ARMY RESEARCH LABORATORY

The Battlefield Environment Division Modeling Framework (BMF) Part I: Optimizing the Atmospheric Boundary Layer Environment Model for Cluster Computing

by Benjamin MacCall, Yansen Wang, and Giap Huynh

ARL-TR-6813 February 2014

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Adelphi, MD 20783-1197

ARL-TR-6813 **February 2014**

The Battlefield Environment Division Modeling Framework (BMF) Part I: Optimizing the Atmospheric Boundary Layer Environment Model for Cluster Computing

Benjamin MacCall, Yansen Wang, and Giap Huynh

Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
February 2014	Final	October 2012-September 2013

4. TITLE AND SUBTITLE
The Battlefield Environment Division Modeling Framework (BMF) Part I: Optimizing the Atmospheric Boundary Layer Environment Model for Cluster Computing

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Benjamin MacCall
Yansen Wang
Giap Huynh

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-CIE-D
Adelphi, MD 20783-1197

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-6813

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
primary author's email: <benajmin.t.maccall.civ@mail.mil>

14. ABSTRACT
We have updated the Atmospheric Boundary Layer Environment (ABLE) model to use multicore/multinode parallelization via the Message Passing Interface (MPI). The model source code has been rewritten to use the Battlefield Environment Division Modeling Framework (BMF), an object-oriented framework developed to facilitate implementation of new technologies and algorithms. BMF is documented in this report.

15. SUBJECT TERMS
Fluid Modeling, SIMPLE solver, Strongly Implicit Procedure, ABLE, BMF

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Benjamin T. MacCall
Unclassified	Unclassified	Unclassified	UU	48	19b. TELEPHONE NUMBER (*Include area code*) 301-394-1463

Contents

List of Figures **iv**

1. Introduction **1**

2. Methodology **3**

2.1 Design of BMF 4

2.2 Performing Serial and Parallel Calculations with Battlefield Environment Division Modeling Framework 5

2.3 Initiating and Completing Message Passing Interface (MPI) Communication 6

3. Results **12**

4. Conclusions **15**

Appendix. Battlefield Environment Division Modeling Framework – Fortran 2008 Interface **18**

List of Symbols, Abbreviations, and Acronyms **41**

Distribution List **42**

List of Figures

Figure 1. Calculation ordering for the standard Strongly Implicit Procedure (SIP) and for red-black ordering (from Reeve et al. [2001]) 4

Figure 2. Results from ABLE (blue line) with red-black ordering compared with laboratory data (circles) (Prasad and Koseff, 1989). (a) Vertical profile of non-dimensional, x-component of the velocity. (b) Horizontal profile of the vertical component of the velocity scaled by the velocity at the top of the cavity. (c) Vertical slice through the center of the domain, showing the non-dimensional magnitude and direction of the x- and z-velocity components 13

Figure 3. Scalability results for the red-black SIP-based semi-implicit method for pressure-linked equations (SIMPLE) integration scheme. (a) Normalized ABLE runtime compared with ideal scalability by dividing a constant-size domain over an increasing number of processes. (b) Scalability based on keeping the number of grid points per process constant. Adding processes, requires increasing the total computational domain size. All runtimes are normalized by the original serial version of ABLE without the overhead associated with including BMF 14

Figure 4. Rates of convergence for the serial version of the ABLE model using the standard SIP and for the red-black ordered ABLE model 15

1. Introduction

The Atmospheric Boundary Layer Environment (ABLE) model effort seeks to create a new atmospheric model tailored for microscale application (Wang et al., 2012). The model is currently being applied to highly idealized flows in order to characterize the dynamical core of the model—the schemes that form the fundamental basis for how fluid quantities are transported and diffused by the flow. The next stage will see the implementation of various physics parameterizations (e.g., turbulence closure) to enable more realistic simulations. ABLE is currently a basic research tool; however, as the model development advances, ABLE will be useful in applied research and operations and be deployable onto smaller platforms not just massively parallel clusters.

In the early stage of development, source code flexibility, ease of development, and ease of maintenance is often prioritized over performance. Standard practice in computer science is to develop programming libraries to aid in developer efficiency and code maintainability. In object-oriented design, libraries of routines are replaced with sets of commonly used data structures and associated operations that have been gathered into classes, and sets of classes combine to create *frameworks*. The Battlefield Environment Division Modeling Framework (BMF) adopts this design and is being developed in tandem with the ABLE model to increase code maintainability, reduce programming errors, and aid in the incorporation of new technology and ideas. BMF was designed to facilitate scientific computing by providing design clues to model developers and including abstracted classes for commonly used operations including parallel communication using Message Passing Interface (MPI) and grid decomposition. The basic usage of BMF and the results of implementing MPI-based parallel calculations in ABLE are documented in this report. Part 2 details features related to time-dependent simulations, structured grids, and parallel input and output using the Network Common Data Form (NetCDF) and Hierarchal Data Format version 5 (HDF5) libraries.

The current version of BMF is v0.85; it was developed using the latest Fortran standard, Fortran 2008, and has been tested primarily using the Intel Fortran compiler versions 12.1 and 13.0. The primary feature-set includes object-oriented programming, such as classes (using derived types), data encapsulation (via private components of derived types), methods (via type-bound procedures), and enhancements to Fortran pointer definitions (i.e., contiguous attribute) for performance intensive tasks. The framework will eventually be ported to other programming languages to aid in porting models such as ABLE to new platforms.

The initial test case for using BMF is the inclusion of efficient message passing for massively parallel computing applications via the MPI library. Current massively parallel clusters use a collection of multicore systems or nodes connected to a network fabric such as the low-latency, high-bandwidth, Infiniband. Even for cutting-edge technology, low-latency is relative to other network technologies like Ethernet; latencies are still large compared with communication between components within a single node (e.g., between processors and RAM). Developing applications capable of efficiently utilizing today's computers, with tens of thousands of processing cores, requires carefully selecting appropriate algorithms and hiding the latency by overlapping inter-process (especially inter-node) communication with calculations. Much of this framework has been designed to facilitate this kind of calculation by reducing the amount of duplicated code and allowing calculations to proceed with proper cache optimization.

BMF also facilitates the adoption of newer technologies and updated strategies. For example, the MPI-3 standard is in development, with some implementations already providing access to some of the newer features. A re-examination of the methods (e.g., user data packing) used to generate efficient MPI communication will be in order. Future ABLE development will involve acceleration using many-core architecture such as Intel's many integrated core (MIC) architecture or NVIDIA's Compute Unified Device Architecture (CUDA).

2. Methodology

The initial three-dimensional (3-D) version of ABLE uses the semi-implicit method for pressure-linked equations (SIMPLE) finite-volume method for steady-state convection-diffusion problems. The solution scheme uses the Strongly Implicit Procedure (SIP) (Stone, 1968) to solve the various sparse matrices. SIP is efficient and converges rapidly; however, it is not amenable to parallel computation. A parallel version of SIP was implemented by Reeve et al. (2001) by switching to a red-black ordering. Our implementation follows this strategy; however, the combination of the red-black scheme and the SIMPLE iterative method leads to difficulties that reduce the parallel efficiency.

The Cartesian Laplacian operator in three dimensions generates a multi-diagonal matrix, $\mathbf{M}$, with seven diagonals. SIP begins by generating a matrix that is close to the Laplacian operator, but has an $\mathcal{LU}$-factorization with four diagonals in the lower triangular matrix at the same bands as the original matrix, and four diagonals in the upper-triangular matrix again at the same bands as the original matrix. The new matrix, $\mathbf{M} + \mathbf{N}$ is put into the following iterative form,

$$\mathbf{M}\mathbf{x} = \mathbf{b} \tag{1}$$

$$(\mathbf{M} + \mathbf{N})\,\mathbf{x} = (\mathbf{M} + \mathbf{N})\,\mathbf{x} + (\mathbf{b} - \mathbf{M}\mathbf{x}) \tag{2}$$

$$(\mathbf{M} + \mathbf{N})\,\mathbf{x}^{n+1} = (\mathbf{M} + \mathbf{N})\,\mathbf{x}^{n} + \mathbf{R}^{n} \tag{3}$$

where $\mathbf{x}$ is the field of interest, $\mathbf{x}^n$ and $\mathbf{x}^{n+1}$ are the vector values of the current and next iteration, respectively, and $\mathbf{R}^n$ is the residual of the current iteration. Stone (1968) recognized that by constraining the extra diagonals in $\mathbf{M} + \mathbf{N}$ using a Taylor expansion, SIP will rapidly converge to a solution. SIP is commonly used in computational fluid dynamics (CFD), heat transfer, and other applications (Reeve et al., 2001). When parallelizing the method, there are three separate calculations: calculating the diagonals of the lower and upper triangular matrices for the $\mathcal{LU}$-factorization, the forward substitution using the lower triangular matrix ($\mathcal{L}$), and the backward substitution using the upper triangular matrix ($\mathcal{U}$).

For each of these calculations, the method begins in one corner and proceeds to the next element, which is dependent on, say, the elements directly adjacent in the negative Cartesian directions (see figure 1). The forward and backward substitutions have a similar dependency chain. This *wavefront* parallelism is not amenable to efficient distributed computing because of the time a process is idle waiting for the wavefront to reach it. To overcome this, Reeve et al. (2001)

developed a scheme in analogy to the red-black Gauss-Seidel iterative solver. With the red-black scheme during the forward and backward calculations, the domain is envisioned as a checkerboard and the SIP calculation is performed on all of the "red" points and then repeated for all of the "black" points (see figure 1).

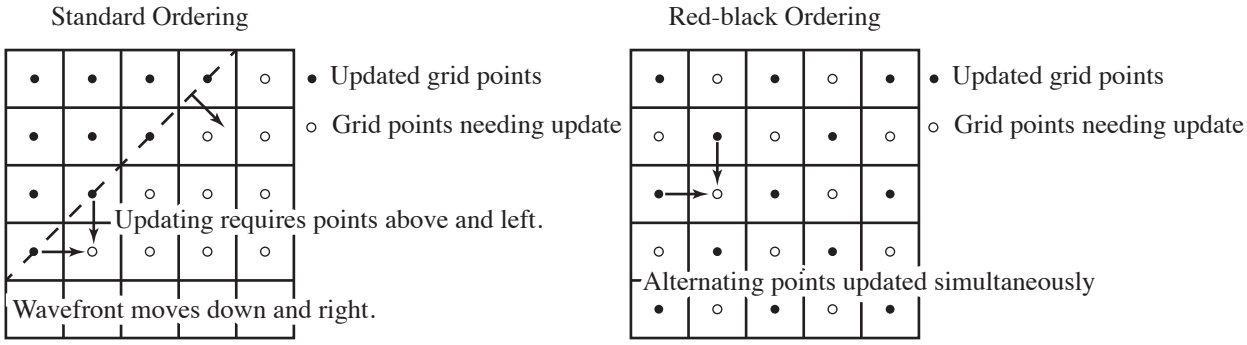

Figure 1. Calculation ordering for the standard SIP and for red-black ordering (from Reeve et al. [2001]) .

Dividing the domain using a Cartesian MPI decomposition in combination with red-black ordering is worthwhile, because now each process can perform a significant amount of computation before requiring edge communication with the neighboring processes. The red-black ordering is applied to the forward and backward substitution phases, calculating all of the red points can happen without parallel communication. Inter-process communication with neighboring processes exchanges the updated red points, and then the black points are calculated again independently. Unfortunately, the $\mathcal{LU}$-factorization phase cannot be reordered this way; the wavefront parallelism of the $\mathcal{LU}$-factorization significantly constrains the scalability of the system.

2.1 Design of BMF

The BMF design loosely follows the Model-View-Controller (MVC) design pattern. While some design considerations have been incorporated to ease the implementation of future features, BMF is primarily concerned with currently needed functionality. For example, view classes to create graphical user interfaces (GUIs) have not been created. All current classes in BMF can be divided into model classes, which are the data structures and associated operations on this data, and controller classes, which contain much of the program logic. The framework also makes significant use of the delegate design pattern, which uses helper objects to complete specific tasks. For example, one of the fundamental model classes is the `RealVariable` class; it contains an array of floating-point numbers and associated metadata. A `RealVariable` object (an instance of the class) can have a `MPIController` object attached, which will be responsible for

all inter-process communication. Attaching a different `MPIController` object allows for communication to occur between different sub-groups of processes. In addition, extensive use of polymorphic variables, inheritance, and method overloading help to overcome some of the peculiarities of object-oriented design in Fortran 2008.

Several abstractions have been incorporated into BMF to reduce errors and increase maintainability and flexibility. For example, the ordering of indices for multi-dimensional arrays can be dynamically configured (currently at compile time). This flexibility was included because an algorithm may require certain index ordering to efficiently use processor cache. Efficient cache utilization is critical in demanding computational applications. MPI Cartesian decompositions add additional complexity, because the ordering of directions for process coordinates is not required to match the ordering of directions for multi-dimensional arrays. Objects of `coordinate` class are used to describe grid-associated coordinates.

An `ArrayBounds` object describes a contiguous array with a lower and upper bound for each grid direction. Multiple `ArrayBounds` objects can be combined describe a more complicated region using an `ArrayRegion` object. The most common application of `ArrayRegion` objects is to facilitate the overlapping of calculation and MPI communication by performing a calculation over an inner region while the MPI communication completes, then repeating the calculation along the boundaries.

An additional convenience class is the `GroupedComController` class, which allows for multiple variable objects to be grouped together for the purpose of MPI communication. Often, in CFD applications the same inter-process communication is performed on multiple variables at the same time. By grouping the variables into a single buffer, a smaller number of larger messages can be sent rather than many small messages.

2.2 Performing Serial and Parallel Calculations with Battlefield Environment Division Modeling Framework

When using `RealVariable` objects, there can be a performance hit when accessing the variable directly through the derived-type (e.g., `variable%data`). Instead, point a declared contiguous pointer to the variable data, as shown in listing 1. The dynamically determined array index ordering means that, in general, loops should not need to tie a specific direction to an index. The ordering within a loop construct (such as `do concurrent`) should proceed in the most efficient from a processor cache perspective. Because Fortran uses column-major array ordering, it means the inner loop index should correspond to the first index in the array to prevent striding in the memory structure. When an index in a particular direction is required, using the intrinsic

`dot_product` function combined with a unit vector defined at compile-time provides a low impact method to extract the value (see listing 1).

For a calculation that requires MPI communication, the calculation can be divided into pieces and non-blocking MPI communication initiated so that the calculation is performed over the regions not dependent on neighbor-process data, while the communication is completing. Convenience routines exist to facilitate this kind of division of a calculation.

The example code shown in listing 2 can be incorporated into a function that takes an `ArrayRegion` object as an argument. Performing the parallel calculation, would then proceed as shown in listing 3. Within the `ArrayRegion` class definition, see section A-5 for convenience routines to facilitate the region decomposition.

2.3 Initiating and Completing MPI Communication

Directly interacting with the MPI environment is meant to be limited to initializing the MPI environment, creating process topologies (e.g., multidimensional Cartesian decompositions), and finalizing the environment. BMF allows the dimensionality and ordering of the directions of the MPI decomposition to set during compilation. The build system, CMAKE, is used to ensure proper dependencies. The actual number of processes used in each direction is runtime configurable. Other interactions, especially inter-process communication, are meant to be performed by higher-level classes, such as the `RealVariable`,`IntegerVariable`, or `GroupedComController` classes.

To reduce coupling between the `MPIController` class and other classes, the `MPIController` objects simply take a one-dimensional data array and perform the communication. The higher-level classes ensure proper buffering, data type, and array shape, and extract the appropriate array elements on both the *send* and *receive* sides of the communication. These abstracted operations are meant to be transparent to the developer allowing alternative strategies for efficient communication to be explored without major modifications to the source code using the higher-level classes—the ABLE model should not need to be significantly modified if the communication strategy shifts from custom data packing routines to derived data types when shifting to MPI-3.

The grid-associated details of `RealVariable` objects are internally tied to the array dimensions and, therefore, to the MPI decomposition dimensions. `RealVariable` objects support automatic internal buffering to allow continued access to interior grid points while edges are being updated via MPI communication. Routines, such as `receiveBoundary(...)` and

Listing 1. Performing a serial averaging operation in the x-direction on a 3-D variable with a weighting function that is only dependent on the x-coordinate.

```
! The declarations of the necessary varables are provided. Not all
! initialization operations are shown.
class(RealVariable) :: variable
real(RealKind), dimension(:,:,:), contiguous, pointer :: varPtr
class(RealVariable) :: weighting
real(RealKind), dimension(:), contiguous, pointer :: wghtPtr
class(RealVariable) :: theResult
real(RealKind), dimension(:,:,:), contiguous, pointer :: resPtr
type(ArrayBounds) :: calculationBounds
integer :: idx1, idx2, idx3, xidx
integer, dimension(NUM_GRID_DIMS) :: lbnd, ubnd
integer, dimension(NUM_GRID_DIMS) :: xUV

! The current version of Fortran compilers may have some difficulty
! in optimizing the pointer variables when they are contained within
! a derived type. Instead for use in a calculation, explicitly assign
! the data component of a RealVariable object to a pointer.
varPtr => variable%data
wghtPtr => weighting%data
resPtr => theResult%data

! set the upper and lower limits for the 3-D loop
lbnd = calculationBounds%lowerBound()
ubnd = calculationBounds%upperBound()

! use a unit vector to isolate a specific directional index
xUV = xxUnitVector

do concurrent (idx3=lbnd(3):ubnd(3), &
    idx2=lbnd(2):ubnd(2), idx1=lbnd(1):ubnd(1))
  ! several strategies were attempted to isolate a specific directional
  ! index using the intrinsic dot_product had the least performance
  ! impact.
  xidx = dot_product( (/idx1,idx2,idx3/), xUV )

  ! the xUV shifts the variable one point in the x-direction
  resPtr(idx1,idx2,idx3) = &
        wghtPtr(xidx)  * varPtr(idx1+xUV(1),idx2+xUV(2),idx3+xUV(3)) &
   + (1.-wghtPtr(xidx)) * varPtr(idx1,idx2,idx3)
end do
```

Listing 2. Performing the same x-averaging calculation using an array region.

```
! The declarations of the necessary varables are provided. Not all
! initialization operations are shown.
class(RealVariable) :: variable
real(RealKind), dimension(:,:,:), contiguous, pointer :: varPtr
class(RealVariable) :: weighting
real(RealKind), dimension(:), contiguous, pointer :: wghtPtr
class(RealVariable) :: theResult
real(RealKind), dimension(:,:,:), contiguous, pointer :: resPtr
type(ArrayRegion) :: calculationRegion
integer :: idx1, idx2, idx3, xidx, curPiece
integer, dimension(NUM_GRID_DIMS) :: lbnd, ubnd
integer, dimension(NUM_GRID_DIMS) :: xUV

! The current version of Fortran compilers may have some difficulty
! in optimizing the pointer variables when they are contained within
! a derived type. Instead for use in a calculation, explicitly assign
! the data component of a RealVariable object to a pointer.
varPtr => variable%data
wghtPtr => weighting%data
resPtr => theResult%data

! use a unit vector to isolate a specific directional index
xUV = xxUnitVector

! Loop over the number of pieces that make up the array region
do curPiece=1,calculationRegion%numberOfPieces()
  ! set the upper and lower limits for the 3-D loop
  ! based on the current piece
  lbnd = calculationRegion%lowerBound(piece=curPiece)
  ubnd = calculationRegion%upperBound(piece=curPiece)

  do concurrent (idx3=lbnd(3):ubnd(3), &
    idx2=lbnd(2):ubnd(2), idx1=lbnd(1):ubnd(1))

    ! get the index for the x-direction
    xidx = dot_product( (/idx1,idx2,idx3/), xUV )

    ! the xUV shifts the variable one point in the x-direction
    resPtr(idx1,idx2,idx3) = &
          wghtPtr(xidx)  * varPtr(idx1+xUV(1),idx2+xUV(2),idx3+xUV(3)) &
     + (1.-wghtPtr(xidx)) * varPtr(idx1,idx2,idx3)
  end do
end do
```

Listing 3. Performing the same x-averaging calculation using array regions and non-blocking MPI communication.

```
! define an array of ArrayRegion objects to
! hold the inner and outer regions
type(ArrayRegion), dimension(:), pointer :: splitRegions

! Begin the parallel communication.
call variable%beginSynchronizeBoundary(direction=XX)

! Use a convenience routine to split the calculationBounds into
! inner and outer parts. The result will be saved in the
! array {\tt splitRegions}; the regions can be extracted using
! the defined constants: InnerRegion and OuterRegion.
call splitInnerAndOuterRegions( &
                    inputArrayBounds=calculationBounds, &
              widthOfLowerOuterRegion=Coordinate(XX=1,YY=0,ZZ=0), &
              widthOfUpperOuterRegion=Coordinate(XX=1,YY=0,ZZ=0), &
                    separatedRegions=splitRegions )

! perform the inner calculation first, then wait for the MPI
! communication to complete.
do curRegion = InnerRegion, OuterRegion
  if (curRegion == OuterRegion) then
    call variable%endSynchronizeBoundary(direction=XX)
  end if

  call averageFunction(    region=splitRegions(curRegion), &
                         variable=inputVariable, &
                        weighting=X_weightFactor, &
                          average=averagedVariable   )
end do

! clean up the allocated regions to avoid a memory leak.
! clear regions is required because of a finalization bug
! in the Intel Fortran compiler
call clearRegions( splitRegions )
deallocate( splitRegions )
```

`packAndSendBoundary(...)` allow fine-grained control of the MPI sends and receives– useful in the Strongly Implicit Solver; but also convenience routines, such as `beginSynchronizeBoundary(...)`, which automatically pair the appropriate MPI sends and receives.

MPI communications require arguments based on the physical grid rather than the MPI decomposition to allow the MPI configuration to be decoupled from the grid configuration. If a communication is initiated or finalized in a physical direction that is not decomposed, then either nothing is done if the domain is not periodic in that direction or the boundaries are synchronized without buffering and no communication is initiated. See listing 4 for an example initiating the environment and using the convenience routines and listing 5 for example procedure calls to initiate and complte MPI communications.

Listing 4. Procedure to initialize the MPI environment and to initiate and complete various types of communication.

```
class(MPIController) :: world
class(MPICartesianController) :: cartCtrl

call world%initMPIworld() ! init MPI environment

! Create a periodic domain in both directions and divide into
! three subdomains.
call cartCtrl%init( parentController=world, &
                    numProcsInDir=Coordinate(XX=1,YY=3), &
          periodicInDir=TRUE_COORD )

! Initiate MPI sends in both the positive and negative
! y-directions.
call aVariable%beginSynchronizeBoundary(direction=YY)

! Initiate a send in the positive x-direction. Because
! the x-direction is not decomposed (only one process in
! the decomposition) and the domain is periodic, the lower
! boundary is immediately set to the values of the upper
! boundary.
call aVariable%beginSendUpperEdgeUpward(direction=XX)

! By ending the communication, the program will check
! if the communication is finished, if not it will wait
! until it is.
call aVariable%endSynchronizeBoundary(direction=YY)

! The end call here is still required because the x-decomposition
! is runtime configurable.
call aVariable%endSendUpperEdgeUpward(direction=XX)
```

Listing 5. Example procedure calls to initiate and complete a MPI communications.

```
class(RealVariable), pointer :: var1, var2 ! several real variables
class(MPICartesianController), pointer :: cartCtrl ! Cartesian controller
class(GroupedComController), pointer :: commCtrl

! Initialize the Grouped communication to use the
! appropriate MPI controller.
call commCtrl%init(mpiController=cartCtrl)

! add variables to the GroupedComController
call commCtrl%addVariable( inVariable=var1 )
call commCtrl%addVariable( inVariable=var2 )

! begin synchronizing process boundaries for var2 and var3 in
! x-direction
call commCtrl%beginSynchronizeBoundary(direction=XX)

! end synchronization processes
call commCtrl%endSynchronizeBoundary(direction=XX)

! clear the variables from the communication controller
call commCtrl%reset()
```

Often, prior to a calculation, multiple variables require MPI communication. In general, it is more efficient to send fewer, larger messages rather than many small messages. Rather than performing MPI communication individually, an BMF provided class, `GroupedComController`, allows for multiple variables to be grouped into a single message. Communication is initiated by the appropriate type-bound procedure, such as beginSynchronizeBoundary() (see section A-9 in the appendix). The object will then take care of buffering the edge(s) being sent to neighboring processes. Reduction operations for multiple variables may also be bundled together. Because of the larger amount of data involved, gathering operations are primarily done separately.

3. Results

The results of the MPI version of ABLE remain in good agreement with the laboratory data of Prasad and Koseff (1989), as shown in figure 2. Switching to the red-black version of SIP does not change the accuracy of the simulation, and enables more efficient parallel calculation. The initial, serial version of the steady-state ABLE model (Wang et al., 2012) was efficient and

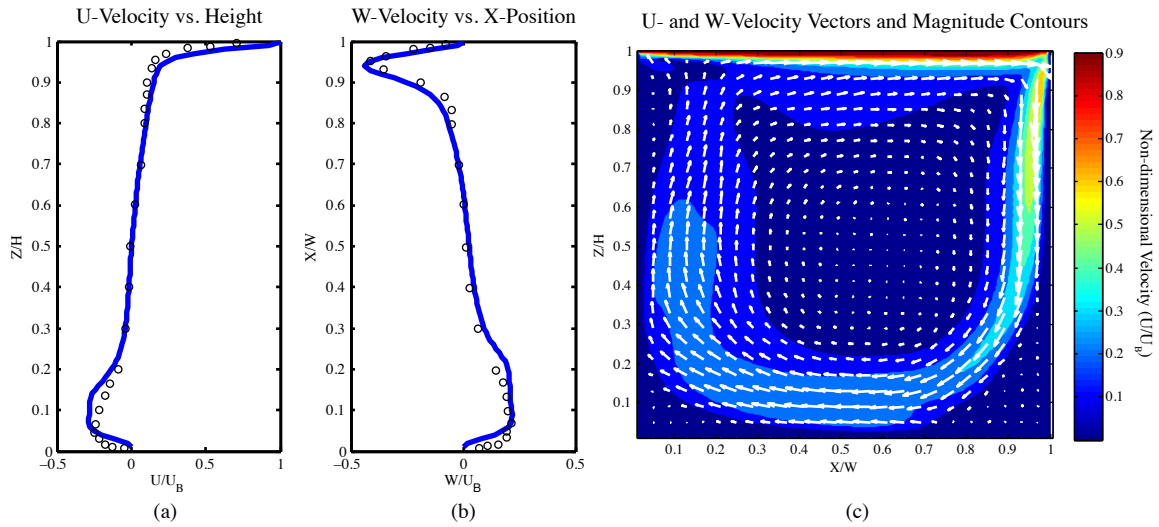

Figure 2. Results from ABLE (blue line) with red-black ordering compared with laboratory data (circles) (Prasad and Koseff, 1989). (a) Vertical profile of non-dimensional, x-component of the velocity. (b) Horizontal profile of the vertical component of the velocity scaled by the velocity at the top of the cavity. (c) Vertical slice through the center of the domain, showing the non-dimensional magnitude and direction of the x- and z-velocity components .

compact; changing the implementation to use BMF results in a decrease in performance due to the added overhead of the framework and the removal of some optimizations that were less effective in parallel. This degree of performance decrease is platform dependent; using the Intel Fortran Compiler on new Intel hardware, results in about a 50% increase in runtime for serial codes with optimization. Figure 3 show the results of limited scalability tests normalized by the runtime of the non-BMF version of ABLE. The first plot is the traditional division of a computational domain among more and more processes, while the second plot shows the results of expanding the domain to maintain constant domain size per process.

The MPI version of ABLE contains an algorithm that is not especially amenable to parallel computation, and this is reflected in figure 3. The wavefront parallelism in the $\mathcal{LU}$-factorization part of SIP, which must be repeated four times per iteration, reduces scalability. However, even with the reduced scalability, the addition of one extra process will allow each iteration to complete more quickly than for the serial version.

The red-black ordering within SIP enabled better parallel performance; however, the new ordering interacted with the outer iterative scheme, the SIMPLE method, for pressure-velocity coupling. The result is a slowing convergence of the outer iterations. The red-black scheme excites a 2-Δx wave in the pressure, an already ill conditioned matrix, which results in noise in the velocity field. Without care, this wave will cause the scheme to become unstable after a

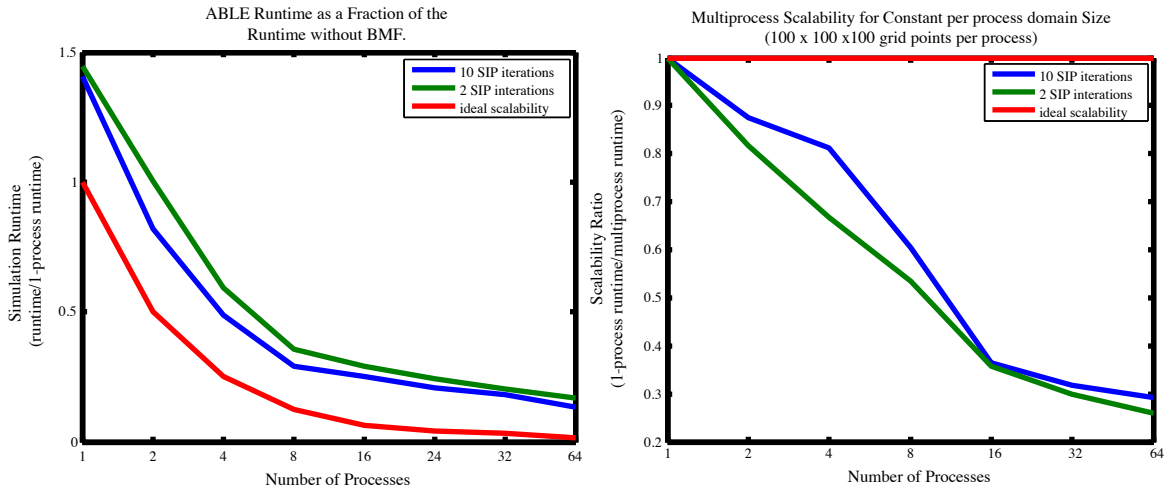

Figure 3. Scalability results for the red-black SIP-based SIMPLE integration scheme. (a) Normalized ABLE runtime compared with ideal scalability by dividing a constant-size domain over an increasing number of processes. (b) Scalability based on keeping the number of grid points per process constant. Adding processes, requires increasing the total computational domain size. All runtimes are normalized by the original serial version of ABLE without the overhead associated with including BMF .

couple hundred SIMPLE iterations.

The easiest way to contain the instability is to increase the number of SIP iterations in when calculating each velocity component. These matrices are well formed and will converge quickly. The pressure matrix, however, is ill conditioned; performing additional SIP iterations actually excites instability requiring the number of SIP iterations to remain one. A slightly more involved method is to ensure that for each SIMPLE iteration each application of SIP alternates between calculating the red points or the black points first. The resulting behavior still converges more slowly than when using the regular, serial SIP, but the model remains stable. The convergence behavior of the SIMPLE iterations is shown in figure 4. Again the noise introduced by the red-black ordering prevents the SIMPLE iterations from converging to the same value as the serial version. Given the larger number of iterations for convergence, using the parallel version of the steady-state ABLE model currently requires four processors to show a runtime improvement over the serial version.

Within the current implementation, a thorough attempt at optimization could result in some gains in performance, especially within BMF itself. In addition, switching to a different scheme for the pressure-velocity coupling, such as the semi-implicit method for pressure-linked equations revised (SIMPLER) method, may exhibit faster convergence (Versteeg and Malalasekera, 2007). Alternative sparse matrix solvers more amenable to MPI decomposition than SIP, such as

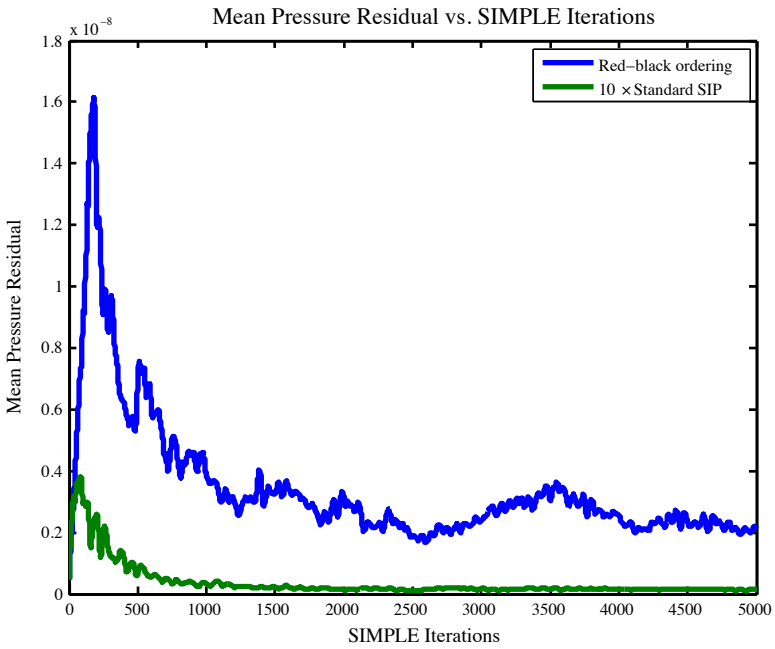

Figure 4. Rates of convergence for the serial version of the ABLE model using the standard SIP and for the red-black ordered ABLE model .

alternating direction implicit (ADI), provide additional avenues. These alternatives have not yet been explored within ABLE, as attention has shifted to adding a time-dependent integration scheme.

4. Conclusions

BMF was designed and implemented to ease development and maintenance of the ABLE model and provide single conceptual interface for performing many of the tasks often repeated in atmospheric modeling and numerical analysis. By encapsulating related data into classes and abstracting common operations, the object-oriented framework provides data structures and associated procedures that efficiently implements MPI without requiring the programmer to repeatedly delve into the error-prone minutiae of MPI. Furthermore, by reducing the flexibility to common use-scenarios, BMF provides a system by which communication and calculation can overlap using more reliable and easily understandable procedure calls.

Parallelization of the SIMPLE iterative scheme with SIP used a red-black scheme similar to the red-black Gauss-Seidel solver. The results show good agreement with laboratory data and with

the previous series version of ABLE. The $\mathcal{LU}$-factorization of the sparse operator matrix retains wavefront parallelism; however, the forward and backward substitution phases can be performed simultaneously with only edge communications between steps. An issue can occur when applying the red-black SIP in the SIMPLE scheme, because the red-black calculation results in a 2-Δx wave causing computational instability. By changing ordering of the red-black calculation on a per-iteration basis, the short wave appears to be controlled without the need for smoothing. The instability can also be controlled by increasing the number of inner iterations for the three momentum fields, but not the pressure calculation because in SIMPLE the pressure matrix is ill conditioned. Other methods for solving the pressure-velocity coupling should be explored, such as SIMPLER, and for better parallel scalability, other sparse matrix solvers could be implemented.

Benchmarking results against the original, serial version of ABLE, shows a performance penalty of around 50% due to BMF overhead and the removal of certain optimizations that were not MPI friendly. However, the longer runtime is easily compensated by adding a second process. The overall convergence properties of the red-black SIP are slower than the serial version of ABLE; the steady-state residuals for the pressure, which is always larger than the momentum fields, oscillate around a larger value, and reach equilibrium in at least double the number of time steps. Part of this behavior may be attributed to the interactions between the 2-$\Delta\ x$ wave and the pressure gradient calculation. A possible solution may be to use a different scheme for the pressure-velocity coupling; for example, the SIMPLER method does away with the need for a correction of the pressure field by introducing a more complete pressure equation (Versteeg and Malalasekera, 2007). The scheme requires additional computational resources, but exhibits better convergence properties and may be more amenable to the red-black SIP. Replacing SIMPLE with SIMPLER was not attempted at this time; priority was instead given to implementing a new time-dependent integration scheme.

References

Wang, Y.; Williamson, C.; MacCall, B. *A Description of the Framework of the Atmospheric Boundary Layer Environment (ABLE) Model*; Technical Report ARL-TR-6177, U.S. Army Research Laboratory: Adelphi, MD, 2012.

Stone, H. L. *SIAM J. Numerical Analysis* **1968**, *5*, 530–558.

Reeve, J. S.; Scurr, A. D.; Merlin, J. H. *Concurrency and computation: practice and experience* **2001**, *13*, 1049–1062.

Prasad, A. K.; Koseff, J. R. *Physics of Fluids A* **1989**, *1*, 208–218.

Versteeg, H. K.; Malalasekera, W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*, 2nd ed.; Pearson Prentice Hall, 2007.

Appendix. Battlefield Environment Division Modeling Framework – Fortran 2008 Interface

The public interface (i.e., neglecting internally used data and routines) of BMF v0.85 is now described. The framework makes extensive use of Fortran pointers to avoid significant data duplication and performance issues associated with the creation of array temporaries. The framework seeks to emphasize code readability and self-documentation (i.e., a code fragment that does not need comments to explain its purpose) above performance except in the case of large or deep loops. The following conventions and priorities are employed:

- To aid in code readability and self-documentation, more descriptive routine and variable names are preferred to shorter, generic names.
- When calling functions, a named argument list is preferred to an unnamed argument list. Both are acceptable Fortran, but named lists are more readable and less prone to erroneous arguments, especially if the compiler is unable to detect the error (e.g., mixing the order of two integer arguments).

```
! preferred
call anObject%createArray(size=ArraySize, name="name")

call anObject%createArray(ArraySize, "name")
```

- A pointer that has been set using '=>' should never be deallocated instead it should be nullified. This convention is followed by BMF and should be followed when using BMF.

```
anExamplePointer => anExampleObject%returnPointer()
...
nullify(anExamplePointer) ! proper convention

! using deallocate here will likely cause a segmentation
! fault later in the program
deallocate( anExamplePointer )
```

- A pointer that is passed, unallocated, as an argument into a routine and then set inside the routine, should be deallocated when no longer useful.

```
! the routine setPointer will allocate the object and assign it
! to aPointer
call anObject%setPointer( newPointer=aPointer )
...
deallocate( aPointer ) ! proper convention

! Using nullify will cause a memory leak and
! will eventually crash the program
nullify(aPointer)
```

A-1 GlobalParameters Module

The GlobalParameters module contains a few configuration parameters fixed by the pre-processor (CMAKE) during building. While other parameters are used internally, model codes using the BMF framework should only need the following constants:

- **NUM_GRID_DIMS** indicates the dimensionality of the model grid (for ABLE this will be three)
- **XX, YY, ZZ** These are integer constants that define the array index ordering (i.e., which index corresponds to which physical direction) and are used when a routine needs to know which direction it is operating on (e.g., `findArraySize(direction=XX)`).
- **xxUnitVector, yyUnitVector, zzUnitVector** These are integer arrays for use in calculations that require an offset in a physical direction. For example, the finite-difference, partial derivative with respect to x at coordinates (x, y) requires the value at $(x + 1, y)$ and $(x - 1, y)$. Using the unit vectors allows for compile-time flexibility in choosing (x, y) array ordering or (y, x) array ordering.
- **NUM_MPI_DIMS** indicates the number of dimensions used for the MPI decomposition (ABLE decomposes in all three dimensions).

A-2 LogicalCoordinate Class

A derived-type of type LogicalCoordinate class groups grid associated logical scalars into a single object for use in manipulating spatial vectors. The goal is to avoid using logical arrays, where

order is important, and instead improve clarity by specifying the directions explicitly. For example compare "periodicBoundary = (/ .True., .True., .False. /)" to "periodicBoundary = LogicalCoordinate(XX=.True., YY=.True., ZZ=.True.)".

The accessible derived-type components, type-bound procedures, and external routines are below:

- accessible components

  XX x-component logical

  YY y-component logical

  ZZ z-component logical

- type-bound procedures

  asArray() returns a logical array (size NUM_GRID_DIMS) with the components in appropriate order.

  setWithArray(array) sets the components to the contents of `array`.

  inDir(direction) gets the component corresponding to the proper direction constant.

  setInDir(direction, value) sets the component corresponding to `direction` to `value`.

- overloaded operators

  assignment (=) assigns either a logical constant or one `LogicalCoordinate` type to a `LogicalCoordinate` type.

  comparison equality (==) returns a `LogicalCoordinate` indicating whether elements of the corresponding elements of the left-hand side are equivalent to the right-hand side.

  compound (.and.) returns a LogicalCoordinate with the results of the binary `.and.` operator for each component of the left- and right-hand sides.

  compound (.or.) returns a LogicalCoordinate with the results of the binary `.or.` operator for each component of the left- and right-hand sides.

- external procedures

  any(aLogCoord) returns the logical scalar `.True.` if any of the components are true.

  all(aLogCoord) returns the logical scalar `.True.` if all of the components are true.

- Defined parameters

  TRUE_COORD Constant `LogicalCoordinate` with all components set to true.

  FALSE_COORD Constant `LogicalCoordinate` with all components set to false.

The "asArray" and "setWithArray" routines are designed to be used with other procedures that expect or return properly ordered integer arrays.

A-3 Coordinate Class

Similar to the previously defined LogicalCoordinate class, the Coordinate class defines an order independent, vector of integers related to grid directions.

- accessible components

  XX x-component integer

  YY y-component integer

  ZZ z-component integer

- type-bound procedures

  asArray() returns an integer array (size NUM_GRID_DIMS) with the components in appropriate order.

  setWithArray(array) sets the components to the contents of the integers in `array`.

  inDir(direction) gets the component corresponding to the proper direction constant.

  setInDir(direction, value) set the component corresponding to `direction` to `value`.

  setConditionally(condition, trueValue, falseValue) takes a LogicalCoordinate `condition` and sets the components based on logical tests. `trueValue` and `falseValue` can be scalar integers, (properly ordered) integer arrays, or `Coordinate` types.

- overloaded operators

  assignment (=) assigns a `Coordinate` type or scalar integer to the left-hand side. If it is a scalar integer, all components are set to the same value.

  arithmetic (+) returns a `Coordinate` type with the sum of the individual components of two `Coordinate` types or of one coordinate type and a scalar integer.

arithmetic (-) returns a `Coordinate` type with the difference of the individual components of two `Coordinate` types or of one coordinate type and a scalar integer.

arithmetic (*) returns a `Coordinate` type with the product of the individual components of two `Coordinate` types or of one coordinate type and a scalar integer.

arithmetic (/) returns a `Coordinate` type with the quotient of the individual components of two `Coordinate` types or of a coordinate type divided by a scalar integer.

comparison equality (==) returns a LogicalCoordinate with the results of component by component tests for equality between two `Coordinate` types or a `Coordinate` type and an integer.

comparison equality (/=) returns a LogicalCoordinate with the results of component by component test for not equal between two `Coordinate` types or a `Coordinate` type and an integer.

comparison equality (<) returns a LogicalCoordinate with the results of component by component test for inequality between two `Coordinate` types or a `Coordinate` type and an integer.

comparison equality (<=) returns a LogicalCoordinate with the results of component by component test for inequality between two `Coordinate` types or a `Coordinate` type and an integer.

comparison equality (>) returns a LogicalCoordinate with the results of component by component test for inequality between two `Coordinate` types or a `Coordinate` type and an integer.

comparison equality (>=) returns a LogicalCoordinate with the results of component by component test for inequality between two `Coordinate` types or a `Coordinate` type and an integer.

- external procedures

mod(coord_A, coord_B) returns a `Coordinate` type with the result of the components of `coord_A` modulus `coord_B`.

max(coord_A, coord_B) returns a `Coordinate` type with the maximum of the individual components of `coord_A` and `coord_B`.

min(coord_A, coord_B) returns a `Coordinate` type with the minimum of the individual components of `coord_A` and `coord_B`.

See listing A-1 for example usage.

Listing A-1. Basic example showing usage of Coordinate and LogicalCoordinate types.

```
type(Coordinate) :: coord_1
type(Coordinate) :: coord_2
type(LogicalCoordinate) :: logCoord_1
logical :: test

coord_1 = Coordinate(XX=1, YY=-2, ZZ=3)
coord_2 = 2 * coord_1  ! value will be (XX=2, YY=-4, ZZ=6)

! value will be (XX=.False., YY=.True., ZZ=.False.)
logCoord_1 = coord_2 < coord_1

test = any(logicalCoordinate) ! returns .true.
test = all(logicalCoordinate) ! returns .false.

! sets coord_2 to the same as above, but using a loop over the
! different directions. This is more useful in a subroutine or function
! that takes the direction as an argument.
do curDir=1, NUM_GRID_DIMS
  call coord_2%setInDir( direction=curDir, &
                         value=2*coord_1%inDir(curDir) )
end do
```

A-4 ArrayBounds Class

The ArrayBounds class defines a simple region for a multidimensional array. For each dimension, the lower bounds (default to 1) and upper bounds need to be specified.

- No accessible components.
- type-bound procedures

  lowerBound([direction]) If direction is not present, `lowerBound()` returns an integer array with the lower bound. If `direction` is specified, the integer lower bound in that specific direction is returned.

  lowerBoundAsCoordinate() returns a `Coordinate` type with the lower bound.

  upperBound([direction]) If direction is not present, `upperBound()` returns an integer array with the upper bound. If `direction` is specified, the integer upper bound in that specific direction is returned.

upperBoundAsCoordinate() returns a `Coordinate` type with the upper bound.

setLowerBound([direction], newBound) sets the lower bound to `newBound`. If `direction` is included, `newBound` should be an integer. If not, then `newBound` can be an integer array or `Coordinate` type.

setUpperBound([direction], newBound) sets the upper bound to `newBound`. If `direction` is included, `newBound` should be an integer. If not, then `newBound` can be an integer array or `Coordinate` type.

upLoBounds(direction) return a two-element integer array with the lower bound in `direction` as the first element and the upper bound as the second.

setUpLoBounds(direction, newBounds) sets the lower and upper bound in the specified direction using `newBounds`, a two-element integer array.

setUpLoBounds(newLowerBound, newUpperBound) sets the lower and upper bounds using `newLowerBound` and `newUpperBound`. These can be integer arrays or Coordinate types.

size([direction]) If `direction` is specified, `size` returns the number of elements in the specified direction. If `direction` is not specified then, `size()` returns the total number of elements in the array.

shape() returns an integer array with the total number of elements in each grid direction.

shapeAsCoordinate() returns a `Coordinate` type with the total number of elements in each grid direction.

addToLowerBound([direction], change) If direction is not present, add `change` to the lower bound. `change` can be a scalar integer (applied to all components), integer array, or a `Coordinate` type. If `direction` is specified, then `change` must be an integer and it is only applied in the specified direction.

addToUpperBound([direction], change) If direction is not present, add `change` to the upper bound. `change` can be a scalar integer (applied to all components), integer array, or a `Coordinate` type. If `direction` is specified, then `change` must be an integer and it is only applied in the specified direction.

shiftBoundsInDirection(direction, change) adds `change` to the lower and upper bounds in the specified `direction`.

- overloaded operators

assignment (=) assigns to an ArrayBounds type. Right-hand side can be either another ArrayBounds or an integer, which is copied to all components.

arithmetic (+) returns an ArrayBounds with the sum of the two ArrayBounds objects.

arithmetic (-) returns an ArrayBounds with the difference of the two ArrayBounds objects.

comparison (==) returns a scalar logical if both the lower and upper bounds of two ArrayBounds objects are equal.

comparison (/=) returns a scalar logical if any of the lower and upper bounds of two ArrayBounds objects are not equal.

comparison (<) returns true if the left-hand side array would fit inside or share some of the boundaries of the right-hand side array, but they are not equal.

comparison (<=) returns true if the left-hand side array would fit inside or is equal to the boundaries of the right-hand side array.

- No external procedures

See listing A-2 for example usage.

Listing A-2. Basic example showing usage of ArrayBounds type.

```
type(ArrayBounds) :: array_1
type(ArrayBounds) :: array_2
integer :: numberOfElements
type(Coordinate) :: arrayShape
type(Coordinate) :: upperBound
type(Coordinate) :: lowerBound

call array_1%setBounds( newLowerBound=Coordinate(XX=0, YY=1, ZZ=-1), &
                        newUpperBound=Coordinate(XX=3, YY=3, ZZ=3) )

numberOfElements = array_1%size( XX ) ! returns 4

numberOfElements = array_1%size() ! returns 60

! The below returns Coordinate(XX=4, YY=3, ZZ=5)
arrayShape = array_1%shapeAsCoordinate
```

A-5 ArrayRegion Class

The ArrayRegion class describes a piecewise-defined array space. An ArrayRegion object's primary purpose is to easily separate a simple domain into multiple regions when only part of the domain is ready to be calculated (e.g., other parts are waiting on MPI communication to

complete). An ArrayRegion object is a container object holding multiple ArrayBounds objects as *pieces* of the array region. Pieces can be added, removed or modified directly.

- No accessible components.
- type-bound procedures

  numberOfPieces() returns the number of ArrayBounds pieces currently defined in the region.

  addPiece([newPiece, newLowerBound, upperBound]) The `addPiece` subroutine can take either a single ArrayBounds object `newPiece` or *both* `newLowerBound` and `newUpperBound` arguments that are either integer arrays or `Coordinate` types.

  removePiece([pieceIndex, theArrayBounds]) remove piece will remove an ArrayBounds object from the region either by passing its `pieceIndex` or by removing all regions that are equal to a passed ArrayBounds object.

  reset() reset the array region to have zero pieces.

  pieceBounds(piece) return the ArrayBounds object that corresponds to index `piece`.

  lowerBound(piece, [direction]) return the lower bound of the ArrayBounds object with index `piece` as an integer array. If `direction` is specified a scalar integer is returned.

  lowerBoundAsCoordinate(piece) return the lower bound of the ArrayBounds object that corresponds to index `piece` as a `Coordinate` type.

  upperBound(piece, [direction]) return the upper bound of the ArrayBounds object with index `piece` as an integer array. If `direction` is specified a scalar integer is returned.

  upperBoundAsCoordinate(piece) return the upper bound of the ArrayBounds object that corresponds to index `piece` as a `Coordinate` type.

  setLowerBound(piece, [direction], newBound) sets the lower bound of the ArrayBounds object at index `piece` with an integer array or a `Coordinate` type. If `direction` is specified, then `newBound` should be a scalar integer.

  setUpperBound(piece, [direction], newBound) sets the upper bound of the ArrayBounds object at index `piece` with an integer array or a `Coordinate` type. If `direction` is specified, then `newBound` should be a scalar integer.

setBounds(piece, [newBounds, newLowerBound, newUpperBound]) replace the ArrayBounds object at index `piece` with the ArrayBounds object `newBounds`, or replace with an ArrayBounds object defined by `newLowerBound` and `newUpperBound` which can be either integer arrays or Coordinate types.

- No overloaded operators.
- external procedures

splitIntoInnerAndOuterRegions(inputArrayBounds, widthOfLowerOuterRegion, widthOfUpperOuterRegion, separatedRegions) This is a convenience routine that takes an ArrayBounds object (`inputArrayBounds`) and separates it into an inner and outer region, based on the passed Coordinate types `widthOfLowerOuterRegion` and `widthOfUpperOuterRegion`. The result is an allocatable array of ArrayRegion objects (i.e., the argument passed into `separatedRegions` should have the following attributes: dimension(:), pointer. Accessing the array regions is done with the parameters **fullRegion, innerRegion**, and **outerRegion** for the undecomposed array region or the inner and outer regions.

splitIntoInnerAndOuterRegions(inputArrayBounds, widthOfOuterRegions, separateLowerEdge, separateUpperEdge, separatedRegions) This is a convenience routine that takes an ArrayBounds object (`inputArrayBounds`) and separates it into an inner and outer regions, based on the integer `widthOfOuterRegions` and the `LogicalCoordinate` types `separateLowerEdge` and `separateUpperEdge`, which indicate whether the edge in each direction is separated from the inner array region. The result is the type as above.

clearRegions(regions) Clear regions addresses a bug in the Intel Fortran compilers where final routines are not called if the derived type is part of an allocatable array. This means that the internal data structures are not properly deallocated when calling `deallocate(regions)`. Use this function before deallocating an array of regions.

- defined parameters

FullRegion Constant to get the full ArrayRegion when using the above convenience routines to split inner and outer regions.

InnerRegion Constant to get the inner ArrayRegion when using the above convenience routines to split inner and outer regions.

OuterRegion Constant to get the outer ArrayRegion when using the above convenience routines to split inner and outer regions.

See listing A-3 for example usage.

A-6 MPIController Class

The MPIController and MPICartesianController classes control MPI-based communication between processes. MPIController is built around a general MPI communicator with a few limited routines for probing information about the number of processes and the unordered gather and reduce operations. Every model using the BMF framework should have an instance of MPIController class that will be initialized with MPI_COMM_WORLD– a communicator representing all available processes. All model programs using BMF should call initWorld at the beginning of the program and finalizeWorld at the end. If the MPI library is not available, an option is available in the build options to turn off MPI. The code will run in series mode, and all routines will return sensible results for the series run. Values passed as arguments related to multiple processes will be ignored. See listing A-4 in section A-7 below for the most common calls to objects of type `MPIController` and the following `MPICartesianController`.

MPIController contains the following routines:

- No accessible components.
- type-bound procedures

  initWorld() initializes the MPI environment and creates a copy of MPI_COMM_WORLD for future use in the program. **This routine must be called before any other MPI related routines**

  finalizeWorld() cleans up the MPI environment before the program exits. **This routine must be called after the last MPI routine before the program exits.**

  communicator() returns a scalar integer with the MPI communicator handle.

  rank() Return a scalar integer with the current process rank.

  numberOfProcesses() returns a scalar integer indicating the number of processes in the communicator.

Listing A-3. An example showing usage of ArrayRegion type for a two-dimensional model.

```
type(ArrayBounds) :: anArrayBounds
type(ArrayRegion), dimension(:), allocatable :: splitRegions

⋮

! create an inner region that does not include 1-point side regions
! in the x-direction
call splitIntoInnerAndOuterRegions( &
                    inputArrayBounds=anArrayBounds, &
              widthOfLowerOuterRegion=Coordinate(XX=1,YY=0), &
              widthOfUpperOuterRegion=Coordinate(XX=1,YY=0), &
                    separatedRegions=splitRegions )

call mpi%beginSynchronizeBoundary(direction=XX)
do curRegion= InnerRegion, OuterRegion
  if (curRegion == outerRegion) then
    call mpi%endSynchronizeBoundary(direction=XX)
  end if

  numOfPieces = splitRegions(curRegion)%numberOfPieces()

  do curPiece=1, numOfPieces
    lowerBound = splitRegions(curRegion)%lowerBound(piece=curPiece)
    upperBound = splitRegions(curRegion)%upperBound(piece=curPiece)

    ! Note: for fortran first index should be in inner loop.
    do index2=lowerBound(2),upperBound(2)
      do index1=lowerBound(1):upperBound(1) )
        result(index1, index2) =   variable_1(index1, index2) &
                                 - variable_1(index1-1, index2)
      end do
    end do

  end do ! end loop over all pieces of curRegion

end do ! end loop over inner and outer regions

! clean up internal data in the regions
call clearRegions( splitRegions )
deallocate( splitRegions )
```

copy(copy) copies the communicator to a new communicator (in MPI terms the new communicator will refer to the same processes, but will have a separate context). `copy` should be an unassociated pointer to an MPIController. Because the pointer is set in an argument, the resulting copy should call finalizeMPIController() and then be deallocated (a bug in the Intel Fortran compiler prevents the use of Fortran's finalization routine).

split(color, [key], newController) splits the communicator into multiple communicators based on the scalar integer `color` as the criterion. The rank order of the processes assigned to a specific communicator can be customized using the optional integer argument `key`. Again, the new communicator must be an unassociated pointer of type MPIController, and should be finalized and deallocated when no longer needed.

splitAndLink(validColors, color, [key], localController, remoteController) splits the communicator into two communicators and links the two communicators by an intercommunicator. The separation criterion is the scalar integer `color`, and the valid values for color are passed as a two-element array. Intercommunicators only link two intracommunicators; thus, only two color values are allowed. The rank order of the processes assigned to a specific communicator can be customized using the optional integer argument `key`. Both the local and remote `MPIController` types must be unassociated pointers, and should be finalized and deallocated when no longer needed.

finalizeMPIController() frees the communicator and readies the controller for a separate deallocation.

gather(sendBuffer, [allBufferSizes], root, receiveBuffer) collects all values input via `sendBuffer` on a single process with rank `root`. `sendBuffer` can be a scalar or one-dimensional array of type integer or real(RealKind). The `receiveBuffer` should be an unallocated contiguous pointer of the appropriate type (integer or real(RealKind)). If the sendBuffers are not all equal size, then an integer array, `allBufferSizes`, is required to indicate the number of elements coming from each process in the communicator. The gathered values will be in order of process rank.

allGather(sendBuffer, [allBufferSizes], root, receiveBuffer) Similar to `gatherBuffer(...)`, but instead of collecting the values onto a single buffer, all processes receive a copy of the result.

reduce(buffer, operation, root) returns the result of a single operation on the values input via `buffer`, which can be integer, logical, or real(RealKind) and scalar or

one-dimensional array. Only process with rank equal to `root` will get the result, and all processes must have the same size buffers. The available operations are the following:

  - **MaxOp** Maximum value of all elements
  - **MinOp** Minimum value of all elements
  - **SumOp** Sum of all elements.
  - **ProdOp** Product of all elements
  - **AndOp** Logical AND operation
  - **OrOp** Logical OR
  - **XorOp** Logical XOR (exclusive or)

allReduce(buffer, operation) Similar to `reduceBuffer(...)` except all processes will receive the computed value.

- No overloaded operators.
- No external procedures.

A-7 MPICartesianController Class

The MPICartesianController is a subclass of MPIController. As such it inherits all of the above type-bound procedures. It also defines additional routines to allow for communication between the processes. A cartesian decomposition divides a rectangular grid into smaller rectangles. The term face is used to describe the sides of the decomposed rectangles. An argument named face takes either the constants `LowerFace` or `UpperFace`.

- No accessible components.
- type-bound procedures

init(parentController, numProcessesInDir, periodicInDir) creates a new communicator using the same processes in `parentController` but with a cartesian topology. The dimensions of the decomposition are specified in the `Coordinate` type argument `numProcessesInDir`. Each direction in the grid must be specified as either periodic or nonperiodic by passing the `LogicalCoordinate` type `periodicInDir`.

isPeriodic([direction]) returns a `LogicalCoordinate` of whether the grid is periodic. If the scalar integer `direction` is specified a logical scalar is returned.

coordinates([direction]) returns a `Coordinate` type (or scalar if integer `direction` is specified) of the current process' coordinates in the MPI topology. *Coordinates are zero based as is the MPI convention*. For directions that are undecomposed, the coordinate will always be zero.

coordinatesForRank(rank) returns the coordinates (as a Coordinate type) for the process with rank equal to `rank`.

shape([direction]) returns the number of processes in each direction (as a Coordinate type) or a scalar integer if integer `direction` is specified.

iamLowerEdge([direction]) returns a LogicalCoordinate (or scalar logical if integer `direction` is specified) with true values if the process is a lower edge process (i.e., coordinate in the direction is zero).

iamUpperEdge([direction]) returns a LogicalCoordinate (or scalar logical if integer `direction` is specified) with true values if the process is an upper edge process (i.e., coordinate in the direction is the number of processes in that direction minus one).

neighborExists(face, direction) returns a scalar logical true if the neighbor exists. `face` should be one of integer constants `upperFace` or `lowerFace`, and direction should be one of the integer direction constants.

nullRequest() returns a constant used to indicate a completed or non-existant request (a request contains status information for a non-blocking MPI communication operation). This is a constant specified by the MPI library, but if MPI is not available a different constant is used.

sendReceiveNeighbor(direction, sendface, sendBuffer, sendRequest, receiveFace, receiveBuffer, receiveRequest) send a one-dimensional array of data (`sendBuffer`) in the specified direction (positive direction for `upperFace` negative direction for `lowerFace`). Received data is stored in `receiveBuffer`. The send and receive requests should be queried before either the send or receive buffers are read or modified.

receiveFromNeighbor(direction, receiveFace, receiveBuffer, receiveRequest) receive a one-dimensional array of data (`sendBuffer`) from the specified direction (positive direction for `upperFace` negative direction for `lowerFace`). Calling this routine should be followed by a call to sendToNeighbor. This is useful when more fine-grained control over initiating sends and receives is required.

sendToNeighbor(direction, sendface, sendBuffer, sendRequest) send a one-dimensional array of data (`sendBuffer`) in the specified direction (positive

direction for `upperFace` negative direction for `lowerFace`). Calling this routine should have been preceeded by a call to receiveFromNeighbor. This is useful when more fine-grained control over initiating sends and receives is required.

waitForMPItoComplete(request) waits for a send or receive to complete. `request` can either be a scalar integer or an array of integers all of which must finish before the subroutine will return.

- No overloaded operators.
- No external procedures
- Defined parameters

  LowerFace Constant used to refer to the lower side of a decomposed piece of the domain.

  UpperFace Constant used to refer to the upper side of a decomposed piece of the domain.

The communication routines are not meant to be used directly in the model. Instead, the MPICartesianController acts as a delegate for other objects which will package the data properly for MPI communication. See listing A-4 for the most common calls to MPICartesianControllers.

A-8 RealVariable Class

The `RealVariable` class is a container for an array of real type. By connecting a reference to an `MPICartesianController` as a delegate, MPI communication is enabled. The object can maintain its own send and receive buffers and requests, to ensure communication has completed. If several variables will be performing MPI communications in the same direction at the same time, a `GroupedComController` should be used to decrease the number of messages being sent.

- accessible components

  data allows for performance optimization; the primary data pointer is made accessible. The variable component is a 3-D contiguous pointer of type real(RealKind).

- type-bound procedures

  initScalar([name], initialValue, [mpiController], [processDependent]) initializes the variable as a single element array with an initial value given by `initialValue`. If `mpiController` (a pointer of type `MPICartisianController`) is present, it

Listing A-4. Preparing a program to use MPI communication.

```
type(MPIController) :: world
class(MPICartesianController), pointer :: cartCtrl
type(Coordinate) :: decomposition
type(LogicalCoordinate) :: periodic
type(Coordinate) :: myCoords

call world%initWorld() ! required before other MPI calls

myRank = world%rank()  ! rank of each process in MPI_COMM_WORLD

! create a 2-D decomposition that is
! periodic in both directions
decomposition = Coordinate(XX=2,YY=3)
periodic = LogicalCoordinate(XX=.True.,YY=.True.)

allocate(cartCtrl)
call cartCtrl%init( parentController=world, &
                    numProcsInDir=decomposition, &
                    periodicInDir=periodic )

! processes may be reordered so myRank may not equal myNewRank
myNewRank = cartCtrl%rank()

myCoords  = cartCtr%coordinates()

! three processes will return true and three false
lowerEdge = cartCtrl%iamLowerEdge(XX)

call cartCtrl%finalizeMPIController() ! done using cartCtrl
deallocate( cartCtrl )

call world%finalizeWorld() ! required before program ends
```

will be attached to allow for MPI operations. The `processDependent` argument of type `LogicalCoordinate` is for parallel output purposes. If set to true in a direction, output will ensure that each process writes its copy of the variable (default is true).

init([name], [bounds, extent], [initialValue, initialSubroutine], [ghostPointWidth], [mpiController], [processDependent]) initializes the variable with size given by a `Coordinate` type `extent` or explicit array bounds given by `bounds` of type `ArrayBounds`. Initialization of the internal array can be set using the constant `initialValue` or by a function pointer passed into `initSubroutine`. If `mpiController` (a pointer of type `MPICartisianController`) is present, it will be attached to allow for MPI operations. `ghostPointWidth` adds additional layers of points to store data from neighboring processes. This argument is ignored if `mpiController` is not passed. The `processDependent` argument of type `LogicalCoordinate` is for parallel output purposes. If set to true in a direction, output will ensure that each process writes its copy of the variable (default is true).

copy([newBounds, ghostPointWidth], copy]) creates a copy of the RealVariable with different bounds based on the `newBounds` and `ghostPointWidth` arguments, or creates a full copy. `copy` is an unassociated pointer; as such, it should be deallocated when no longer needed.

name() returns a trimmed string containing the variable name.

setName(newName) sets the name of the variable to `newName`.

bounds() returns an ArrayBounds type with the local (decomposed) variable bounds. BMF uses the Fortran feature that allows arrays to begin with numbers other than one, so the local part of an array that has been spread across multiple processes will continue the element numbering from the neighbor.

boundsWithoutBoundaries() returns the ArrayBounds type of the local variable ignoring ghost points and lateral boundaries on processes along the edge of the domain for non-periodic boundary conditions.

boundsWithGhostPoints() returns the ArrayBounds type of the local variable including the ghost-point buffers used to store data from neighboring processes.

as1D() returns a one-dimensional pointer to the variable data. This serializes the 3-D array, and is primarily for variables that are actually one dimensional arrays (e.g., a $1 \times 1 \times 10$ array).

as3D() returns a 3-D pointer to the variable data.

asScalar() returns a scalar pointer to the variable data. Used when the variable array contains only a single element.

Communication routines

beginSynchronizeBoundary(direction) begins two separate communication operations with neighboring processes in the soecified direction sending the appropriate data to both the upper and lower neighboring processes. The variable keeps track of the MPI requests. Should be paired with an `endSynchronizeBoundary` in the same direction. Communications in separate directions can be in process simultaneously by calling this routine in each direction.

endSynchronizeBoundary(direction) ensures the MPI communication operations initiated by a `beginSynchronizeBoundary` call have completed and the resulting data is unpacked and saved to the appropriate ghost points.

beginSendLowerBoundaryDownward(direction) Similar to `beginSynchronizeBoundary`, this routine initiates a one-sided communication, sending the lower face in `direction` to the appropriate neighbor process.

endSendLowerBoundaryDownward(direction) Similar to `endSynchronizeBoundary`, this routine ensures the communication already initiated has completed and communication buffers are unpacked and deallocated.

beginSendUpperBoundaryUpward(direction) same as `beginSendLowerBoundaryDownward`, but sending the upper face to the appropriate neighbor process.

endSendUpperBoundaryUpward(direction) similar to `endSendUpperBoundaryUpward`, but sending the upper face to the appropriate neighbor process.

Additional Communication routines (for more fine-grained control over MPI communication). A single MPI communication should utilize all four subroutines.

receiveBoundary(recvFace, direction) creates receive buffers for an MPI communication and initiates a non-blocking MPI receive. Should be called before a corresponding `packAndSendBoundary`.

packAndSendBoundary(sendFace, direction) populates communication buffers

and initiates a non-blocking MPI communication with the appropriate neighbor process.

cleanupSendBoundary(sendFace, direction) waits for MPI send request to complete, and then cleans up send buffers. Both `receiveBoundary` and `packAndSendBoundary` should have been called, otherwise program will be stuck in an infinite loop.

unpackReceivedBoundary(recvFace, direction) waits for MPI receive request to complete, and then unpacks and deallocates the receive buffers. Both `receiveBoundary` and `packAndSendBoundary` should have been called already, else the program will be stuck in an infinite loop.

- No overloaded operators.
- No external procedures

See listing A-5 for example usage, including a basic calculation.

A-9 GroupedComController Class

While individual variables are capable of utilizing MPI for sharing data with separate processes. Because of the overhead required to initiate an MPI message, it is more efficient to send multiple variables with a single message. A `GroupedComController` uses pointer references to variables to create appropriately sized send and receive buffers, and to post MPI send and receive calls.

- No accessible components.
- type-bound procedures

init(mpiController, [numberOfVariables]) initializes the object and saves a reference to the passed `MPICartesianController` pointer. If the optional `numberOfVariables` is present, an array of pointers is pre-allocated.

addVariable(variable) adds the RealVariable pointer type `variable` to the array of RealVariable pointers.

clearVariables() deallocates the array of pointers containing RealVariable references.

Communication routines

Listing A-5. Example usage of the `RealVariable` class.

```
class(RealVariable), pointer :: scalarVariable
class(RealVariable), pointer :: variable_1D
class(RealVariable), pointer :: variable_3D
class(MPICartesianController), pointer :: cartCtrl
type(ArrayBounds) :: variableBounds
real(RealKind), pointer :: scalarValue
real(RealKind), dimension(:), contiguous, pointer :: array_1D
real(RealKind), dimension(:,:,:), contiguous, pointer :: array_3D

! create a scalar RealVariable that will have the
! same value on all processes
call scalarVariable%init(    name="a scalar", &
                        initialValue=2._RealKind, &
                   processDependent=FALSE_COORD )

! create a 1-D array with 5 elements in the z-direction
call variable_1D%init(  name="a 1-D array", &
                         extent=Coordinate(XX=1,YY=1,ZZ=5), &
                  initialValue=4._RealKind, &
             processDependent=FALSE_COORD )

! create a 3-D array with bounds=variableBounds
! that has ghost points along the boundaries.
call variable_3D%init(name="a 3-D array", &
                        bounds=variableBounds, &
                 initialValue=6._RealKind, &
        cartesianController=cartCtrl &
            ghostPointWidth=2 )

! a sample calculation by accessing the variable component
scalarValue => scalarVariable%asScalar()
array_1D => variable_1D%as1D() ! access data as a 1-D array
array_3D => variable_1D%as3D() ! access data as a 3-D array

! spread the values in oneDimArray to threeDimArray
! note reverse ordering of the loops to prevent memory striding
do concurrent (index3=1,5, index2=1,5, index1=1,5)
  ! zzUnitVector is Globally accessible
  zIndex = dot_product( (/ index1, index2, index3 /), zzUnitVector )

  array_3D(index1, index2, index3) = scalarValue * array_1D(zIndex)
end do
```

beginSynchronizeBoundary(direction) begins two separate communication operations with neighboring processes in the specified direction sending the appropriate data to both the upper and lower neighboring processes. The variable keeps track of the MPI requests. Should be paired with an `endSynchronizeBoundary` in the same direction. Communications in separate directions can be occurring simultaneously.

endSynchronizeBoundary(direction) ensures the MPI communication operations initiated by a `beginSynchronizeBoundary` call have completed and the resulting data is unpacked and saved to the appropriate ghost points.

beginSendLowerBoundaryDownward(direction) Similar to `beginSynchronizeBoundary`, this routine initiates a one-sided communication, sending the lower face in `direction` to the appropriate neighbor process.

endSendLowerBoundaryDownward(direction) Similar to `endSynchronizeBoundary`, this routine ensures the communication already initiated has completed and communication buffers are unpacked and deallocated.

beginSendUpperBoundaryUpward(direction) Same as `beginSendLowerBoundaryDownward`, but sending the upper face to the appropriate neighbor process.

endSendUpperBoundaryUpward(direction) Similar to `endSendUpperBoundaryUpward`, but sending the upper face to the appropriate neighbor process.

Additional Communication routines (for more fine-grained control over MPI communication). A single MPI communication should utilize all four subroutines.

receiveBoundary(recvFace, direction) creates receive buffers for an MPI communication and initiates a non-blocking MPI receive. Should be called before a corresponding `packAndSendBoundary`.

packAndSendBoundary(sendFace, direction) populates communication buffers and initiates a non-blocking MPI communication with the appropriate neighbor process.

cleanupSendBoundary(sendFace, direction) waits for MPI send request to complete, and then cleans up send buffers. Both `receiveBoundary` and `packAndSendBoundary` should have been called, otherwise program will be stuck in an infinite loop.

unpackReceivedBoundary(recvFace, direction) waits for MPI receive request to complete, and then unpacks and deallocates the receive buffers. Both `receiveBoundary` and `packAndSendBoundary` should have been called already; if not the program will be deadlocked.

- No overloaded operators.
- No external procedures

See listing A-6 for an example using the `GroupedComController`.

Listing A-6. An example, initializing and using a `GroupedComController`.

```
class(GroupedComController), pointer :: grpComm
class(MPICartesianController), pointer :: cartCtrl
class(RealVariable), pointer :: variable1
class(RealVariable), pointer :: variable2

! init with pointer reference to an
! MPICartesianController object
call grpComm%init( cartCtrl )

! add variable references
call commCtrl%addVariable( variable1 )
call commCtrl%addVariable( variable2 )

! Begin synchronize ghost points in the x-direction for both
! sides of the local variables.
call commCtrl%beginSynchronizeBoundary( direction=XX )

! ensure the communication is finished and that
! buffers are unpacked and cleaned up.
call commCtrl%endSynchronizeBoundary(direction=XX )
```

List of Symbols, Abbreviations, and Acronyms

3-D	three-dimensional
ABLE	Atmospheric Boundary Layer Environment
ADI	alternating direction implicit
BMF	Battlefield Environment Division Modeling Framework
CFD	computational fluid dynamics
CUDA	Compute Unified Device Architecture
GUI	graphical user interface
HDF5	Hierarchal Data Format version 5
MIC	many integrated core
MPI	Message Passing Interface
MVC	Model-View-Controller
NetCDF	Network Common Data Form
SIMPLE	semi-implicit method for pressure-linked equations
SIMPLER	semi-implicit method for pressure-linked equations revised
SIP	Strongly Implicit Procedure

NO. OF COPIES	ORGANIZATION
1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
3 (PDF)	US ARMY RESEARCH LAB ATTN RDRL CIE M B MACCALL ATTN RDRL CIE M Y WANG ATTN RDRL CIE M G HUYNH
1 (PDF)	GOVT PRNTG OFC A MALHOTRA