A Parallel Multigrid Algorithm for Solving Elliptic Equations in the Cardiac Bidomain Model

CSD-TR090019

R. Samade, K.R. Raghavan, and B. Kogan

December 2009
A Parallel Multigrid Algorithm for Solving Elliptic Equations in the Cardiac Bidomain Model

R. Samade†, K.R. Raghavan‡, and B. Kogan†
Departments of †Bioengineering and ‡Computer Science,
University of California, Los Angeles, Los Angeles, CA 90095

Abstract

Elliptic partial differential equations (PDEs) are ubiquitous in problems concerned with modeling steady state field distributions in different media. In biology and medicine, the rapid numerical solution of elliptic PDEs are necessary for the bidomain model of myocardium and in the study of phenomena associated with external shock applications to the heart (defibrillation therapy). Here, we summarize the development and study of a novel parallel multigrid algorithm, adapted from a parallel multigrid solver from the hypre numerical library, for subsequent use in studying wave propagation in the two-dimensional (2D) bidomain model of the myocardium (with and without the presence of external defibrillation shocks). Our simulation study, using this novel multigrid algorithm and implemented on a massively parallel supercomputer, shows: (1) the computation time for the bidomain model fell by a factor of 10 (in comparison to the sequential multigrid implementation), while using 64 processors, without introducing significant error in the solution; (2) as the number of processors increased beyond the linear range of scalability, the computation time of the bidomain model continued to decrease, but only asymptotically; and (3) virtually identical distributions of $V_m$ during 2D spiral wave propagations were produced by our original sequential implementation of the multigrid algorithm and our novel algorithm. The significant time savings realized by our novel algorithm will make possible future computer simulation studies of three-dimensional myocardium using the bidomain model, with particular emphases on defibrillation-associated phenomena.

Introduction

Elliptic partial differential equations (PDEs) are frequently employed in the modeling physical field distributions for a variety of media. Inclusion of boundary conditions and source functions into the formulation for an elliptic PDE provides a closed-form expression that is subsequently solvable. However, most elliptic PDEs cannot be solved using an analytical method, which necessitates a numerical integration method. This is accomplished by discretizing the continuous formulation of the elliptic PDE to a set of algebraic equations, by using an approximation such as the finite difference approximation. Each equation in the set corresponds to a distinct point within a particular medium and the entire system of equations is numerical integrated with digital computers in order to determine the corresponding solutions for each of these points. The numerical method employed can belong to one of two classes of algorithms: direct methods, such as Gaussian elimination; or iterative methods, such as conjugate gradient or multigrid methods [1-4]. Owing to the lower number of overall floating point operations of the
multigrid methods, we elected to utilize them as our numerical method for integrating elliptic PDEs.

Within the fields of biology and medicine, two major scientific problems exist that represent physiological processes with elliptic PDEs and require rapid solutions to these equations: the first consists of proper modeling of electrical excitation and propagation in the myocardium with the bidomain model, which is described and presented in its general form in earlier publications [5,6]; and the second entails the modeling of phenomena during and after defibrillation therapy, when an electrical shock is applied to the external domain of the myocardium. Our previous investigation, which employed multigrid algorithms for studying the above-mentioned biological and medical problems [7], demonstrated that a sequential multigrid algorithm does not provide a sufficient decrease in computation time that would permit long timescale simulations with two-dimensional (2D) myocardium and studies using three-dimensional (3D) myocardium.

The absence of an analytical solution for the bidomain model, combined with the computational complexity of modeling the reaction and diffusion processes required in our grid representation of the 2D myocardium (often having dimension of 257 x 257 = 66,409 nodes) necessitates the use of massively parallel supercomputers. Previously, the operator splitting algorithm [8] has been utilized for the parallelization of the solution of reaction-diffusion equations. Effectively parallelizing the numerical integration of elliptic PDEs, however, has represented an important computational challenge.

In this report, we studied the possibilities for parallelizing multigrid algorithms for the purpose of determining the spatial distributions of electrical potential in the external domain of the myocardium (with and without the presence of external defibrillation shocks). We accomplished this task by adapting the parallelized multigrid solver, described in the pfmg class of functions from the hypre numerical library [9], and we obtained a 10 fold decrease in computation time in comparison to a sequential numerical integration of the elliptic PDE. In addition, the fully parallel solution of our mathematical model gives nearly identical results to those obtained in the sequential implementation.

Methods

Mathematical Model

As a simplification, we assumed in this study that the size of the simulated myocardium was relatively small (3.2 cm by 3.2 cm), which permits us to neglect fiber curvature and instead represent the fibers as being uniformly parallel. This, in turn, allows the generalized form of the bidomain equations to be reduced to the following set of partial differential equations (PDEs):

\[
\frac{\partial V_m}{\partial t} = D_{i,x} \frac{\partial^2 V_m}{\partial x^2} + D_{i,y} \frac{\partial^2 V_m}{\partial y^2} + D_{i,x} \frac{\partial^2 \Phi_e}{\partial x^2} + D_{i,y} \frac{\partial^2 \Phi_e}{\partial y^2} - \frac{I_{ion} + I_{shock}}{C_m} \tag{1}
\]

\[
\frac{\partial^2 \Phi_e}{\partial x^2} + \frac{\partial^2 \Phi_e}{\partial y^2} = \beta_1 \frac{\partial^2 V_m}{\partial x^2} + \beta_2 \frac{\partial^2 V_m}{\partial y^2} + \beta_3 i_{shock} \tag{2}
\]

Here, \( V_m \) represents the transmembrane potential (mV), \( \Phi_e \) is the extracellular potential (mV), \( t \) is the time (ms), \( x \) is the displacement along the longitudinal direction of the cardiomyocytes (cm), \( y \) is the displacement along the transverse direction of the
cardiomyocytes (cm), $D_{lx}$ is the diffusion coefficient along the longitudinal (x) axis (cm$^2$/ms), $D_{ly}$ is the diffusion coefficient along the transverse (y) axis (cm$^2$/ms), $I_{ion}$ is the sum of the transmembrane ionic currents (µA/cm$^2$), $I_{stim}$ is an applied transmembrane stimulus current (µA/cm$^2$), $i_{shock}$ is an external defibrillation shock current that is applied to the extracellular domain (µA/cm$^2$), and the parameters $\epsilon = \frac{(g_{e,y} + g_{i,y})}{(g_{e,x} + g_{i,x})}$, $\beta_1 = -\frac{g_{i,x}}{g_{e,x} + g_{i,x}}$, $\beta_2 = -\frac{g_{i,y}}{g_{e,x} + g_{i,x}}$, and $\beta_3 = \frac{1}{g_{e,x} + g_{i,x}}$ are derived from the four conductivities present in the bidomain model. Two of these conductivities, $g_{i,x}$ and $g_{i,y}$, represent the longitudinal and transverse conductivities in the intracellular domain and the other two conductivities, $g_{e,x}$ and $g_{e,y}$, represent the longitudinal and transverse conductivities in the extracellular domain. Values for the four conductivities are identical to those used in a previous investigation [10].

The neglect of fiber curvature, however, introduces a complication in the modeling of the defibrillation shock in that the distribution of the shock current must ensure that the concomitant spatial distribution of $\Phi_e$ is non-linear. Otherwise, the contribution of the terms in Eq. (1) that are derived from the Laplacian of $\Phi_e$ will have no effect on the spatial distribution of $V_m$. For the sake of simplicity, we elected to rectify this problem by modeling the shock current as having a simple one-dimensional (1D) distribution in space, as done previously [7]: $i_{shock} = f(x) = -ax - b$. Here, $a = -2b/L$ (µA/cm$^4$) and $b$ (µA/cm$^2$) are constants that determine the gradient of $\Phi_e$ only during the shock application and $L = 3.2$ cm is the size of the tissue along the x-direction.

A system of nonlinear ordinary differential equations is also necessary in order to describe all of the transmembrane ionic current components of $I_{ion}$ and relevant intracellular trafficking processes so that Eq. 1 is a closed-form expression. We accomplished this by using the Luo and Rudy [11] ionic model of action potential (AP) in the guinea pig ventricular cardiomyocytes, with subsequent modifications to intracellular calcium (Ca$_i$) dynamics by Chudin and associates [12]. Initial conditions and parameters for the AP model variables are the same as those utilized by Chudin [12]. In addition, the space step in the myocardium is set at $h = 0.0125$ cm.

**Numerical Methods and Computer Simulation**

For our simulations, we employed a Cray XT4 parallel supercomputer that is situated in the National Energy Research Scientific Computing Center in the Lawrence Berkeley National Laboratory. The computer code for the mathematical model is written in the C++ programming language and uses Message Passing Interface in order to administer interprocessor communication.

Numerical integration of Eq. (1) on the parallel supercomputer is implemented with an operator splitting algorithm [13,14], which has been previously used for solving reaction-diffusion PDEs in the monodomain model [15,16]. In this algorithm, the diffusion portion of Eq. (1), which contains the terms derived from the Laplacian of $V_m$ and $\Phi_e$, is solved for each integration time step $\Delta t = 0.1$ ms. The reaction portion, which contains only the transmembrane current terms, is solved either once or twenty times per $\Delta t$ by utilizing a variable time step $\Delta t_a$. Additional details of this algorithm can be found elsewhere [13,14].

Both the full multigrid (FMG) algorithm and what is termed the conventional multigrid (CMG) algorithm, in this paper, utilize successive numbers of V-cycles: these V-cycles consist of restricting from the original grid size to smaller grid sizes, using a
relaxation method such as Gauss-Seidel to refine the restricted values, computing the
direct solution at the coarsest grid level, and then interpolating the solution to
progressively finer grids until the original grid has been reached. However, the full
multigrid algorithm and the conventional multigrid algorithm differ in how V-cycles are
structured, which defines the two approaches. In the full multigrid algorithm, iterations of
V-cycles towards progressively higher grid levels are employed, with smoothing,
residual, restriction, correction, and interpolation computations on each sub-grid level.
This is distinguished from a conventional multigrid algorithm, which utilizes V-cycles
that restrict directly from the finest to coarsest grid and interpolate back from the coarsest
grid to the finest grid. A juxtaposition of these two types of V-cycle structures for the full
and conventional multigrid algorithms is summarized in Figure 1.

![Figure 1: V-cycle structures of the conventional multigrid algorithm (left) and the full multigrid algorithm (right) for solving an elliptic PDE on a 5 x 5 grid. Each node denotes each of the 3 grid levels obtained by semi-coarsening (5 x 5, 5 x 3, and 3 x 3 grids). A downward sloping line denotes a restriction operation and an upward sloping line denotes an interpolation operation.]

The numerical integration of Eq. (2), an elliptic equation, employs a novel algorithm
in comparison to our previous studies [7]. In our prior investigations, the elliptic equation
was solved using a full multigrid approach [4], which collected values for $V_m$ distributed
amongst several processors as input into the full multigrid algorithm on a single
processor. This processor then sequentially computes the corresponding values $\Phi_e$ for
each point within the grid and distributes these values to the appropriate processors. In
this study, we modified and subsequently utilized a parallelized conventional multigrid
algorithm, which was originally excerpted from the hypre numerical methods library
developed in the Lawrence Livermore National Laboratory [9]. These modifications were
necessitated by a desire to reduce the computation time of the algorithm for our variant of
the bidomain model and the fact that the original parallel multigrid implementation was
tailored toward the solution of boundary-value problems with Dirichlet boundary
conditions, instead of the no-flux Neumann boundary conditions that apply to Eq. (2).
The adaptation of the algorithm included four major alterations: the incorporation of a
zebra-line Gauss-Seidel smoother [1], which is present in the previous sequential full
multigrid implementation; modification of the smoothing, interpolation, and restriction
steps in order to account for Neumann boundary conditions; the use of Gram-Schmidt
orthogonalization to determine the solution error in the coarsest grid; and reformulation
of the discretization stencil so that myocardial anisotropy is appropriately represented.
The parallelized multigrid algorithm employed here is also distinguished from the
previous sequential implementation in that it distributes the computational workload of
the different phases of the multigrid algorithm (smoothing, restriction, and interpolation)
amongst multiple processors. For illustrative purposes, a 9 x 9 starting grid size is used as
an example in Figure 2 and a 5 x 5 grid is presented as the finest grid level in Figures 3
Figure 2: Distribution of the grid representation of simulated 2D myocardium for smoothing, interpolation, and restriction operations in the parallel conventional multigrid algorithm. Presented here is an illustration of dividing up (illustrated by dotted grey boxes and arrows) increasingly smaller grids (from 9 x 9 at the top, to 5 x 5 in the center, and to 3 x 3 at the bottom) along the longitudinal (x-axis) amongst a fixed number of processors (4 in this example) that are determined at run-time. As the grid becomes coarser via restriction (denoted by large black arrows), it is partitioned into smaller sections along the longitudinal axis until only a portion of the processors remain active (grey boxes) and continue to exchange information via MPI (black double-headed arrows). The remaining processor remains idle (white box) until interpolation to a sufficiently finer grid occurs.
and 4. It follows from the partitioning scheme presented in Figure 2 that a grid of $(2^k + 1) \times (2^k + 1)$ nodes can be divided between at most $(2^k + 1)$ processors, where $k$ is an integer. In addition, it uses alternating semi-restriction and semi-interpolation, which are illustrated in Figures 3 and 4, respectively. Usage of these types of restriction and interpolation operations results in a requirement of $2^k - 1$ total grids for a multigrid numerical solution of a grid of $(2^k + 1) \times (2^k + 1)$ nodes; therefore, the $257 \times 257$ grid used here employs 15 total grids in the parallel multigrid algorithm.

**Experimental Protocol**

Rectilinear wave propagation in two-dimensional (2D) myocardium is initiated by applying a S1 stimulus with a value $I_{\text{stim}} = -60 \ \mu\text{A/cm}^2$ along the entire transverse fiber direction (left to right in our particular case) and a sufficient number of nodes (the top 21 in these experiments) along the longitudinal direction in order to overcome source-sink mismatch effects. The speed of propagation is measured using a central difference approximation of the wave’s conduction velocity, which is described in a previous investigation [15].

Spiral wave propagation in 2D myocardium that are characteristic of ventricular tachycardia (VT) are initiated using an S1-S2 pacing protocol. The S1 stimulus is applied in the same location and orientation as in the study of rectilinear propagation, which is followed by a premature S2 beat 30 nodes applied near the tail of the propagating wave at $t = 0.210$ s with amplitude $I_{\text{stim}} = -90 \ \mu\text{A/cm}^2$. A 30 node gap was left on the right side of the S2 beat, permitting the formation of a clockwise-rotating spiral wavefront. Spiral wave circulation is then permitted for approximately 1 second of simulation time, in order for propagation to become nearly stationary, and then a defibrillation shock was applied for 20 ms at $t = 1.015$ s.

**Results**

**Parallel Multigrid Yields Computation Time Improvement for Rectilinear Propagation**

The first portion of our investigation is concerned with the measurement of the overall savings in computation time achieved by replacing the sequential full multigrid algorithm that was previously used to solve Eq. (2) with the parallelized conventional multigrid algorithm derived from the hypre library. A summary of the findings from this investigation, which presents the relationship between total execution time and the number of processors, are in Table 1. Each simulation studied in this portion of the study was 1.0 s in duration and exhibited rectilinear propagation from the top portion of the tissue to the bottom along the direction of greatest conductivity, using the protocol described in the Methods section. A comparison of the wavefront morphologies (as a function of time) obtained using the sequential and parallel multigrid algorithms revealed no noticeable differences between the two approaches. Moreover, both experiments exhibited equal conduction velocities for the rectilinear waves ($\theta = 52.5$ cm/s). However, the parallel multigrid algorithm demonstrates consistently decreasing overall execution times for 1.0 s simulations as the number of processors increases. Initially, the execution time for the sequential multigrid algorithm is somewhat faster than the parallel multigrid version on a single processor (~700 s versus 832.711 s). However, the parallel version
Figure 3: Alternating semi-restriction operations converting a 2D grid into successively coarser grids. In the semi-restriction operation, a square grid of size \((2^k + 1) \times (2^k + 1)\), which is 5 x 5 in this illustration, has information from the individual grid points transferred to a coarser grid that retains the same number of grid points along one dimension, but only \((2^{k-1} + 1)\) grid points along the other dimension, which is 5 x 3 in this case. During the restriction (denoted by large black arrows), grid points on the coarser grid are computed as a weighted average of the corresponding grid point on the finer grid and its neighbors (illustrated with grey dotted arrows). The restriction operation is then repeated, but along the alternate and perpendicular direction.
Figure 4: Alternating semi-interpolation operations converting a 2D grid into successively finer grids. In the semi-interpolation operation, a square grid of size $(2^k + 1) \times (2^k + 1)$, which is $3 \times 3$ in this illustration, has information from the individual grid points transferred to a finer grid that retains the same number of grid points along one dimension, but now $(2^{k+1} + 1)$ grid points along the other dimension, which is $5 \times 3$ in this case. During the interpolation (denoted by large black arrows), new grid points on the finer grid (shown as grey circles) are computed as a simple average of its neighbors on the finer grid (illustrated with grey dotted arrows). The interpolation operation is then repeated, but along the alternate and perpendicular direction.
Table 1: Relation between computation time for 1000 ms of simulation time exhibiting rectilinear propagation and the number of processors utilized. In this table, 1000 ms of a simulation involving rectilinear propagation in 2D myocardium is executed (both the operator splitting and multigrid portions) on a varying number of processors. Marginal reductions in the total time required to execute the simulations begin to become evident with 128 processors used for the 257 x 257 grid representing the 2D myocardium.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Actual Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>832.711</td>
</tr>
<tr>
<td>2</td>
<td>425.828</td>
</tr>
<tr>
<td>4</td>
<td>228.459</td>
</tr>
<tr>
<td>8</td>
<td>172.043</td>
</tr>
<tr>
<td>16</td>
<td>116.964</td>
</tr>
<tr>
<td>32</td>
<td>81.205</td>
</tr>
<tr>
<td>64</td>
<td>58.124</td>
</tr>
<tr>
<td>128</td>
<td>37.033</td>
</tr>
<tr>
<td>256</td>
<td>30.411</td>
</tr>
</tbody>
</table>

executes in a shorter time interval with only 2 processors and continues to exhibit significant marginal gains in execution time until 128 processors are used. Beyond that number of processors, the execution time for the parallel version continues to decrease, albeit at an attenuated rate.

The gain in execution speed is expressed in graphical form in Figure 5, which displays the execution speed-up using the parallel multigrid algorithm for a given number

![Speed-up of Rectilinear Propagation Simulations Relative to Sequential Execution](image)

Figure 5: Relation between the speed-up of simulation execution and the number of processors utilized. In this figure, 1000 ms of a simulation involving rectilinear propagation in 2D myocardium is executed (both the operator splitting and multigrid portions) on a varying number of processors. By definition, the gain in computational speed of utilizing multiple processors to solve the bidomain model equations is in relation to the model’s numerical solution on a single processor using the modified version of the parallelized conventional multigrid algorithm.
of processors with respect to the algorithm’s execution time on a single processor. It is evident that initially there is a near doubling of the speed-up of the overall program as the number of processors is doubled until 4 processors are used. Thereafter, the program execution time increases approximately 1.5 fold each time the number of processors is doubled until 128 processors are used. When the number of processors is increased to 256, the speed-up continues to increase to approximately 27 times of the sequential execution time, but the marginal gain in speed-up significantly decreases.

The investigation of rectilinear propagation in simulated 2D myocardium reveals no discrepancy between experiments utilizing the sequential or parallel multigrid algorithm. Furthermore, up to a 27 fold in program execution time is evident when increasing the number of processors used for the simulation, with the parallel multigrid algorithm, from 1 to 256. Thus, the parallel multigrid algorithm is a suitable substitute for the sequential version in the study of rectilinear propagation in 2D myocardium.

**Parallel Multigrid Yields Computation Time Improvement for Spiral Wave Propagation**

The second component of our study extended the comparison of the sequential and parallel multigrid algorithms in the case of rectilinear propagation in simulated 2D myocardium to the case of spiral wave propagation, with the results of different execution times summarized in Table 2. As before, each simulation studied in this portion of the investigation was 1.0 s in duration. However, spiral wave propagation was initiated using the S1-S2 pacing protocol described in the Methods section, in contrast to the simple S1 stimulus used for initiating rectilinear propagation. A comparison of the spiral wavefront morphologies (as a function of time) obtained using the sequential and parallel multigrid algorithms revealed only a small (0.0125 cm) shift of the spiral wave in the parallel multigrid case ahead of the sequential multigrid case. This discrepancy disappeared after commencing the application of the defibrillation shock and the postshock phase of the simulation exhibited identical spatial distributions of \( V_m \). Moreover, both multigrid implementations presented equivalent periods of rotation (approximately 150 ms) and times that the spiral wave reached the stationary phase (near \( t = 0.64 \) s). An interesting difference compared to rectilinear propagation is evident in that the marginal gain in total execution time saturates with only 64 processors, instead of 128 processors. Nevertheless, the parallel multigrid algorithm shows decreasing overall execution times for an increasing number of processors, albeit at an attenuated rate.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Actual Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5316.96</td>
</tr>
<tr>
<td>2</td>
<td>3827.68</td>
</tr>
<tr>
<td>4</td>
<td>1721.17</td>
</tr>
<tr>
<td>8</td>
<td>1226.58</td>
</tr>
<tr>
<td>16</td>
<td>803.53</td>
</tr>
<tr>
<td>32</td>
<td>683.48</td>
</tr>
<tr>
<td>64</td>
<td>563.42</td>
</tr>
<tr>
<td>128</td>
<td>545.32</td>
</tr>
<tr>
<td>256</td>
<td>523.95</td>
</tr>
</tbody>
</table>

Table 2: Relation between computation time for 1000 ms of simulation time exhibiting spiral wave propagation and the number of processors utilized. In this figure, 1000 ms of a simulation involving 2D propagation is executed (both the operator splitting and multigrid portions) on a varying number of processors. Marginal reductions in the total time required to execute the simulations begin to become evident with 64 processors used for the 257 x 257 grid representing the 2D myocardium.
The gain in execution speed is expressed in graphical form in Figure 6, in the same manner as Figure 1. Starting with only 2 processors, the program execution time increases approximately 1.5 fold each time the number of processors is doubled until 64 processors are used. When the number of processors is increased beyond that number, the speed-up continues to increase to approximately 10 times of the sequential execution time, but the marginal gain in speed-up significantly decreases.

The investigation of spiral wave propagation in simulated 2D myocardium displayed only a minor difference in wave position between experiments utilizing the sequential or parallel multigrid algorithm, which was extinguished by applying the defibrillation shock. Furthermore, up to a 10 fold speed-up in program execution time is evident when increasing the number of processors used for the simulation, with the parallel multigrid algorithm, from 1 to 256. Thus, the parallel multigrid algorithm is a more efficient and suitable substitute for the sequential multigrid algorithm in the study of defibrillation phenomena in simulated 2D myocardium.

![Speed-up of Spiral Wave Simulations Relative to Sequential Execution](image)

Figure 6: Relation between the speed-up of simulation execution and the number of processors utilized. In this figure, 1000 ms of a simulation involving spiral wave propagation is executed (both the operator splitting and multigrid portions) on a varying number of processors. By definition, the gain in computational speed of utilizing multiple processors to solve the bidomain model equations is in relation to the model’s numerical solution on a single processor using the modified version of the parallelized conventional multigrid algorithm.

**Discussion**

The multigrid approach is only one of several different approaches for solving elliptic PDEs. Direct methods that simultaneous solve for the system of equations that arises
from discretization of the PDE, such as matrix inversion, are inherently advantageous in comparison to multigrid methods since an accurate spatial distribution of $\Phi_e$ can be obtained in a relatively straightforward fashion. However, direct methods are also among the most computational time-intensive solution techniques, since they typically entail $O(N^2)$ or greater mathematical operations in a serial implementation, where $N$ is the total number of grid points [17]. In contrast, indirect methods such as Jacobi, Gauss-Seidel, and conjugate gradient use estimates of the spatial distribution of $\Phi_e$ and successively refine these estimates to achieve the desired solution to a specified degree of accuracy within a shorter period of time. Multigrid approaches, combined with indirect methods for numerical error reduction, are able to solve elliptic PDEs in relatively short time intervals, requiring an average of $O(N)$ and $O(\log^2 N)$ mathematical operations for serial and idealized (by neglecting latencies due to interprocessor data exchange) parallel implementations [17], respectively. This overall order of operations closely approaches the lower theoretical limit for floating point operations and hence multigrid approaches can achieve a desired solution with a minimal number of computations. In the following sections, a comparison of the sequential full multigrid algorithm and the parallel conventional multigrid algorithm is presented, followed by a consideration of the performance of the latter type of multigrid in the presence of rectilinear and spiral wave propagation, and concluded by a critique of the study’s limitations.

**Parallel Multigrid Results in More Effective Parallelization of the Entire Numerical Solution Scheme**

Our initial investigation into the mechanisms by which early afterdepolarizations (EADs) could affect defibrillation efficacy [7] utilized a full multigrid algorithm that executed on a single processor. We found in unpublished portions of our study that the full multigrid algorithm, on average, exhibited a faster execution time than the conventional multigrid algorithm on a single processor. In its previous implementation, the sequential execution of the full multigrid algorithm utilizes only one processor to compute the entire spatial distribution of $\Phi_e$ corresponding to the spatial distribution of $V_m$. This approach, while able to provide $\Phi_e$ at each grid point with a reasonable degree of accuracy, is flawed by the significant time that the remaining processors are idle during the multigrid computations. Attempts to parallelize the full multigrid algorithm in order to reduce this idling of processors becomes complicated by the fact that successive $V$-cycles occur first on the coarsest grid level and progresses to finer grids; if the number of processors exceeds the number of grid points along one dimension of the grid, processors will remain idle throughout a significant portion of the multigrid $V$-cycle iterations.

This dilemma can be ameliorated by adopting the conventional approach for the multigrid algorithm and parallelizing this variant. Since each $V$-cycle in the conventional multigrid algorithm entails visiting all grid levels only twice (once in the residual/restriction phase and a second time in the interpolation/correction phase), it is possible to utilize 1 $V$-cycle if the residual associated with the solution’s error is reduced to an acceptably miniscule value. Furthermore, the time spent in the portion of the conventional multigrid algorithm that involves grid sizes with linear dimensions along one dimension smaller than the number of processors is reduced, which permits better distribution of computational tasks among several processors. As a consequence, parallelization of the conventional multigrid algorithm leads to decreased windows of
time exhibiting processor idling, facilitating the scalability of not only the multigrid algorithm, but the entire numerical solution of the bidomain model. The scalability trends (presented in Figures 5 and 6) continue until the number of processors approaches the size of the grid along one dimension. At this point, the multigrid algorithm exhibits a majority of its time in grid levels with a size along one-dimension smaller than the number of processors.

**Mode of Wave Propagation Affects Computational Speed of Parallel Multigrid**

A comparison of the rectilinear and spiral wave propagation experiments reveals that for the same grid size and mathematical AP model, the parallel conventional multigrid algorithm yielded approximately the same results as the sequential full multigrid algorithm. This was observed in both types of electrical propagation in the myocardium: rectilinear wave propagation and spiral wave propagation. In the case of spiral wave propagation, the results were nearly identical in the distribution of $V_m$, with a slight shifting of 0.0125 cm between the parallel conventional multigrid algorithm and the sequential full multigrid algorithm (Figure 7). No such shifting was seen with purely rectilinear propagation.

The most straightforward explanation for this outcome is the fact that an alternating Gauss-Seidel method that solves for all grid points in a given column simultaneously (the zebra-line method) more effectively solves for $\Phi_e$ when the distribution of $V_m$ is approximately one dimensional in distribution versus a more complex pattern, as is exhibited in spiral wave propagation.

![Figure 7: Comparison of distribution of $V_m$ in 3.2 x 3.2 simulated myocardium, during 2D spiral wave propagation at time $t = 1.25$ s, using the sequential full multigrid algorithm (left) and the parallel conventional multigrid algorithm. The color scheme is the following: red is above +21 mV, yellow is between +21 mV and +10 mV, green is between +10 mV and -74 mV, and blue is below -74 mV.](image)

**Limitations**

The significance of the development of this new parallel multigrid algorithm must also account for the inherent limitations of our study. First, our investigation only utilized a structured grid representation of the myocardium as rectangular in shape and assumed uniform anisotropy throughout the grid. This had the effect of simplifying the general form of the bidomain equations to the reduced form presented in Eqs. (1) and (2), which then permitted the use of multigrid algorithms tailored toward linear PDEs. If the grid had represented variable fiber orientation within the plane of the myocardium, as present in the real heart, it would have been necessary to formulate a variation of the multigrid approach that appropriately treats nonlinear elliptic PDEs.
Another potential drawback lies in the performance of the multigrid algorithm itself. The current approach entails descent to the coarsest grid before ascending back to the original grid level, with several grid levels too small to employ all of the processors. Future modifications could divide the grid into blocks after the grid has been coarsened to a certain level, permitting continued usage of as many of the processors as possible. Alternatively, coarsening of the grid to the point where all processors could still be employed and executing several smoothing steps (instead of further restriction operations) could also maximize processor utilization.

**Conclusion**

The presented results show that in order for the investigation of excitation-propagation in 2D and 3D tissue using the bidomain model of myocardium to be possible, it is necessary to fully parallelize the solution using massively parallel supercomputers. For this purpose, we proposed the parallelization of conventional multigrid algorithm for the solution of the field distribution in the extracellular domain, which is described by an elliptic PDE.

Our investigation, using this novel multigrid algorithm, produced several results: (1) a 10-fold decrease of the calculation speed for the bidomain model, using 64 processors, without introducing significant error in the solution, which followed a linearly decreasing trend of computation time when the number of processors increased from 1 to 64; (2) as the number of processors increased beyond 64, the computation time of the bidomain model continued to decrease, but only asymptotically; and (3) virtually identical distributions of \( V_m \) during spiral wave propagations were produced by the sequential full multigrid and parallel conventional multigrid algorithms, differentiated by a slight shift of the \( V_m \) distribution by 0.0125 cm.

These results open future avenues of investigation for computer simulations of electrical phenomena in 3D tissue that is described by the bidomain model and, in particular, the investigation of phenomena associated with defibrillation.

**Acknowledgements**

Mr. Samade was supported by the UCLA Graduate Division under the Dissertation Year Research Fellowship. Mr. Raghavan was supported by funding from a UCLA Computer Science Leonard Kleinrock grant. Supercomputing resources were provided by the National Energy Research Scientific Computing Center, Office of Energy Research of the United States Department of Energy, under contract number DEAC0376SF00098.
Appendix

General Description of Modifications to the hypre Numerical Library

We implemented our modifications in an original 2.0.0 version of hypre and summarized them in the list below. For brevity, only modifications to hypre are presented in the samples of C++ code and portions of unmodified code that intersperse code blocks are denoted with an ellipsis. All alterations are applied to the pfmg structured linear solver class of functions, which is found in the directory hypre-2.0.0/src/struct ls.

Below is the list of all changes to the library.

_hypre_struct_ls.h (Page 17):
- Header file containing prototypes. Added headers for new function hypre_LineRelax

semi_setup_rap.c (Pages 17-18):
  hypre_SemiBuildRAP:
  - Setup of the A-matrix coefficients for coarser grid sizes. Modified to hard-code matrix coefficients instead of using the Galerkin operator and incorporates Neumann boundary conditions.

semi_restrict.c (Pages 18-19):
  hypre_semiRestrict:
  - The restriction operator. Implements full-weighted linear restriction and incorporates Neumann boundary conditions.

semi_interp.c (Pages 19-20):
  hypre_semiInterp:
  - The interpolation operator. Implements full-weighted linear interpolation and incorporates Neumann boundary conditions.

redblack_gs.c (Pages 20-24):
  hypre_RedBlackGS:
  - The point red-black Gauss Seidel smoother. Modified to handle non-Galerkin coefficient matrix given in a structured 9-point stencil.
  hypre_LineRelax:
  - New function. The zebra-line smoother based on the Thomas trigidiagonal algorithm. Code based off of the point smoother (above) and based on the same 9-point stencil.

pfmg_solve.c (Pages 25-27):
  hypre_PFMGSolve: List of changes:
  - Before running multigrid, check for near-zero initial distribution and return zero solution
  - Pre-relaxation line smoothing using hypre_LineRelax, residual, and semi-restriction operations performed down to a coarse 3x3 grid level. Originally, goes down to 1x1 grid.
  - At the bottom of the V-cycle, before interpolation of error, enforce zero-mean via Gram-Schmidt orthogonalization.

pfmg_relax.c (Not Presented)
  hypre_PFMGRelax, hypre_PFMGRelaxSetup, hypre_PFMGRelaxSetType, hypre_PFMGRelaxSetPreRelax, hypre_PFMGRelaxSetPostRelax:
  - Added cases relax_type 4 and 5 to allow external setup of the pfmg solver to choose hypre_lineRelax as the smoother. The additions are trivial, and not shown.
/* Beginning of Modifications to the _hypre_struct_ls.h file: */

// prototype function that is zebra line Gauss-Seidel smoother
// implemented using the Thomas Tridiagonal algorithm
int hypre_LineRelax ( void *relax_vdata , hypre_StructMatrix *A , hypre_StructVector *b , hypre_StructVector *x );

/* End of Modifications to the _hypre_struct_ls.h file: */

/* Beginning of Modifications to the semi_setup_rap.c file: */

/* semi_setup_rap.c */

// Setup the coefficient A-matrix for coarse grids. Modified to hard-code matrix coefficients instead of using the Galerkin operator.

hypre_SemiBuildRAP( ... )
{
    // ...
    hypre_ForBoxI(ci, cgrid_boxes)
    {
        while (fgrid_ids[fi] != cgrid_ids[ci]) fi++;
        cgrid_box = hypre_BoxArrayBox(cgrid_boxes, ci);
        cstart = hypre_BoxIMin(cgrid_box);
        hypre_StructMapCoarseToFine(cstart, cindex, cstride, fstart);
        hypre_BoxGetSize(cgrid_box, loop_size);

        A_dbox = hypre_BoxArrayBox(hypre_StructMatrixDataSpace(A), fi);
        P_dbox = hypre_BoxArrayBox(hypre_StructMatrixDataSpace(P), fi);
        R_dbox = hypre_BoxArrayBox(hypre_StructMatrixDataSpace(R), fi);
        RAP_dbox = hypre_BoxArrayBox(hypre_StructMatrixDataSpace(RAP), ci);

        // find the offset and size of the local grid relative to global grid
        int local_xmin = hypre_BoxIMinX(RAP_dbox)+1;
        int local_ymin = hypre_BoxIMinY(RAP_dbox)+1;
        hypre_Box *rap_box = RAP->grid->box_man->bounding_box;
        int size_x = hypre_BoxIMaxX(rap_box);
        int size_y = hypre_BoxIMaxY(rap_box);

        // obtain pointers to 5-point stencil about current point.
        double *coef_cc = hypre_StructMatrixBoxData(RAP, ci,0);
        double *coef_xmax = hypre_StructMatrixBoxData(RAP, ci,1);
        double *coef_xmin = hypre_StructMatrixBoxData(RAP, ci,2);
        double *coef_ymax = hypre_StructMatrixBoxData(RAP, ci,3);
        double *coef_ymin = hypre_StructMatrixBoxData(RAP, ci,6);

        // The coefficients are constant in each direction
        double eps2 = 0.09;
        double coef_x = size_x*size_x*eps2;
        double coef_y = size_y*size_y;
        double coef_diag = -2*(coef_x + coef_y);

        // set the coefficients for each point in the local grid
        hypre_BoxLoop1Begin(loop_size, RAP_dbox, cstart, stridec, iAc);
        #define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj,iAc
        #include "hypre_box_smp_forloop.h"
        hypre_BoxLoop1For(loopi, loopj, loopk, iAc)
{  
  int xi = local_xmin + loopi;
  int xj = local_ymin + loopj;

  // The coefficients are setup according to boundary conditions
  coef_cc[iAc] = coef_diag;
  coef_xmin[iAc] = coef_x;
  coef_xmax[iAc] = coef_x;
  coef_ymin[iAc] = coef_y;
  coef_ymax[iAc] = coef_y;

  if(xi==0)
  {
    coef_xmin[iAc] -= coef_x;
    coef_xmax[iAc] += coef_x;
  }

  if(xj==0)
  {
    coef_ymin[iAc] -= coef_y;
    coef_ymax[iAc] += coef_y;
  }

  if(xi==(size_x))
  {
    coef_xmin[iAc] += coef_x;
    coef_xmax[iAc] -= coef_x;
  }

  if(xj==(size_y))
  {
    coef_ymin[iAc] += coef_y;
    coef_ymax[iAc] -= coef_y;
  }

  hypre_BoxLoop1End(iAc);
}

// ...

#ifndef HYPRE_REGIONS

/* End of Modifications to the semi_setup_rap.c file: */

/* Beginning of Modifications to the semi_restrict.c file: */

#ifndef HYPRE_REGIONS

/* semi_restrict.c */

// The semi-restriction function
hypre_semiRestrict ( ... )
{
  // ...
  if(constant_coefficient = 1) // assume constant coefficients, always true
  {
    // get the local grid offset and size
    hypre_Box* ac_box = R->grid->box_man->bounding_box;
    int local_xmin = hypre_BoxIMinX(R_dbox)+1;
    int local_ymin = hypre_BoxIMinY(R_dbox)+1;
    int size_x = hypre_BoxIMaxX(ac_box);
    int size_y = hypre_BoxIMaxY(ac_box);

    // direction of restriction (alternates)
    int cdir = (stride[0] == 2) ? 0 : 1;

    // loop over points in local grid, depends on stride

#endif /* HYPRE_REGIONS */

#endif /* HYPRE_REGIONS */

18
hypre_BoxLoop2Begin(loop_size, r_dbox, start, stride, ri,
         rc_dbox, startc, stridedc, rci);
#define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj,ri,rci
#include "hypre_box_smp_forloop.h"
hypre_BoxLoop2For(loopi, loopj, loopk, ri, rci)
{
    int xi = local_xmin + loopi;
    int xj = local_ymin + loopj;

    // rp0[ri], rp1[ri] are the immediate neighbors in the direction
    // of restriction.
    // The coefficients depend on boundary conditions,
    // which, in turn, depends on direction of restriction.
    if(((cdir==0) && (xi==0)) || ((cdir==1) && (xj==0)))
        rcp[rci] = 0.5*(rp[ri]) + 0.5*(rp1[ri]);
    else if(((cdir==0) && (xi==size_x)) || ((cdir==1) &&
        (xj==size_y)))
        rcp[rci] = 0.5*(rp[ri]) + 0.5*(rp0[ri]);
    else rcp[rci] = 0.5*(rp[ri]) + 0.25*(rp0[ri] + rp1[ri]);
}
hypre_BoxLoop2End(ri, rci);

/* End of Modifications to the semi_restrict.c file: */

/* Beginning of Modifications to the semi_interp.c file: */

/* semi_interp.c */
// The semi-interpolation function
hypre_SemiInterp ( ... )
{
    if ( constant_coefficient  = 1) // assume constant coefficients, always true
    {
        Pi = hypre_CCBoxIndexRank( P_dbox, startc );

        // loop over points in local grid, depending on stride.
        hypre_BoxLoop1Begin(loop_size, e_dbox, start, stride, ei);
        #define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj,ei
        #include "hypre_box_smp_forloop.h"
        hypre_BoxLoop1For(loopi, loopj, loopk, ei)
        {
            // simply interpolate depending on the immediate neighbors
            // in the direction of interpolation, ep0[ei] and ep1[ei] are
            ep[ei] = (0.5 * ep0[ei] + 0.5 * ep1[ei]);
        }
        hypre_BoxLoop1End(ei);
    }

    // ...
}
/* redblack_gs.c */
// Only the modification highlighted. Add “case 9” to the switch(stencil_size)
// statement.
// This is necessary since the hardcoding of the A-matrix at all grid levels is
// using a 9-point stencil
hpre_RedBlackGS ( ... )
// ...
switch(stencil_size)
{
    // ... other cases (unmodified)
    // take care of the stencil_size==9 (non-Galerkin structured stencil)
    case 9:
        Ap3 = hpre_StructMatrixBoxData(A, i, 6);
        Ap2 = hpre_StructMatrixBoxData(A, i, 3);
        Ap1 = hpre_StructMatrixBoxData(A, i, 2);
        Ap0 = hpre_StructMatrixBoxData(A, i, 1);
        xoff3 = hpre_BoxOffsetDistance(x_dbox, stencil_shape[6]);
        xoff2 = hpre_BoxOffsetDistance(x_dbox, stencil_shape[3]);
        xoff1 = hpre_BoxOffsetDistance(x_dbox, stencil_shape[2]);
        xoff0 = hpre_BoxOffsetDistance(x_dbox, stencil_shape[1]);
    }
    // ...
}

/* End of Modifications to the semi_interp.c file: */

/* Beginning of Modifications to the redblack_gs.c file: */

/* redblack_gs.c */
// The entire function is included for completeness. This function is
// is not part of the original HYPRE library
/*--------------------------------------------------------------------------
 * hypre_LineRelax – Zebra line relaxation using Thomas tridiagonal algorithm.
 * The code below is derived from the point smoother: hypre_REDBlackGS
 *--------------------------------------------------------------------------*/
int
hpre_LineRelax( void                *relax_vdata,
    hpre_StructMatrix *A,
    hpre_StructVector *b,
    hpre_StructVector *x )
{
    // initialize data structures
    hpre_RedBlackGSData  *relax_data = relax_vdata;

    int          max_iter    = (relax_data -> max_iter);
    int          zero_guess  = (relax_data -> zero_guess);
    int          rb_start    = (relax_data -> rb_start);
    int          diag_rank   = (relax_data -> diag_rank);
    hpre_ComputePkg      *compute_pkg = (relax_data -> compute_pkg);

    hpre_CommmHandle     *comm_handle;
    hpre_BoxArrayArray   *compute_box_aa;
    hpre_BoxArray        *compute_box_a;
    hpre_Box             *compute_box;
hypre_Box             *A_dbox;
hypre_Box             *b_dbox;
hypre_Box             *x_dbox;

int                    Ai, Astart, Ani, Anj;
int                    bi, bstart, bni, bnj;
int                    xi, xstart, xni, xnj;
int                    xoff0, xoff1, xoff2, xoff3, xoff4, xoff5;

double                *Ap;
double                *bp;
double                *xp;

hypre_IndexRef         start;
hypre_Index            loop_size;

hypre_StructStencil   *stencil;
hypre_Index            *stencil_shape;
int                    stencil_size;
int                    offd[6];

int                    iter, rb, redblack;
int                    compute_i, i, j, ii, jj, kk;
int                    ni, nj, nk;

int                    ierr = 0;

hypre_BeginTiming(relax_data -> time_index);
{
    stencil       = hypre_StructMatrixStencil(A);
    stencil_shape = hypre_StructStencilShape(stencil);
    stencil_size  = hypre_StructStencilSize(stencil);

    /* get off-diag entry ranks ready */
    i = 0;
    for (j = 0; j < stencil_size; j++)
    {
        if (j != diag_rank)
        {
            offd[i] = j;
            i++;
        }
    }
}

rb = 0;
iter = 0;

if(zero_guess)
{
    hypre_StructVectorSetConstantValues(x,0.0);
}

int sweep;
// There are two sweeps, red and black
for(sweep = 0; sweep < 2; sweep++) {

// either get the "box" from local storage or through retrieve boundary
// boxes from neighboring processors, automatically done by the HYPRE compute
package
for (compute_i = 0; compute_i < 2; compute_i++)
{
    switch(compute_i)
    {
    case 0:
    {
        xp = hypre_StructVectorData(x);
        hypre_InitializeIndtComputations(compute_pkg, xp, &comm_handle);
        compute_box_aa = hypre_ComputePkgIndtBoxes(compute_pkg);
    }
    break;
    case 1:
    {
        hypre_FinalizeIndtComputations(comm_handle);
        compute_box_aa = hypre_ComputePkgDeptBoxes(compute_pkg);
    }
    break;
    }

    // loop through each box
    hypre_ForBoxArrayI(i, compute_box_aa)
    {
        compute_box_a = hypre_BoxArrayArrayBoxArray(compute_box_aa, i);
        A_dbox = hypre_BoxArrayBox(hypre_StructMatrixDataSpace(A), i);
        b_dbox = hypre_BoxArrayBox(hypre_StructVectorDataSpace(b), i);
        x_dbox = hypre_BoxArrayBox(hypre_StructVectorDataSpace(x), i);
        Ap = hypre_StructMatrixBoxData(A, i, diag_rank);
        bp = hypre_StructVectorBoxData(b, i);
        xp = hypre_StructVectorBoxData(x, i);

        hypre_ForBoxI(j, compute_box_a)
        {
            compute_box = hypre_BoxArrayBox(compute_box_a, j);
            start = hypre_BoxIMin(compute_box);
            hypre_BoxGetSize(compute_box, loop_size);

            /* Are we relaxing index start or start+(1,0,0)? */
            redblack = hypre_abs(hypre_IndexX(start) +
                                hypre_IndexY(start) +
                                hypre_IndexZ(start) + sweep) % 2;

            Astart = hypre_BoxIndexRank(A_dbox, start);
            bstart = hypre_BoxIndexRank(b_dbox, start);
            xstart = hypre_BoxIndexRank(x_dbox, start);

            ni = hypre_IndexX(loop_size);
            nj = hypre_IndexY(loop_size);
            nk = hypre_IndexZ(loop_size);
            Ani = hypre_BoxSizeX(A_dbox);
            bni = hypre_BoxSizeX(b_dbox);
            xni = hypre_BoxSizeX(x_dbox);
            Anj = hypre_BoxSizeY(A_dbox);
            bnj = hypre_BoxSizeY(b_dbox);
            xnj = hypre_BoxSizeY(x_dbox);
```c
// Thomas algorithm intermediate variables
double *cprime = (double*) malloc(ni * sizeof(double));
double *dprime = (double*) malloc(ni * sizeof(double));
double a_i, b_i, c_i, d_i, denom;

// Setup pointers to vector and grid coefficients depending on the stencil
switch(stencil_size)
{
    case 5:
        Ap3 = hypre_StructMatrixBoxData(A, i, offd[3]);
        Ap2 = hypre_StructMatrixBoxData(A, i, offd[2]);
        Ap1 = hypre_StructMatrixBoxData(A, i, offd[1]);
        Ap0 = hypre_StructMatrixBoxData(A, i, offd[0]);
        // the stencil offsets: ymin, ymax, xmax, xmin immediate neighbors
        xoff3 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[offd[3]]);
        xoff2 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[offd[2]]);
        xoff1 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[offd[1]]);
        xoff0 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[offd[0]]);
        break;
    case 9:
        Ap3 = hypre_StructMatrixBoxData(A, i, 6); // ymin
        Ap2 = hypre_StructMatrixBoxData(A, i, 3); // ymax
        Ap1 = hypre_StructMatrixBoxData(A, i, 1); // xmax
        Ap0 = hypre_StructMatrixBoxData(A, i, 2); // xmin
        xoff3 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[6]);
        xoff2 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[3]);
        xoff1 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[2]);
        xoff0 = hypre_BoxOffsetDistance(x_dbox, stencil_shape[1]);
}
```

```c
#define HYPRE_BOX_SMP_PRIVATE ii,jj,Ai,bi,xi
#include "hypre_box_smp_forloop.h"
for (kk = 0; kk < nk; kk++)
{
    for (jj = 0; jj < nj; jj+=1)
    {
        ii = 0;
        Ai = Astart + kk*Anj*Ani + jj*Ani + ii;
        bi = bstart + kk*bnj*bni + jj*bni + ii;
        xi = xstart + kk*xnj*xni + jj*xni + ii;
    }
}
```

```c
// One sweep (red/black) of the Thomas tridiagonal algorithm (TTA)
// applied to each box
for (kk = 0; kk < nk; kk++)
{
    for (jj = redblack; jj < nj; jj+=2)
    {
        ii = 0;
        Ai = Astart + kk*Anj*Ani + jj*Ani + ii;
```
bi = bstart + kk*bnj*bni + jj*bni + ii;
xi = xstart + kk*xnj*xni + jj*xni + ii;

// assign initial values to TTA variables
c_i = Ap1[Ai];
b_i = Ap[Ai];
a_i = Ap0[Ai];
d_i = bp[bi] - (Ap3[Ai]*xp[xi+xoff3]) -
(Ap2[Ai]*xp[xi+xoff2]);
cprime[ii] = c_i/b_i;
dprime[ii] = d_i/b_i;
i++; Ai++; bi++; xi++;

// iterate, building up array of c' and d'
for (; ii < ni; ii++, Ai++, bi++, xi++)
{
c_i = Ap1[Ai];
b_i = Ap[Ai];
a_i = Ap0[Ai];
d_i = bp[bi] -
(Ap3[Ai]*xp[xi+xoff3]) - (Ap2[Ai]*xp[xi+xoff2]);
denom = (b_i - cprime[ii-1] * a_i);
cprime[ii] = c_i/denom;
dprime[ii] = (d_i - dprime[ii-1] * a_i)/denom;
}
ii--; Ai--; bi--; xi--;
xp[xi] = dprime[ii];
i--; Ai--; bi--; xi--;

// apply changes to original vector
for (; ii >= 0; ii--, Ai--, bi--, xi--)
{
xp[xi] = dprime[ii] - cprime[ii] * xp[xi+1];
}
}

// memory-management
free(cprime);
free(dprime);
}
}

hypre_IncFLOPCount(relax_data -> flops);
hypre_EndTiming(relax_data -> time_index);
return ierr;

/* End of Modifications to the redblack_gs.c file: */
/* Beginning of Modifications to the pfmg_solve.c file: */

/* pfmg_solve.c */
// The main loop of the multigrid solver. Iterates through V-cycles, calling appropriate
// interpolation and restriction functions.
// Only the additions or modifications to the pfmg_solve function are highlighted
hypre_PFMGSolve( ... )
{
    // ...

    /*---------------------------------------------------------------------
    * Initialize some things and deal with special cases
    *---------------------------------------------------------------------*/
    if (fabs(b_dot_b) < 1e-10)
    {
        hypre_StructVectorSetConstantValues(x, 0.0);
        // return ...
    }

    /*---------------------------------------------------------------------
    * Do V-cycles:
    *   For each index l, "fine" = l, "coarse" = (l+1)
    *---------------------------------------------------------------------*/
    for (i = 0; i < max_iter; i++)
    {
        // Restrict residuals down to a 3x3 grid
        for (l = 1; l <= (num_levels - 6); l++)
        {
            /*------------------------------------------------------------------
            * Down cycle : unmodified
            *------------------------------------------------------------------*/
        }

        /*---------------------------------------------------------------------
        * Bottom: compute 3x3 solution
        *---------------------------------------------------------------------*/
        hypre_PFMGRelaxSetZeroGuess(relax_data_l[l], 1);
        hypre_PFMGRelax(relax_data_l[l], A_l[l], b_l[l], x_l[l]);
        hypre_PFMGRelaxSetMaxIter(relax_data_l[l], 10);

        /*---------------------------------------------------------------------
        * Enforce a zero-mean solution
        *---------------------------------------------------------------------*/

        /*---------------------------------------------------------------------
        * Step 0: Initialize variables
        *---------------------------------------------------------------------*/
        hypre_StructVector *x = x_l[l];
        //
        double          *local_result_ref[hypre_MAX_THREADS];
        int              threadid = hypre_GetThreadID();
        double          final_result;
        double          local_result = 0.0;
        double          process_result = 0.0;
        hypre_Box       *x_data_box;
        int              xi;
double *xp;
hYPRE_BoxArray *boxes;
hYPRE_Box *box;
hYPRE_Index loop_size;
hYPRE_IndexRef start;
hYPRE_Index unit_stride;
int i;
int loopi, loopj, loopk;

hypre_setIndex(unit_stride, 1, 1, 1);
boxes = hypre_StructGridBoxes(hypre_StructVectorGrid(x));

/// Get the sum of the vector in local storage
hypre_ForBoxI(i, boxes)
{
    box = hypre_BoxArrayBox(boxes, i);
    start = hypre_BoxIMin(box);
    x_data_box = hypre_BoxArrayBox(hypre_StructVectorDataSpace(x), i);

    xp = hypre_StructVectorBoxData(x, i);
    hypre_BoxGetSize(box, loop_size);
    #ifdef HYPRE_USE_PTHREADS
    local_result_ref[threadid] = &local_result;
    #endif
    hypre_BoxLoop1Begin(loop_size, x_data_box, start, unit_stride, xi);
    #define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj, xi,yi
    #define HYPRE_SMP_REDUCTION_OP +
    #define HYPRE_SMP_REDUCTION_VARS local_result
    #include "hypre_box_smp_forloop.h"
    hypre_BoxLoop1For(loopi, loopj, loopk, xi)
    {
        local_result += xp[xi]; // add to local sum
    }
    hypre_BoxLoop1End(xi);
}

#ifdef HYPRE_USE_PTHREADS
if (threadid != hypre_NumThreads)
{
    for (i = 0; i < hypre_NumThreads; i++)
    process_result += *local_result_ref[i];
}
else
{
    process_result = *local_result_ref[threadid];
}
#else
process_result = local_result;
#endif

/// Step 2: Sum up all local results into final_result
MPI_Allreduce(&process_result, &final_result, 1,
        MPI_DOUBLE, MPI_SUM, hypre_StructVectorComm(x));

#ifdef HYPRE_USE_PTHREADS
if (threadid == 0 || threadid == hypre_NumThreads)
#endif
hypre_IncFLOPCount(2*hypre_StructVectorGlobalSize(x));

/// Step 3: Subtract out the mean from each element
hypre_Box *abox = A_l[l]->grid->box_man->bounding_box;
double vector_mean = final_result / 9.0;
hypre_ForBoxI(i, boxes)
{
    box   = hypre_BoxArrayBox(boxes, i);
    start = hypre_BoxIMin(box);
    x_data_box = hypre_BoxArrayBox(hypre_StructVectorDataSpace(x),
        i);
    xp = hypre_StructVectorBoxData(x, i);
    hypre_BoxGetSize(box, loop_size);

#ifdef HYPRE_USE_PTHREADS
    local_result_ref[threadid] = &local_result;
#endif
    hypre_BoxLoop1Begin(loop_size,
        x_data_box, start, unit_stride, xi);
#define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj,xi
#define HYPRE_SMP_REDUCTION_OP +
#define HYPRE_SMP_REDUCTION_VARS local_result
#include "hypre_box_smp_forloop.h"
    hypre_BoxLoop1For(loopi, loopj, loopk, xi)
    {
        xp[xi] -= vector_mean;
    }
    hypre_BoxLoop1End(xi);
} // end loop
} // end enforce-zero-mean block

/* Up cycle: interpolate error back to finest grid level */
for (l = (num_levels - 6); l >= 1; l--)
{
    // code unmodified
}

/* fine grid post-relaxation */
hypre_PFMGRelaxSetPostRelax(relax_data_l[0]);
hypre_PFMGRelaxSetMaxIter(relax_data_l[0], num_post_relax);
hypre_PFMGRelaxSetZeroGuess(relax_data_l[0], 0);
hypre_PFMGRelax(relax_data_l[0], A_l[0], b_l[0], x_l[0]);

    // keep track of iterations
    (pfmg_data -> num_iterations) = (i + 1);
} // end max_iter loop
// ... return

/* End of Modifications to the pfmg_solve.c file: */

References

   Applications in Science and Engineering.
2. Trottenberg, U., C.W. Oosterlee, and A. Schuller, Multigrid: Basics, Parallelism


