applications. 2012. AIAA-2012-2779, 30th Applied Aerodyanics Conference, American Institute of Aeronautics and Astronautics. 6

[19] Aaron Katz and Dalon Work. High-order flux correction/finite difference schemes for strand grids. 2014. 52nd Aerospace Sciences Meeting, American Institute of Aeronautics and Astronautics. 2, 11, 12

[20] J. L. Lions, Y. Maday, and G. Turinici. Résolution d'EDP par un schéma en temps pararéel. *C.R.Acad Sci. Paris Sér. I Math*, 332:661–668, 2001. 1

[21] U. Trottenberg M. Ries and G. Winter. A note on MGR methods. *Linear Algebra Appl.*, 49:1–26, 1983. 2

[22] Yvon Maday. The "parareal in time" algorithm. In F. Magoulès, editor, *Sub-Structuring Techniques and Domain Decomposition Methods*, Computational Science, Engineering & Technology, chapter 2, pages 19–44. Saxe-Coburg Publications, Stirlingshire, UK, 2010. 1

[23] M. L. Minion and S. A. Williams. Parareal and spectral deferred corrections. In T. E. Simos, editor, *Numerical Analysis and Applied Mathematics*, number 1048 in AIP Conference Proceedings, pages 388–391. AIP, 2008. 1

[24] Michael L. Minion. A hybrid parareal spectral deferred corrections method. *Comm. App. Math. and Comp. Sci.*, 5(2):265–301, 2010. 1

[25] J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Comm. ACM*, 7:731–733, 1964. 1

[26] Philip L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. J. Comput. Phys., 43:357–372, 1981. 5

[27] Philipe R. Spalart and Stephen R. Allmaras. A one-equation turbulence model for aerodynamic flows. *Le Recherche Aerospátiale*, 1:5–21, 1994. 4, 5

[28] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon. A massively space-time parallel N-body solver. 2012. Proc. of the SC12 International Conference for High Performance Computing, Networking, Storage and Analysis. 1

[29] H. De Sterck, Thomas A. Manteuffel, Stephen F. McCormick, and Luke Olson. Least-squares finite element methods and algebraic multigrid solvers for linear hyperbolic PDEs. *SIAM J. Sci. Comput.*, 26(1):31–54, 2004. 2

[30] V. E. Henson W. L. Briggs and S. F. McCormick. *A Multigrid Tutorial*. SIAM, 2000. 3

[31] Andrew Wissink, Buvana Jayaraman, Anubhav Datta, Jayanarayanan Sitaraman, Mark Potsdam, Sean Kamkar, Dimitri Mavriplis, Zhi Yang, Rohit Jain, Joon Lim, and Roger Strawn. Capability enhancements in version 3 of the helios high-fidelity rotorcraft simulation code. In *AIAA-2012-0713, 50th AIAA Aerospace Sciences Meeting*, Jan 2012. 12

[32] David E. Womble. A time-stepping algorithm for parallel computers. *SIAM J. Stat. Comput.*, 11(5):824–837, 1990. 1

[33] Irad Yavneh, Cornelis H. Venner, and Achi Brandt. Fast multigrid solution of the advection problem with closed characteristics. *SIAM J. Sci. Comput.*, 19(1):111–125, 1998. 2