# The Message Passing Implementation of GISS modelE

T. Clune, R. deFainchtein, H. Oloso, and U. Ranawake
NASA Goddard Space Flight Center
Advanced Software Technology Group
Code 610.3
Greenbelt, MD 20771

June 5, 2005

**Abstract**

The message-passing implementation for modelE is presented. Details include the approach to domain decomposition, interfaces for new procedures, and discussions of special cases.

# Contents

# 1 Introduction

MODELE [7]is a well-known and highly effective general circulation model used to study various aspects of the Earth's climate including long term effects due to human activities. This application has been developed over many years by researchers at the Goddard Institute for Space Studies (GISS) and their collaborators, and currently consumes a substantial fraction of NASA supercomputing resources each year.

This document is primarily intended to serve as a developer guide for the message-passing parallel implementation of GISS MODELE, but also as a final report on the project which has produced this implementation. Although MODELE had previously been parallelized using OpenMP (shared-memory) directives, the inherent performance/portability limitations were often inadequate for some of the more demanding (e.g. higher-spatial resolution) computational experiments. In the Fall of 2003, a collaboration among the Advanced Software Technology Group (ASTG) [1], the Global Modeling and Assimilation Office (GMAO), and GISS was established to produce firstly a parallel implementation of MODELE suitable for running across nodes of a distributed-memory parallel platform and subsequently to use the recently developed Earth System Modeling Framework (ESMF) to produce a variant of MODELE which uses the Lin-Rood finite-volume (FV) dynamical core [5, 6, 3, 4]. The ASTG was to perform the majority of the software modifications, while the GMAO was to provide guidance based upon experience both with the FV core and with parallelization of similar models. GISS was to provide expertise on the existing implementation, and is primarily the customer for these activities.

Because MODELE is a constantly evolving instrument for scientific investigation with many and varied developers, special measures were required to ensure that the parallelization activity did not unduly impact software modifications related to scientific improvement of the model. In particular, software change procedures were created which enabled the default build-configuration (parallelization via OpenMP) to function correctly throughout the entire development period. Well-documented verification procedures were established to guarantee bit-wise reproducibility prior to merging changes

[1]For the majority of the duration of this activity, the ASTG was part of the Science Computing Branch, but has been relocated within the newly formed Software Integration and Visualization Office (SIVO) as part of an unrelated Goddard reorganization, effective in January 2005.

into the software repository. An interesting complication arose from the fact that some of the verification procedures are rather time-intensive and therefore interfered with the desired policy of frequent software commits. To minimize this inherent conflict, the ASTG created a separate repository to facilitate routine software changes/commits and performed merges of the two repositories on a less-frequent basis when full verification procedures could be performed. On the few occasions where the merge was thought to be quite significant, a request was made for a freeze in activity on the part of other developers. A secondary benefit of this approach was a reduction in the number of rather cumbersome interactions with GISS' computer security fire-wall.

Another important guiding principle for software modifications was to ensure that the pre-existing scientific developers of MODELE be able to readily recognize, maintain and modify essentially all parts of the delivered software. An all-too-frequent failure mode of projects such as the one described here is the development of a fully-functional, clean software application that is never adopted by the customer due to lack of familiarity. It must be emphasized, however, that there are significant costs associated with our approach. Such conservative software policies often increases the so-called "code-debt" - aspecs of the code that should be restructured to improve long-term maintenance and/or flexibility, but are deferred due to more immediate concerns. The trade-offs which must be considered are time-to-delivery and the number/frequency of future similar projects on the same code base. Fortunately, the familiarity issue does not arise for entirely new software modules, and the project afforded a greater deal of flexibility in the implementation of these.

Over the past decade, many teams of investigators have developed domain-specific parallelization tools/libraries, but most of these were never intended for external consumption. Rather than develop yet another suite, the implementation team decided to select from those that have some public support, and ultimately chose to use the Earth System Modeling Framework (ESMF), which was just beginning development but aligned well with several other activities. In hindsight, this decision was unfortunate and at this time relatively little functionality is obtained from the ESMF. The consequences of this decision will be discussed in more detail later in this document.

At this time, the MPI implementation of MODELE is able to generate correct results (i.e. bitwise identical results to those obtained from the serial build) on multiple architectures. For $2^o \times 2.5^o$ resolution grids, reasonable efficiency is obtained through 10's of processors, and later sections address

the possibility of increasing this scaling even further. Figure **??** shows the current performance of MODELEon the HP SC45 the SGI Altix.[2]

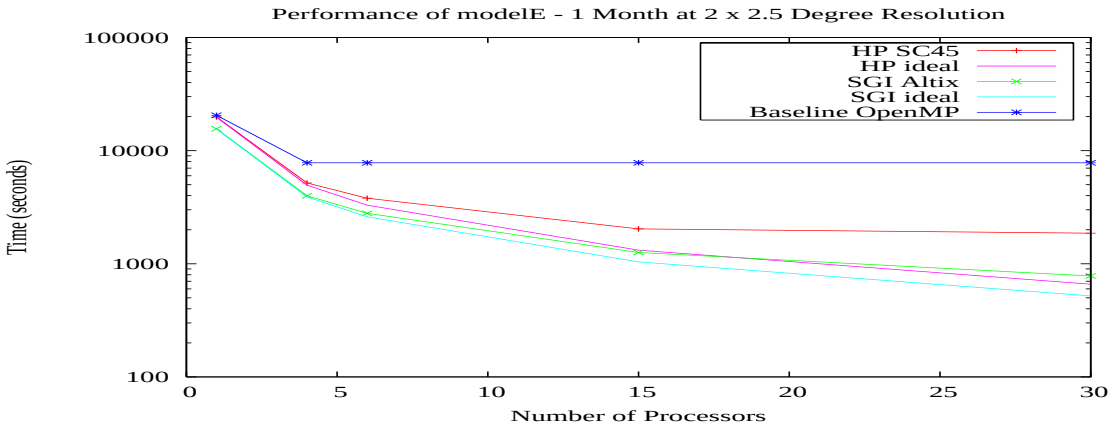


Figure 1: Time versus the number of processors for MODELE for the E1F20 ($2 \times 2.5^o$ resolution) on the HP SC45 (halem) and the SGI Altix (palm). For comparison the performance of the OpenMP implementation is also shown for the HP SC45. (OpenMP is restricted to 4 cpus on that architecture.)

# 2 Message Passing and Domain Decomposition

Although there are a variety of techniques to exploit parallel computing architectures, the technique of *domain decomposition* has proven to be the most portable and robust for almost all models which treat partial differential equations on large computational grids. The principle of domain decomposition rests on the mathematical foundation that in many instances, the solution of a set of equations on a given domain can be obtained from the solution of nearly identical sets of equations on a set of non-overlapping subdomains with appropriate boundary conditions. An efficient implementation of such a decomposition on a parallel architecture then simply consists of assigning each subdomain to a particular processor and adding appropri-

[2]The recent upgrade of halem has reduced the performance on 30 processors by 20%, and the ASTG is actively working to restore the performance.

ate, hopefully lightweight, mechanisms to deal with the boundaries between subdomains.

In terms of software/implementation, a model which exploits domain decomposition, should closely resemble its serial cousin with hopefully minor modifications. Typically identical copies of a suitable program are run on each processor of a parallel architecture but acting on different data. This is called "Single Program Multiple Data" (SPMD) paradigm. At the very least, the mechanism which launches the multiple copies of the program must also provide a means to distinguish which instance is which - usually a routine which returns an integer index known as the "rank" of the process. By means of the rank index, logic in the program can determine which subdomains are to be treated on that processor and which processors contain data required for handling boundaries.

Subdomains are generally represented by arrays of slightly greater extent than the actual sudbomain. The extra elements in these arrays are variously referred to as "halo", "ghost", or "guard" region and are used to store copies of data from neighboring subdomains. Such copies are essential for many numerical operations, and perhaps best exemplified by the computation of a first derivative by means of a second order finite-difference stencil. Data located entirely within the local domain is insufficient for computing such quantities at the edge of the local domain. In the simplest cases the width of the halo region is merely 1 element, but for higher-order stencils, deeper halos can be used. Most paradigms do not provide any mechanism for ensuring that such copies of data within the halo are consistent with the current values in the corresponding neighboring subdomain, and, thus, applications are generally required to determine at which points in the agorithm fresh copies of data must be obtained from the neigbboring subdomain.

## 2.1 Aside on the Shared-Memory Paradigm

Although the technique of domain decomposition (or more generally *divide-and-conquer*) can be applied in many parallel paradigms, it is generally not necessary within the *shared-memory* paradigm[3]. Perhaps the most well-

[3]The shared-memory paradigm should not be confused with shared-memory *architectures*. The latter refers to hardware capability to globally address all memory within a parallel architecture. It is true that such hardware support is beneficial to implementing the shared-memory paradigm, but it is not in principle necessary. OpenMP can, at least in theory, be implemented on distributed systems, and message-passsing programs certainly

known and widely used implementation of this paradigm is OpenMP [2] which extends Fortran and C/C++ with a small set of compiler directives and an even smaller library. Under the shared-memory paradigm, teams of execution threads share a global-address space and thereby eliminate the need to explicitly distribute subdomains in a more-or-less predetermined manner. A given computational loop can be parallelized by assigning various iterations to processors. Some care must be taken to avoid simultaneous access to the same array element, but in general this approach is relatively easy to implement. Generally scaling of 8-16 processors is readily attained, but extending the scaling efficiency to larger numbers of processors on existing architectures generally involves carefully aligning work distribution with memory distribution - a process very similar to domain decomposition. Another major liability of the shared-memory approach is that it generally requires that the compiler be "parallel aware", thereby reducing portability somewhat due to the availability of appropriate compilers.

## 2.2 Aside on MPI - the Message Passing Interface

By far the most portable and widely used interface for supporting parallelism (including but not exclusive to domain decomposition) on distributed-memory architecures is the standardized Message Passing Interface (MPI) [8]. The capabilities of MPI include

- Point-to-point communication (sending data from one process to another)
- Creation/management of teams of processes
- Reduction operations (e.g. summing data across a team of processors)

MPI supports the implementation of domain decomposition by providing an application the means to explictly copy data from neighboring subdomains into the halo region of the local subdomain. Such operations generally involve both the process providing the data and the process receiving the data, though MPI-2 introduced the capability for so-called "1-sided" communications in which only one processor is explicitly involved in the operation. (The other processor must still be involved, but in a rather indirect sense beyond the scope of this discussion.)

---

have been implemented on shared-memory architectures.

Portability of MPI is partly achieved by virtue of the fact that compilers need not be aware of the parallel environment - all such issues are deferred to the MPI library. Further, since MPI is actually only an interface, multiple implementations can and do exist. Many high-end hardware vendors provide highly optimized MPI implementations that exploit custom aspects of their interconnect, and at least two widely-ported public domain implementetations of MPI have been created: MPICH and LAM-MPI.

It should be noted that although MPI is standardized and nearly universally available, other message-passing interfaces that are still in use today include the Parallel Virtual Machine (PVM), which preceded MPI, and Cray SHMEM which offers a simple (albeit somewhat dangerous) interface that can exploit certain architectures such as those provided by Cray and SGI.

## 2.3 Aside on ESMF Infrastructure

The Earth Science Modeling Framework (ESMF) has as its goals and requirements to simplify and standardize the treatment of all domain decomposition, communication and synchronization issues by its earth science modeling components.

Since ESMF is tailored to a particular kind of applications, namely earth modeling, a lot of the parallelization and grid management details can be handled "behind-the-scene" by the framework. Parallelization is then achieved by calling the higher level ESMF routines, which themselves are MPI based.

The user calls ESMF routines to create the appropriate grid (e.g. C-grid), assign a domain decomposition, specify where in the grid cell each particular distributed array elements are located (e.g. NE corner), as well as creating objects that contain all details necessary to manage parallel issues for particular field arrays. Once that infrastructure is in place, ESMF routines (or their temporary locally provided versions) are available to provide "halo-updates", communication of cell data from the neighboring sub-grids, as well as global sums and other parallel tools.

While the plan is well thought out, at the time of this report not all the ESMF planned capabilities are yet in place. Those that were missing and were required for completion of the Distributed Parallelization of the modelE code have been temporarily implemented by our staff and are to be considered as such: temporary patches to be used until the more robust ESMF versions become available.

## 2.4 Specific Design Choices for modelE

The domain used in MODELE is a three-dimensional grid based upon the traditional "lat-lon" coordinates on a sphere logically equivalent to a three-dimensional Cartesian index space. A variety of decompositions are conceivable, but subdomains formed by splitting along each dimension independently are sufficiently flexible for most purposes. While a 1D decomposition is generally easier to implement, it strongly limits the total number of subdomains and hence the number of processors that can be exploited. Such explicit constraints on scalability are known as processor *starvation* - any additional computing elements will be idle due to lack of assigned tasks. A two dimensional decomposition is generally sufficient to avoid starvation for modest resolutions on existing architectures, and for this reason three-dimensional decomposition is rarely used. [4]

During the planning for the creation of an MPI implementation of MODELE, it was decided to restrict the initial decomposition to one dimension across latitudes. By avoiding a 2D decomposition, final delivery was probably shortened by 1-2 months, but at the expense of a larger total investment if a later decision is made to pursue the 2D decomposition. Figure 2 demonstrates how subdomains are associated with processing elements, and figure 3 illustrates how halo regions are associated with subdomains.

Naively, one might expect that the best axis along which to place a 1D decomposition would be the largest dimension, which in the case of MODELE would be across longitudes. There are 2 important, but subtle, reasons why such a choice is inferior to distributing across latitudes. First, due to serial (pipelining/vectorization) considerations, the loops across longitudes are generally the innermost loops and would become quite short. Such short innermost loops combined with skips over halo points would be very detrimental to serial performance. The second motivation to avoid decomposition across longitudes is that it effectively reduces the packet size for halo fill operations.

Thus, MODELE has been implemented with a 1D decomposition across latitudes. Each subdomain includes all longitudes and all levels (heights), but only a subset of the latitudes. Because MODELE uses a variety of rank

[4] Anoter mechanism to avoid starvation implicit in the 1D decomposition is to combine MPI with OpenMP, allowing OpenMP to manage multiple processors within each MPI subdomain. This can be particularly effective on some architectures, notably the SGI Origin series.

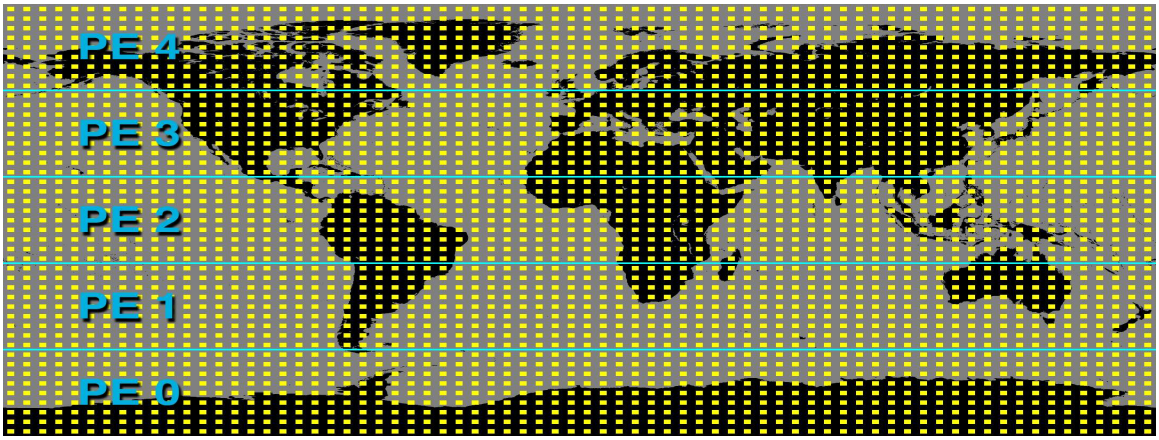


Figure 2: One-dimensional domain decomposition across latitudes for MODELEusing 5 processes.

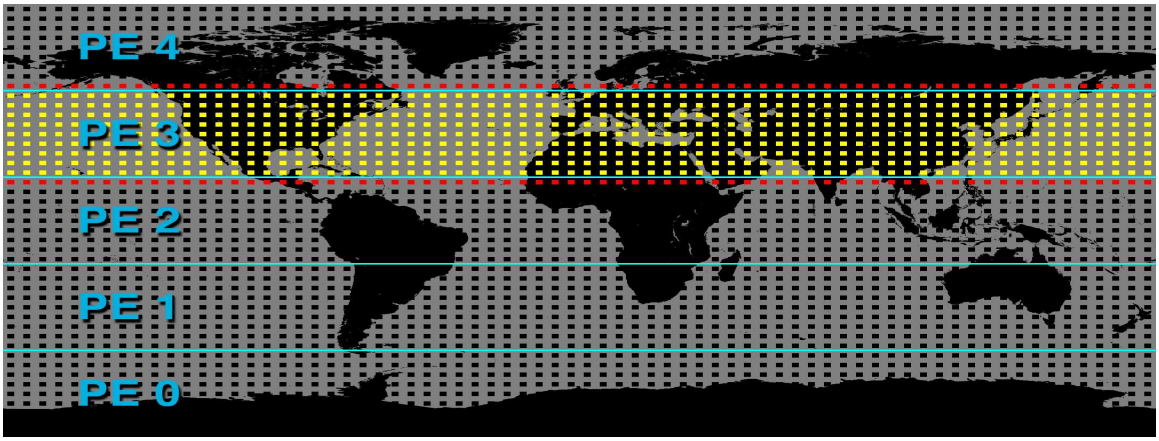


Figure 3: Each subdomain is represented by an array which includes both the interior (yellow points) and additional "halo" regions along the boundary (red points). Such halo regions are necessary for many types of numerical computations such as computing North-South derivatives.

permutations for different multidimensional arrays, one cannot simply describe the subdomain in terms of array slices. However, the most common case is for global arrays dimensioned as `ARR(IM,JM,LM)` with the decomposition affecting only the second rank of the array. Specific details and examples are provided in later sections.

Another major design choice for the MPI implementation of MODELE is that of preserving global indices. Many MPI implementations number the local indices beginning at 1 (or occasionally 0). When a global coordinate is required in such an implementation a process-dependent offset must be combined with the local index. For a homogeneous domain, there is relatively little disadvantage for such a choice, and the convention somewhat simplifies some of the bookeeping for subdomains. However, spherical domains are not homogeneous in the meridonal coordinate (i.e across latitudes). In particular, special logic for the North and South poles is ubiquitous whithin MODELE. Other references to the equator and special regions of interest are less common, but nonetheless present. Our convention maintains direct correspondence between local indices and the global indices so that a given value of say `J` is well defined and easily understood independent of the processor on which the reference is made. E.g. `J=1` always refers to the South pole, and `J=JM` always refers to the North pole. (One must, of course, still excercise caution since not all processors will have those coordinates within the bounds of the arrays for which they are responsible.) The major price for this convention is the fact that the offset in the lower bound is different from Fortran's default of "1" and therefore requires awkward specification statements for procedure dummy variables and local arrays. Again, details and examples are provided in later sections.

# 3 Data Structures

## 3.1 The Defalult "Grid" and Creating New Grids

The computational grid is represented as an object of a derived data type called `DIST_GRID`. One of the components of the "grid" object is an object called `ESMF_GRID`. This has the type `ESMF_GRID` and it represents the computational domain and the domain decomposition from the perspective of ESMF. Therefore, all subroutine calls to the ESMF library routines use this `ESMF_GRID` object as an argument. The remaining components of the "grid"

object represent grid parameters such as the number of longitudes, number of latitudes and various indices of the local domain that is mapped to each processor. The DIST_GRID derived data type is shown in Figure 4.

```
TYPE DIST_GRID

   TYPE (ESMF_Grid) :: ESMF_GRID
   ! Parameters for Global domain
   INTEGER :: IM_WORLD        ! Number of Longitudes
   INTEGER :: JM_WORLD        ! Number of latitudes
   ! Parameters for local domain
   INTEGER :: J_STRT          ! Begin local domain latitude  index
   INTEGER :: J_STOP          ! End   local domain latitude  index
   INTEGER :: J_STRT_SKP      ! Begin local domain exclusive of S pole
   INTEGER :: J_STOP_SKP      ! End   local domain exclusive of N pole
   INTEGER :: ni_loc ! for transpose
   ! Parameters for halo of local domain
   INTEGER :: J_STRT_HALO     ! Begin halo latitude  index
   INTEGER :: J_STOP_HALO     ! End   halo latitude  index
   ! Parameters for staggered "B" grid
   ! Note that global staggered grid begins at "2".
   INTEGER :: J_STRT_STGR     ! Begin local staggered domain
   INTEGER :: J_STOP_STGR     ! End   local staggered domain
   ! Controls for special cases
   LOGICAL :: HAVE_SOUTH_POLE ! South pole is in local domain
   LOGICAL :: HAVE_NORTH_POLE ! North pole is in local domain
   LOGICAL :: HAVE_EQUATOR    ! Equator (JM+1)/2 is in local domain
END TYPE DIST_GRID

TYPE (DIST_GRID) :: GRID
```

Figure 4: The DIST_GRID derived type.

A grid is created by a call to the subroutine INIT_GRID. This subroutine calls ESMF library routines to allocate the grid and initializes components of the "grid" object. Figure 5 describes the INIT_GRID subroutine and the meaning of each argument:

```
INTERFACE:

   SUBTOUTINE INIT_GRID(grid, IM, JM, LM, width, VM)

ARGUMENTS:

   INTEGER, INTENT(IN) :: IM, JM, LM
   TYPE (DIST_GRID), INTENT(INOUT) :: grid
   TYPE (ESMF_VM), INTENT(IN), Target, Optional :: vm
   INTEGER, OPTIONAL :: width


MEANING of AGRUMENTS:

IM - Number of Longitudes
JM - Number of latitudes
LM - Number of vertical levels
VM - An ESMF related variable called the Virtual Machine. This variable
     is an abstraction of the parallel computer.
width - Width of the ghost regions
```

Figure 5: The INIT_GRID subroutine and the meaning of each argument.

## 3.2 Allocating arrays

In the previous version of MODELE statically declared arrays were used to store physical quantities such as U and T. Although such globally-dimensioned arrays could be used in a distributed-memory setting, they would be very wasteful of memory and could be misleading for array references outside of the local domain. In the new distributed-memory implementation of MODELE such array quantities are instead dynamically allocated with the Fortran ALLOCATE statement. As an example consider the declaration of U in the previous and current releases:

```
Previous:
       REAL*8, DIMENSION(IM,JM,LM) :: U
New:
       REAL*8, ALLOCATABLE, DIMENSION(:,:,:) :: U
```

The actual bounds for U are established during the initialization phase of modelE:

```
       ALLOCATE(U(IM,J_0H:J_1H,LM))
```

where J_0H (=J_STRT_HALO) and J_1H (=J_STOP_HALO) are quantities that specify the extent of the (haloed) local domain which are to be extracted from the relevant "grid" data structure. All local extents are determined by making a call to the subroutine GET(). This subroutine takes a "grid" object as an input argument and several optional output arguments corresponding to the requird local indices of the grid. For the sake of clarity, we adopted the convention of grouping all allocation statements from a Fortran module into a separate allocation subroutine. Thus most Fortran modules hava an associated allocation subroutine. Each of the allocation subroutines is called at initialization by a single top level routine, ALLOC_DRV, to create appropriately sized arrays. Figure 7 below shows an excerpt from subroutine ALLOC_MODEL_COM, which is responsible for allocating the module variables used by the MODEL_COM module. A call to GET() at the beginning of each of these routines is used to extract the latitude indices from the "grid" object and assign those values to local variables. Note that since virtually all arrays eventually require references to halo regions, the convention is to allocate arrays to include the halo region.

Figure 6 describes the interface of the subroutine GET() and its arguments.

```
INTERFACE:

      SUBROUTINE GET(grid, J_STRT, J_STOP, J_STRT_HALO, J_STOP_HALO,
     &                     J_STRT_SKP, J_STOP_SKP,
     &                     J_STRT_STGR, J_STOP_STGR,
     &                     HAVE_SOUTH_POLE, HAVE_NORTH_POLE)

ARGUMENTS:

      TYPE (DIST_GRID), INTENT(IN) :: grid
      INTEGER, OPTIONAL :: J_STRT, J_STOP
      INTEGER, OPTIONAL :: J_STRT_HALO, J_STOP_HALO
      INTEGER, OPTIONAL :: J_STRT_SKP, J_STOP_SKP
      INTEGER, OPTIONAL :: J_STRT_STGR, J_STOP_STGR
      LOGICAL, OPTIONAL :: HAVE_SOUTH_POLE, HAVE_NORTH_POLE

MEANING of AGRUMENTS:
```

Figure 6: Interface of the subroutine GET() and its arguments. The meaning of arguments is explained in Figure 4.

```
      SUBROUTINE ALLOC_MODEL_COM(grid)
!@sum  To allocate arrays whose sizes now need to be determined at
!@+    run time
!@auth NCCS (Goddard) Development Team
!@ver  1.0
      USE DOMAIN_DECOMP, ONLY : DIST_GRID
      USE RESOLUTION, ONLY : IM,JM,LM
      USE MODEL_COM, ONLY : NTYPE
      USE MODEL_COM, ONLY : ZATMO,HLAKE,FLAND,FOCEAN,FLICE,FLAKE0,
     *                      FEARTH,WFCS,P,U,V,T,Q,WM,FTYPE
      USE ESMF_CUSTOM_MOD, ONLY: modelE_grid
      USE ESMF_CUSTOM_MOD, ONLY: ESMF_CELL_SFACE
      USE ESMF_CUSTOM_MOD, ONLY: ESMF_CELL_CENTER

      IMPLICIT NONE
      TYPE (DIST_GRID), INTENT(IN) :: grid

      INTEGER :: rc
      INTEGER :: J_1H, J_0H
      INTEGER :: IER
      LOGICAL :: init = .false.

      If (init) Then
         Return ! Only invoke once
      End If
      init = .true.

      CALL GET( grid, J_STRT_HALO=J_0H, J_STOP_HALO=J_1H  )

      ALLOCATE(P(IM,J_0H:J_1H), STAT = IER)
      ALLOCATE(U(IM,J_0H:J_1H,LM), STAT = IER)
      ALLOCATE(V(IM,J_0H:J_1H,LM), STAT = IER)
      ALLOCATE(T(IM,J_0H:J_1H,LM), STAT = IER)
      ALLOCATE(Q(IM,J_0H:J_1H,LM), STAT = IER)
!     ...
      END SUBROUTINE ALLOC_MODEL_COM
```

Figure 7: Subroutine ALLOC_MODEL_COM.

### 3.3 Declaring dummy arrays and local arrays

On those occasions when an array is used as a dummy variable for a procedure, care must be taken to ensure that it is declared in a manner which reflects the underlying domain decomposition. Unfortunately, the default lower bound for Fortran is `1`, which interferes with the scheme to use global indices. Furthermore, Fortran prohibits invoking user-defined procedures in declaration statements, which in turn forces us to make explicit references to the components of the `GRID` object[5].

```
Before:
      REAL*8 :: dum_argument(:,:,:)
After
      REAL*8 :: dum_argument(:,GRID%J_STRT_HALO:,:)
```

This is visually unappealing, but functional.

Purely local arrays can be declared in precisely the same manner as that of the global arrays as discussed in the previous section. Alternatively, Fortran *automatic* arrays can be used as in:

```
      REAL*8 :: local_array(IM,GRID%J_STRT_HALO:GRID%J_STOP_HALO,LM)
```

If allocatable arrays are used, developers are encouraged to use an explicit `DEALLOCATE` statement to restore memory[6].

# 4 Distribution of Parallel Work

## 4.1 Determining Local Extents and Halos

Distributed memory parallelization through domain decomposition has been implemented only for the latitude direction, and therefore primarily impacts loops over the latitude index `J`. Each process must determine what range of `J` values are to be handled locally during a given computation. Complicating factors include special logic for treating coordinates near the poles and also

[5]In general, the ASTG attempts to avoid dangerous coding practices such as global quantities, but in this instance convenience and consistency with existing modelE practices was an overriding concern.

[6]For unsaved allocatable arrays, the Fortran95 standard implies that such quantities are automatically deallocated upon exit of the routine. It is nonetheless a good practice to make this operation explicit.

for working with quantities that are located on the staggered grid. By using local indices accessed through the `GET()` interface discussed in the previous section, developers can clearly indicate the appropriate bounds for the local subdomain and control whether to execute pole specific operations (via the `HAVE_NORTH_POLE` and `HAVE_SOUTH_POLE` logical flags.

## 4.2 Modifying Loops over Latitudes

**Standard loops** The standard computational grid spans from the South pole (`J=1`) to the North pole (`J=JM`), implying the same span for the loops that apply logic to the entire computational domain. In the parallel context, the local lower and upper bounds of such loops can be obtained via the `J_STRT` and `J_STOP` bounds respectively. For the process which includes the South pole in its subdomain, `J_STRT=1` for the process which possesses the North pole `J_STOP=JM`.

**Treatment of North and South poles** In some cases, the logic in MODELE mandates that a loop applies everywhere except at the poles, usually followed by special operations that apply only *at* the poles. In the serial case, such loops range from `J=2` (one latitude north of the south pole) to `JM-1` (one latitude south of the north pole). For this case, the local `J` range for computation as defined in the GRID variable is `J_STRT_SKP` for the begining `J` and `J_STOP_SKP` for the end `J`. If the south pole is present in the local sub-domain,, `J_STRT_SKP` is equal to 2, if the north pole is present, `J_STOP_SKP` is equal to `JM-1`.

To determine whether to apply a pole-specific operation, developers can use the `HAVE_NORTH_POLE` and `HAVE_SOUTH_POLE` flags as returned by the `GET()` interface.

**Staggered grids** Some operations apply to quantities on the so-called "staggered" grid, and thus have `J` ranging from 2 to `JM`. For these cases, the local `J` range for computation as defined in the GRID variable is `J_STRT_STGR` for the begining `J` and `J_STOP_STGR` for the end J. If the south pole is present in the local sub-domain, `J_STRT_STGR` is equal to 2, if the north pole is present, `J_STOP_STGR` is equal to `JM`.

In all of the above three cases, one could call the subroutine `GET()` to extract the relevant begining and end J's. Throughout this MPI implementation of modelE, the calling sequence for subroutine GET follows the format of the following example:

```
CALL GET(grid, J_STRT = J_0, J_STOP = J_1,
&               J_STRT_SKP  = J_0S,   J_STOP_SKP  = J_1S,
&               J_STRT_STGR = J_0STG, J_STOP_STGR = J_1STG)
```

and the local `J` range indices take abreviated names, such as

```
J_0   , J_1       for "Standard loops"
J_0S  , J_1S      for "Skipping poles"
J_0STG, J_1STG    for "Staggered grid"
```

The rationale for using the aliases for the begining and end J's was to reduce the number of characters used in specifying loop bounds and to make it easy to globally replace loop bounds if there is ever a need to do so.

It should be noted that for process *not* doing the south pole,

```
J_0 = J_0S = J_0STG
```

Likewise, for processes *not* doing the north pole,

```
J_1 = J_1S = J_1STG
```

### 4.2.1 Unusual Bounds

There are places where the `J` loop ranges do not fall into any of the categories described so far. Such places are handled locally on a case-by-case basis using the following general approach:
If the starting `J` value in the original sequential code is **some_start_jvalue** that is NOT equal to one of `J_0, J_0S` and `J_0STG`, then the equivalent starting `J` value in the parallel code is **MAX(J_parallel,some_start_jvalue)** where `J_parallel` is one of `J_0, J_0S`, `J_0STG` and even `J_STRT_HALO` depending on how the `J` range in the sequential code is defined. Likewise, if the end `J` value in the original sequential code is **some_end_jvalue** that is NOT equal to one of `J_1, J_1S` and `J_1STG`, then the equivalent end `J` value in the parallel code is **MIN(J_parallel,some_end_jvalue)** where `J_parallel` is one of `J_0, J_0S`, `J_0STG` and `J_STOP_HALO` depending on how the `J` range in the sequential code is defined. Some examples are:

```
ATMDYN.f (around line 662)
   Original: do 2035 j=3,jm-1
   Parallel: do 2035 j=max(J_0S,3), J_1S

CLOUDS2_DRV.f (around line 1193)
   Original: DO J=J5S,J5N
   Parallel: DO J=MAX(J_0,J5S),MIN(J_1,J5N)

LAKES.f (around line 561)
   Original: DO J=1,JM
   Parallel: DO J=MAX(1,J_0-1),MIN(JM,J_1+1)
```

A more exhaustive list of these special cases are shown in Appendix A.

**Remark 1** *LAKES.f special note: Unusual loop bounds are employed to account for filling and using "halo" cells for KDIREC, IFLOW and JFLOW. This became necessary to meet the need of "halo" for these variables resulting from domain decomposition.*

## 4.3 Special Issues

### 4.3.1 Stopping the Parallel Code

The sequential code was equipped with subroutine `stop_model` to cleanly exit a run upon meeting some condition. This subroutine has been augmented with an option to cleanly exit a parallel run as well upon meeting the same condition. The original sequential code snippet is

```
      if ( retcode > 13 .and. dump_core > 0 ) then
        call sys_abort
      else
        call exit_rc ( retcode )
      endif
```

The augmented code snippet is

```
      if ( retcode > 13 .and. dump_core > 0 ) then
#ifdef USE_ESMF
```

```
        call mpi_abort(MPI_COMM_WORLD, retcode,iu_err)
#else
        call sys_abort
#endif
      else
#ifdef USE_ESMF
        call mpi_finalize(mpi_err)
#endif
        call exit_rc ( retcode )
      endif
```

### 4.3.2 Random Number Generator

ModelE uses a random number generator function (RANDU). RANDU is called from within a J loop a different number of times, depending on the value of J. In a parallel implementation this causes the local count of calls to RANDU, and thus the random number seed sequence, to fall out of sinch with the count on the serial implementation. In turn, the seed mismatch impacts the bit-wise reproducibility of the parallel and sequential codes. In order to ensure consistent and bitwise reproducible results by the parallel implementation, the number of times random numbers are burnt (by calling the function RANDU), is kept uniform in all processes, and identical to that in the sequential code. This is achieved by forcing each local loop to burn as many random numbers as a global loop would, but only storing them for the appropriate I and J values.

The following example (from RAD_DRV.f around line 1120) illustrates this process::

**Sequential code**

```
DO J=1,JM                    ! complete overlap
  DO I=1,IMAXJ(J)
    RDMC(I,J) = RANDU(X)
  END DO
END DO
```

**Parallel code**

```
N_BURN = SUM(IMAXJ(1:J_0-1))
CALL BURN_RANDOM(N_BURN)
```

```
DO J=J_0,J_1                        ! complete overlap
  DO I=1,IMAXJ(J)
    RDMC(I,J) = RANDU(X)
  END DO
END DO

N_BURN = SUM(IMAXJ(J_1+1:JM))
CALL BURN_RANDOM(N_BURN)
```

By default, the `BURN_RANDOM()` routine simply invokes `RANDU()` multiple times to guarantee an identical sequence of random numbers on all proccesses. In some instances, when the algorithm for `RANDU()` is known and of a certain form (e.g. linear-congruential random number generators), `BURN_RANDOM()` can be implemented to efficiently skip an entire sub-sequence of random numbers. At this time, an efficient implementation for `BURN_RANDOM()` exists on halem by virtue of reverse engineering the system provided `RAN()` random number generator.

### 4.3.3 Serialized Operations

In certain instances, it becomes necessary to serialize operations on distributed arrrays. The places where such operations are performed are usually surrounded by calls to a variety of `PACK` (to gather data) and `UNPACK` (to scatter data) routines. Moreover, in some cases, the distributed arrray is given the name `VAR_loc` and the corresponding global array is given the name `VAR` where `VAR` in both cases is the array of interest e.g. `AIJ`. In other cases, the distributed array is given the name VAR and the corresponding global array is given the name `VAR_glob`. The usage in both situations depend on context. For diagnsotic variables defined primarily in `DIAG_COM.f` and `TRDIAG_COM.f` , the former definition i.e. `VAR_loc/VAR` is used. However, other modules/subroutines that use `DIAG_COM` and `TRDIAG_COM` modules rename `VAR_loc` to `VAR` for their own local uses so that `VAR` will locally remain distributed. Examples abound in the code. A few are given here:

```
ATMDYN.f
   USE DIAG_COM, only : ajl=>ajl_loc,jl_dudtsdrg

ATURB.f
```

```
USE TRDIAG_COM, only: tajln=>tajln_loc,jlnt_turb
```

# 5 Communication

## 5.1 The Halo Methodology

The term "Halo" refers to grid data that could be required by one processor but resides on a different processor. In finite difference models, such as modelE, the most common need is in providing nearest neighbor data for the update of array cells at the sub-domain boundary. It is common practice to allocate memory for a layer of halo cells surrounding the sub-domain boundary. The halo cells purpose is to cache the necesary data from neighboring sub-domains for the update of sub-domain boundary cells. The approach streamlines the communication process and makes for "cleaner" code where sub-domain boundary cells require no "special treatment". In the ESMF version of modelE, a single cell wide halo surrounds each sub-domain. In order to insure that only up to date halo data is accessed, the halo region is updated prior to any loop or statement that uses it. The halo region update is acomplished by calling the subroutine `HALO_UPDATE`. There are three overloaded versions of this subroutine. Each accomodates different array index order. Overloading covers different data type, as well as array dimension. Figure 4 shows the interface to this subroutines and their arguments from the perspective of a user.

```
INTERFACE:

      SUBROUTINE HALO_UPDATE(grid, array, direction)

      SUBROUTINE HALO_UPDATEj(grid, arrayJ, direction)

      SUBROUTINE HALO_UPDATE_COLUMN(grid, arrayColumn, direction)

ARGUMENTS:

    TYPE (DIST_GRID), INTENT(IN) :: grid
    INTEGER, OPTIONAL, INTENT(IN)  :: direction
```

```
    The "variables" array, arrayJ, and arrayColumn are multi-dimesional with type
    integer or real depending on the context it is used.

MEANING of ARGUMENTS:

    grid        -‘‘grid’’ object
    array       -The local array that is being updated -- (J), (I,J), or (I,J,K)
    arrayJ      -The local array that is being updated -- (J,M)
    arrayColumn -The local array that is being updated -- (M,J),(M,I,J),(M,I,J,K)
    direction   -By default all halo regions for a given subdomain  are updated.
                 options include "NORTH" and "SOUTH" to selectively fill specific
                 sections.
```

Thus, whenever a loop block references a variable with a “j+1” index or “j-1” index, a call to Halo_Update must be made before the loop block to fetch the “Halo” data for the variable from its neighbor processor in the “north” or “south” direction. Following is a simple example from the source file TQUS_DRV.f that demonstrates the use of a HALO_UPDATE.

```
      CALL HALO_UPDATE(grid, MV, FROM=NORTH)
        DO L=1,LM
          DO J=J_0H,J_1S
            DO I=1,IM
              MV(I,J,L) = MV(I,J+1,L)*DT
            END DO
          END DO
          IF (HAVE_NORTH_POLE) MV(:,JM,L) = 0.
        END DO
```

In this example, the loop block references the variable MV(I,J+1,L). For J=J_1S, J+1 will correspond to a halo cell in at least one sub-domain. Therefore, each processor makes a call to HALO_UPDATE to fetch the Halo data from its neighbor to the North. If the loop block had a reference to MV(I, J-1, L) instead, the optional argument “FROM” in the call to HALO_UPDATE will be equal to SOUTH indicating that the data is fetched from the neighbor processor to the South. Similarly, if the loop block references both

cases, the optional "FROM" argument is set to "NORTH+SOUTH" and the HALO_UPDATE subroutine is designed to make two calls to fetch the Halo data from both the North and the South neighbors.

## 5.2 GlobalSum

The Globalsum subroutine call is used to compute the sum of the elements of an array. Different variants of the subroutine Globalsum are provided using overloading in order to support different data types, arrays of different dimensions and to compute the sum of a two or three dimensional array in a given dimension. Therefore, the role played by the Globalsum subroutine in the parallel implementation is similar to that of the Fortran 90 intrinsic function "sum" in the serial implementation. Following is a description of the interface of the Globalsum subroutine and its arguments from the perspective of a user.

```
INTERFACE:

    SUBROUTINE GLOBALSUM(grid, array, gsum, hsum, zsum, istag, iskip, jband, all)

ARGUMENTS:
    TYPE (DIST_GRID),  INTENT(IN) :: grid
    REAL*8, INTENT(IN) :: array(...)
    REAL*8, INTENT(OUT):: gsum(...), hsum(...), zsum(...)
    INTEGER,OPTIONAL,  INTENT(IN) :: istag, iskip, all
    INTEGER,OPTIONAL,   INTENT(IN) :: jband(2)

MEANING of ARGUMENTS:

    grid - ''grid'' object
    array - The local array
    gsum - Global sum
    hsum, zsum - Auxiliary sums (explained later)
    istag - Flag to indicate staggered grids
    iskip - Flag to indicate skipping of poles
    all - Flag to indicate that all processors need the global
          sum instead of only the root processor
    jband - Array to indicate the range of latitudes to be added.
```

In the parallel case, the global sum is computed by gathering the distributed array to the root processor and by having the root processor perform the sum of the elements of the global array as required by the implementation. For example, the root processor may compute the sum of all the elements of the global array in case of a full reduction operation resulting in a scalar output, or it may compute the sum along a given dimension (such as the latitude) resulting in an output array of rank 1 less than the input array. Performing the global sum in this manner is not very efficient but it is necessary in order to guarantee bit-wise agreement with the serial code results. It also allows for the relatively simple handling of various special cases, such as skipping poles, staggered grids, "pole first", hemisphere sums, and zsum.

In case of skipping poles (denoted by the optional "iskip" argument to the subroutine), data at the poles are excluded when computing the global sum because the index for the latitude varies from 2 to JM-1. Staggered grids (denoted by the optional istag argument to the subroutine) are also easily handled because we simply need to consider the latitude ranges from 2 to JM when summing up the elements of the global array. The arguments, zsum and hsum denote some auxiliary sums that are computed by the globalsum subroutine. The "zsum" represents the local sum of the input array along the longitude. Hsum denotes the "Hemisphere sum". When computing Hsum, the sum of the input array is first computed along the longitude. Depending on the input, this results in an intermediate array of size $JM$ or $JM \times LM$ where $JM$ denotes the number of latitudes and $LM$ denotes the number of vertical levels. The Hemisphere sum is the sum of the first and second halves of this array along the latitude and is an array of size 2 or $2 \times LM$. Another important case is unusual bounds for the latitudes where the globalsum is computed on a small range of the latitude indices. This case is handled via the optional argument "jband" - a rank 2 array that denotes the lower and the upperbound of the latitude. Once the array is gathered to the root processor, this case is easily handled by just computing the sum of the elements in the required latitude range. Finally, in some instances, the global sum may be required by all the processors instead of just the root processor. This case is denoted by the optional argument "all" to the subroutine. It requires an extra communication step because the root processor has to send the final output to all the processor by means of a broadcast operation.

In the process of parallelization, recognizing where a global sum operation is required is sometimes much less than obvious. Following is a section of code from the serial source file, SURFACE.f. The parallel version of this

code section will require a global sum operation it accumulates boundary layer diagnostics into variables ADIURN and HDIURN.

```
 IF(MODDD.EQ.0) THEN
   DO KR=1,NDIUPT
     IF(DCLEV(IJDD(1,KR),IJDD(2,KR)).GT.1.) THEN
       ADIURN(IH,IDD_DCF,KR)=ADIURN(IH,IDD_DCF,KR)+1.
       ADIURN(IH,IDD_LDC,KR)=ADIURN(IH,IDD_LDC,KR)
*           +DCLEV(IJDD(1,KR),IJDD(2,KR))
       HDIURN(IHM,IDD_DCF,KR)=HDIURN(IHM,IDD_DCF,KR)+1.
       HDIURN(IHM,IDD_LDC,KR)=HDIURN(IHM,IDD_LDC,KR)
*           +DCLEV(IJDD(1,KR),IJDD(2,KR))
     END IF
   END DO
 END IF
```

In the parallel implementation, each processor accumulates a partial sum of the DCLEV variable into a temporary array. Then a global sum is peformed on the temporay array. Finally, the root processor adds the global sum to the variables ADIURN and HDIURN. Following is how this piece of code is implemented in the parallel case.

```
 IF(MODDD.EQ.0) THEN
   DIURN_partb = 0

   DO KR=1,NDIUPT
     I = IJDD(1,KR)
     J = IJDD(2,KR)
     IF ((J >= J_0) .AND. (J <= J_1)) THEN
       IF(DCLEV(I,J).GT.1.) THEN
         tmp(1)=+1.
         tmp(2)=+DCLEV(I,J)
         DIURN_partb(:,J,KR)=DIURN_partb(:,J,KR)+tmp(1:2)
       END IF
     END IF
  END DO

  CALL GLOBALSUM(grid,  DIURN_partb, DIURNSUMb)
```

```
 IF (AM_I_ROOT()) THEN
    ADIURN(ih,idx3,:)=ADIURN(ih,idx3,:)    + DIURNSUMb
    HDIURN(ihm,idx3,:)=HDIURN(ihm,idx3,:) + DIURNSUMb
 END IF

END IF
```

## 5.3 Input and Output

### 5.3.1 Messages to STDOUT

The majority of the ASCII output goes to STDOUT (Unit 6, 0, or *). There is some ASCII output that goes to other file units, such as 67 and 99. ASCII output is currently handled either by the overloaded routine WRITE_PARALLEL in DOMAIN_DECOMP.f or by the regular Unix "write" and "print" statements. The routine WRITE_PARALLEL forces only the root process to write. The future plan is to replace all "write" and "print" statements with WRITE_PARALLEL, paying special attention to situations where the output data is not local to the root process. The interface to WRITE_PARALLEL DIMAIN_DECOMP is

```
interface WRITE_PARALLEL
   module procedure WRITE_PARALLEL_INTEGER_0
   module procedure WRITE_PARALLEL_INTEGER_1
   module procedure WRITE_PARALLEL_REAL8_0
   module procedure WRITE_PARALLEL_REAL8_1
   module procedure WRITE_PARALLEL_STRING_0
   module procedure WRITE_PARALLEL_STRING_1
end interface
```

where the name of each module procedure is suggestive of the type of data to be written. For example, WRITE_PARALLEL_INTEGER_0 is called to write an integer scalar, WRITE_PARALLEL_REAL8_1 is called to write a one-dimensional real array, etc. The module procedures have the same calling sequence exemplified by WRITE_PARALLEL_INTEGER_0 as shown below:

```
subroutine WRITE_PARALLEL_INTEGER_0 ( data, UNIT, format)
```

```
INTEGER, intent(in   )            :: data
integer,            intent(in   ),  optional :: UNIT
character(len=*),   intent(in   ),  optional :: format
```

Of course, the type declaration for data will be consistent with the type of the actual argument as guaranteed by the interface `WRITE_PARALLEL`. If `UNIT` is not present, the default is `STDOUT`. If `format` is not present and the output is intended to be `ASCII`, the default is to use list-directed output (i.e. use an asterisk (*) instead of an explicit format specification.

### 5.3.2 File based I/O of distributed quantities

Distributed arrays are usually written to or read from binary files. The file unit is given by variable *kunit*. For output, the appropriate overloaded routine `PACK_DATA`, `PACK_COLUMN`, `PACK_DATAj` or `PACK_BLOCK` from `DOMAIN_DECOMP.f` is called to gather the distributed array into the global array at the root process which then writes the data. For input, the root process reads the global array and then the appropriate overloaded routine `UNPACK_DATA`, `UNPACK_COLUMN`, `UNPACK_DATAj` or `UNPACK_BLOCK` from `DOMAIN_DECOMP.f` is called to scatter the data to all processes. For the sake of brevity, `PACK_DATA` will henceforth imply `PACK_DATA`, `PACK_COLUMN`, `PACK_DATAj` and `PACK_BLOCK`. Likewise `UNPACK_DATA` will henceforth imply `UNPACK_DATA`, `UNPACK_COLUMN`, `UNPACK_DATAj` and `UNPACK_BLOCK`. The interfaces for the `PACK_DATA` and the `UNPACK_DATA` routines are defined as follows:

```
!@var PACK Generic routine to pack  a global array
!@+   with the data from the corresponding distributed array.
      PUBLIC :: PACK_DATA
      interface PACK_DATA
         module procedure PACK_1D       ! (i)
         module procedure PACK_2D       ! (i,j)
         module procedure LPACK_2D      ! (i,j)
         module procedure PACK_3D       ! (i,j,l)
         module procedure IPACK_3D      ! (i,j,l)
         module procedure PACK_4D       ! (i,j,l,m)
      end interface
```

```
      PUBLIC :: PACK_DATAj
      interface PACK_DATAj
         module procedure PACKj_2D     ! (j,k)
         module procedure PACKj_3D     ! (j,k,l)
         module procedure PACKj_4D     ! (j,k,l,m)
      end interface

!@var PACK_COLUMN Generic routine to pack  a global array
!@+   with the data from the corresponding distributed array.
      PUBLIC :: PACK_COLUMN
      interface PACK_COLUMN
         module procedure PACK_COLUMN_1D  ! (k,  j  )
         module procedure PACK_COLUMN_2D  ! (k,i,j  )
         module procedure PACK_COLUMN_i2D ! (k,i,j  )
         module procedure PACK_COLUMN_3D  ! (k,i,j,l)
      end interface

!@var PACK_BLOCK  Generic routine to pack  a global array
!@+   with the data from the corresponding distributed array.
      PUBLIC :: PACK_BLOCK
      interface PACK_BLOCK
         module procedure IPACK_BLOCK_2D    ! (k,l,i,j  )
         module procedure  PACK_BLOCK_2D    ! (k,l,i,j  )
         module procedure  PACK_BLOCK_3D    ! (k,l,i,j,m)
      end interface

!@var UNPACK Generic routine to unpack into a distributed
!@+   array the data from the corresponding global array.
      PUBLIC :: UNPACK_DATA
      interface UNPACK_DATA
         module procedure UNPACK_1D      ! (i)
         module procedure UNPACK_2D      ! (i,j)
         module procedure LUNPACK_2D     ! (i,j)
         module procedure UNPACK_3D      ! (i,j,l)
         module procedure IUNPACK_3D     ! (i,j,l)
         module procedure UNPACK_4D      ! (i,j,l,m)
      end interface
```

```
      PUBLIC :: UNPACK_DATAj
      interface UNPACK_DATAj
         module procedure UNPACKj_2D     ! (j,k)
         module procedure UNPACKj_3D     ! (j,k,l)
         module procedure UNPACKj_4D     ! (j,k,l,m)
      end interface

!@var UNPACK_COLUMN Generic routine to unpack into a distributed
!@+   array the data from the corresponding global array.
      PUBLIC :: UNPACK_COLUMN
      interface UNPACK_COLUMN
         module procedure UNPACK_COLUMN_1D  ! (k,  j  )
         module procedure UNPACK_COLUMN_2D  ! (k,i,j  )
         module procedure IUNPACK_COLUMN_2D ! (k,i,j  )
         module procedure UNPACK_COLUMN_3D  ! (k,i,j,l)
      end interface

!@var UNPACK_BLOCK  Generic routine to unpack into a distributed
!@+   array the data from the corresponding global array.
      PUBLIC :: UNPACK_BLOCK
      interface UNPACK_BLOCK
         module procedure IUNPACK_BLOCK_2D    ! (k,l,i,j  )
         module procedure  UNPACK_BLOCK_2D    ! (k,l,i,j  )
         module procedure  UNPACK_BLOCK_3D    ! (k,l,i,j,m)
      end interface
```

The comment at the end of each module procedure name signifies the dimension and the storage sequence of arrays that the procedure is used for. In all of them, index `j` is the distributed one. All `PACK_DATA` module procedures have the same calling sequence, and all `UNPACK_DATA` module procedures have the same calling sequence. for the `PACK_DATA` procedures is exemplified by `PACK_1D` as shown below:

```
      SUBROUTINE PACK_1D(grd_dum,ARR,ARR_GLOB)
      IMPLICIT NONE
      TYPE (DIST_GRID),  INTENT(IN) :: grd_dum
```

```
 REAL*8, INTENT(IN) ::
&        ARR(grd_dum%j_strt_halo:)
 REAL*8, INTENT(OUT) :: ARR_GLOB(grd_dum%JM_WORLD)
```

The calling sequence for the UNPACK_DATA procedures is exemplified by PACKj_3D as shown below:

```
 SUBROUTINE UNPACKj_3D(grd_dum,ARR_GLOB,ARR,local)
 IMPLICIT NONE
 TYPE (DIST_GRID),  INTENT(IN) :: grd_dum

 REAL*8, INTENT(IN) :: ARR_GLOB(:,:,:)
 REAL*8, INTENT(OUT) ::
&        ARR(grd_dum%j_strt_halo:,:,:)
 LOGICAL, OPTIONAL :: local
```

Of course, the type declaration for ARR and ARR_GLOB will be consistent with the type of the actual argument as guaranteed by the interface PACK_DATA and UNPACK_DATA. Those whose names begin with I are for cases where ARR and ARR_GLOB are integer arrays while those whose names begin with L are for cases where ARR and ARR_GLOB are logicals. The rest without I or L are for real*8 arrays. For UNPACK_DATA, the extra optional argument local is used for the case where ARR_GLOB is already available in each process's memory so that the unpacking operation is reduced to each process just copying its own portion of ARR_GLOB into ARR.

## 5.4 Distributed transpose

A distributed transpose subroutine (TRANSP) was added to the MPI parallel code to facilitate parallelization of subroutine TRIDIAG. In the original sequential code, subroutine TRIDIAG was used to solve tridiagonal matrix equations along I (longtitude) for each J (latitude) or along J for each I. After parallelization, for the former case, all information required to solve for each J is still available in the local memory of the process to which that J belongs. In the latter case however, the information required for each I

become distributed. To handle the latter case efficiently where all processes will be engaged with each process solving the matrices for one or moreI's independently, we rewrote the existing TRIDAG.f to now contain a module called TRIDIAG_MOD instead of just subroutine TRIDIAG. The new module has an overloaded interface called TRIDIAG_NEW. This interface provides access to module procedure TRIDIAG (exactly the same as the original subroutine TRIDIAG) for handling the former case and a new module procedure TRIDIAG_2D_DIST for handling the latter case. The interface TRIDIAG_NEW looks like

```
      Interface Tridiag_new
      Module Procedure tridiag
      Module procedure tridiag_2d_dist
      End Interface
```

The module procedures TRIDIAG and TRIDIAG_2D_DIST have the following argument lists:

```
     SUBROUTINE TRIDIAG(A,B,C,R,U,N)
!@sum  TRIDIAG  solves a tridiagonal matrix equation (A,B,C)U=R
!@auth Numerical Recipes
!@ver  1.0
      IMPLICIT NONE
      INTEGER, INTENT(IN):: N         !@var N    dimension of arrays
      REAL*8, INTENT(IN) :: A(N)   !@var A    coefficients of u_i-1
      REAL*8, INTENT(IN) :: B(N)   !@var B    coefficients of u_i
      REAL*8, INTENT(IN) :: C(N)   !@var C    coefficients of u_i+1
      REAL*8, INTENT(IN) :: R(N)   !@var R    RHS vector
      REAL*8, INTENT(OUT):: U(N)   !@var U    solution vector

      SUBROUTINE TRIDIAG_2D_DIST(A_dist, B_dist, C_dist, R_dist,
     &                           U_dist,grid, j_lower, j_upper )
!@sum  TRIDIAG  solves an array of tridiagonal matrix equations (A,B,C)U=R
!@auth Numerical Recipes
!@ver  1.0
      USE DOMAIN_DECOMP, ONLY : DIST_GRID
      USE DOMAIN_DECOMP, ONLY : TRANSP
      IMPLICIT NONE
```

```
Type (DIST_GRID), Intent(IN) :: grid
REAL*8, INTENT(INOUT) :: A_dist(:,grid%j_strt_halo:)
REAL*8, INTENT(INOUT) :: B_dist(:,grid%j_strt_halo:)
REAL*8, INTENT(INOUT) :: C_dist(:,grid%j_strt_halo:)
REAL*8, INTENT(INOUT) :: R_dist(:,grid%j_strt_halo:)
REAL*8, INTENT(OUT)   :: U_dist(:,grid%j_strt_halo:)
INTEGER, INTENT(IN)   :: J_LOWER, J_UPPER
```

Briefly described, TRIDIAG_2D_DIST generates a distributed transpose of each of the 2D arrays (matrices) before solving. This insures that all the data necesary for solution for a give I is local. The tridiagonal solution algorithm at this point is identical to the TRIDIAG routine. The final step is to perform a reverse distributed transpose of the solution. The following example illustrates how `TRIDIAG_2D_DIST` works. We assume the following:

```
IM=5 and JM=3 for the global horizontal
Number of processes = 2 with ranks 0 and 1
Process 0 holds J=1:2; process 1 holds J=3:3
The desired solution vector is dimensioned U(5,3)
The coefficient matrices are dimensioned A(5,3), B(5,3) and C(5,3)
The right hand side is dimensioned R(5,3)
```

The distributions of the matrices between the two processes are:

```
Process 0: A(1:5,1:2), B(1:5,1:2), C(1:5,1:2), R(1:5,1:2), U(1:5,1:2)
Process 1: A(1:5,3:3), B(1:5,3:3), C(1:5,3:3), R(1:5,3:3), U(1:5,3:3)
```

The global representations of the matrices are:

$$A = \begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \\ a41 & a42 & a43 \\ a51 & a52 & a53 \end{pmatrix}, \quad B = \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \\ b41 & b42 & b43 \\ b51 & b52 & b53 \end{pmatrix}, \quad C = \begin{pmatrix} c11 & c12 & c13 \\ c21 & c22 & c23 \\ c31 & c32 & c33 \\ c41 & c42 & c43 \\ c51 & c52 & c53 \end{pmatrix},$$

$$R = \begin{pmatrix} r11 & r12 & r13 \\ r21 & r22 & r23 \\ r31 & r32 & r33 \\ r41 & r42 & r43 \\ r51 & r52 & r53 \end{pmatrix}, \quad U = \begin{pmatrix} u11 & u12 & u13 \\ u21 & u22 & u23 \\ u31 & u32 & u33 \\ u41 & u42 & u43 \\ u51 & u52 & u53 \end{pmatrix}.$$

Distribution along J to the two processes is indicated by dotted lines as shown below:

```
Proc    0       1               0       1
A = a11  a12 | a13       B = b11  b12 | b13
    a21  a22 | a23           b21  b22 | b23
    a31  a32 | a33           b31  b32 | b33
    a41  a42 | a43           b41  b42 | b43
    a51  a52 | a53           b51  b52 | b53

Proc    0       1               0       1               0       1
C = c11  c12 | c13       R = r11  r12 | r13     U = u11  u12 | u13
    c21  c22 | c23           r21  r22 | r23         u21  u22 | u23
    c31  c32 | c33           r31  r32 | r33         u31  u32 | u33
    c41  c42 | c43           r41  r42 | r43         u41  u42 | u43
    c51  c52 | c53           r51  r52 | r53         u51  u52 | u53
```

Now, for each I, the original sequential code solves the equation:

$$\begin{pmatrix} ai1 & bi1 & ci1 \\ ai2 & bi2 & ci2 \\ ai3 & bi3 & ci3 \end{pmatrix} \begin{pmatrix} ui1 \\ ui2 \\ ui3 \end{pmatrix} = \begin{pmatrix} ri1 \\ ri2 \\ ri3 \end{pmatrix}$$

where for our example, I goes from 1 to 5 for a total of five decoupled set of equations. Of course, for our example ci1 and ai3 will be equal to zero each since the resulting equation for each I is tridiagonal in nature. Each process can be made to handle a number of I's since each I is independent of another. For this to happen however, each process will need to get from other processes parts of A, B, C and R needed for its I's and transpose it before solving for the unknowns. For our example, assuming Process 0 is set to do I=1:3 and Process 1 to do I=4:5, then, Process 0 will need elements of A, B, C and R corresponding to I=1:3 and J=3 from Process 1 and likewise, Process 1 will need elements of A, B, C and R corresponding to I=4:5 and J=1:2 from Process 0. After all processes complete their work, each will have elements of the solution U for their I's. For our example, Process 0 will have u(I,J) with I=1:3 and J=1:3 and Process 1 will have u(I,J) with I=4:5 and J=1:3. Thus the solution U will need to be redistributed in a reverse order to what was done for A,B,C and R so that in the end, each process will have u(I,J), I=1:5 but J=1:2 for Process 0 and J=3 for Process 1. To achieve both the

forward and the reverse distributions, an overloaded subroutine `TRANSP` is made available in `DOMAIN_DECOMP.f`. This subroutine is called for each of `A`, `B`, `C`, `R` and `U` from subroutine `TRIDIAG_2D_DIST`. For `U`, the optional logical argument "reverse" is supplied to indicate the direction of distribution. The interface for `TRANSP` looks like

```
INTERFACE TRANSP
  MODULE PROCEDURE TRANSPOSE_ij
  MODULE PROCEDURE TRANSPOSE_ijk
END INTERFACE
```

where `TRANSPOSE_ij` and `TRANSPOSE_ijk` have argument lists

```
SUBROUTINE TRANSPOSE_ijk(grid, x_in, x_out, reverse)
TYPE (DIST_GRID), INTENT(IN) :: grid
REAL*8 :: x_in(:,grid%J_STRT_HALO:,:)
REAL*8 :: x_out(:,:,:)
Logical, Optional, INTENT(IN) :: reverse

SUBROUTINE TRANSPOSE_ij(grid, x_in, x_out, reverse)
TYPE (DIST_GRID), INTENT(IN) :: grid
REAL*8 :: x_in(:,grid%J_STRT_HALO:)
REAL*8 :: x_out(:,:)
Logical, Optional, INTENT(IN) :: reverse
```

# 6 Using Parallel Implementation

## 6.1 Current Limits of Implementation

As described in the previous sections, the distributed memory parallelization is achieved by dividing the number of latitudes among processes. Theoretically speaking, one should be able to achieve a granularity of one latitude per process. However, the current implementation indicates that doing less than three latitudes per process does not give "correct" results. We are currently working on rectifying this but in the meantime, the number of latitudes per process should be limited to not less than three.

## 6.2 System Issues with Running Parallel Code

### 6.2.1 Special Launch Environment

The parallel executable requires MPI parallel environment to run. Each parallel computing center has some prescribed way of making resources available to a parallel job depending on platforms at and the policies of the center. Very often, parallel jobs are allowed to run only in the batch queues using some queuing system. Users are usually allowed to submit a batch-interactive session if they wish to perform some inetractive activities like debugging with respect to the parallel executable. Section 6.3 gives some examples of how to run on NCCS machines halem (HP SC45) and daley (SGI Origin3000).

### 6.2.2 Special Launch Mechanism

The parallel executable requires a mechanism/command for launching MPI jobs. This is platform dependent. The user will need to find out from the parallel computing center what the mechanism/command is. Section 6.3 gives some examples of how to run on NCCS machines halem (HP SC45) and daley (SGI Origin3000).

## 6.3 Building and Running the Parallel Execuatable

The parallel version of the modelE program requires the ESMF (Earth Sciences Modeling Framework) software library. The ESMF library is installed on Halem and could be accessed using the module command. If the ESMF library is not installed on your system, it could be downloaded from

**http://sourceforge.net/projects/esmf/**.

Building and running the parallel executable is similar to the serial case execept that additional arguments are appended to the command line to compile the program with support for ESMF and to specify the number of processors at run time.

The following is an example of how to compile and run the modelE program in the parallel case. In this example, the test case is called myrun and the rundeck used is E1F20.

The steps are:

1. Build the rundeck

```
gmake rundeck RUN=myrun RUNSRC=E1F20
```

2. Compile the program

```
gmake gcm RUN=myrun RUNSRC=E1F20 ESMF=YES
```

3. Run a 1 hour setup simulation using 16 processors

```
gmake setup_nocomp  RUN=myrun RUNSRC=E1F20 ESMF=YES NPES=16
```

4. After the 1 hour setup simulation completes successfully, edit the "I" file in the output directory and specify the duration of the run.

5. Run the simulation

```
setenv MP_NUM_THREADS <number of processors>
./runE myrun
```

Steps 1, 2 and 3 are carried out from the "decks" directory and step 5 is carried out from the "exec" directory. Also, if your system uses a batch system to run parallel jobs, steps 3 and 5 need to be executed from within a batch script. Steps 2 and 3 could be combined into a single step using the "setup" option as follows:

```
gmake setup  RUN=myrun RUNSRC=E1F20 ESMF=YES NPES=16
```

### 6.3.1 Porting Issues

In this section, we address how to modify the Makefiles and run scripts when porting modelE to a different computer system. The make rules for compiling modelE are defined in the file Rules.make under the "model" subdirectory. If the operating system/compiler is not supported, the make rules for the new system could be added by using one of the supported systems as a template. Some of the currently supported systems are IRIX64, Linux, AIX, Mac OS X (Darwin), Intel8 and OSF1. For example, you could open the Rules.make file under an editor, search for the string Intel8 to locate the rules section for the Intel8 compiler and duplicate and make the necessary changes to add the rules for the new system. In addition, you will also need to edit the "if block" for ESMF to specify the library and include file paths for the ESMF

software on your system. You could locate this block by searching for the string ESMF after opening the file Rules.make using an editor. The best way to determine the options required for ESMF support is to first compile one of the validation tests that comes with the ESMF software library and take note of what options were used to build the validation program on your system. For example, following are the options that were used to build the modelE program on a SGI Altix system assuming that the ESMF library is installed in "/usr/local/esmf".

```
ifeq ($(ESMF),YES)
CPPFLAGS += -DUSE_ESMF
LIBS += -size_lp64 -mp -L/usr/local/esmf/lib/libO/Linux.intel.64.default -lesmf -W
FFLAGS += -I/usr/local/esmf/include
INCS += -I /usr/local/esmf/include
endif
```

In order to run the modelE program, you need to specify how a parallel job is launched on your computer system. This is done by modifying the Perl script, "setup_e.pl" located in the exec subdirectory. Once again, open this file using an editor and search for the string "mpi_run" to locate the section where this change needs to be made. Currently supported systems are IRIX64, OSF1, AIX and Linux.

# 7 Tips for Debugging and Tuning

The message passing paradigm introduces significant complexity into the already cumbersome process of repairing defects (bugs) and tuning performance. In the worst case, the developer is effectively repairing/tuning $n_p$ different applications, but fortunately the "lock-step" style of SPMD significantly reduces the complexity. Many special tools, both commercial and free, exist to assist developers with these activities when using MPI. In this section we provide references to some of the more valuable of these tools as well as offer guidance on effective techniques specific to modelE.

## 7.1 Debugging MPI

### 7.1.1 Checksum()

Because it is infeasible to run full validation tests after every modification, situations may arise where the parallel version fails to provide the correct answer on varying numbers of processors, and one must track down the defect. In the distributed memory paradigm (i.e. MPI), examining values contained in distributed arrays can be quite challenging and tedious, so to mitigate this complexity, utility routines `CHECKSUM()`, `CHECKSUMj()`, and `CHECKSUM_COLUMN()` have been created and placed within `DOMAIN_DECOMP.f` to aid in that process. These routines compute a consistent set of global sums for multidimensional arrays and send the result to a default unit or to a unit specified by an optional argument. Additional arguments allow for specifying the line and file from which the call was placed. This is intended to be used in conjunction with the `CPP` predefined macros `__LINE__` and `__FILE__`.

By default, all of these checksum routines are deactivated, but can be activated at compile time via the extra flag `-DDEBUG_DECOMP`. In this manner, such debugging instrumentation can remain in the code, but be deactivated for production runs. With the debugging option, the initialization phase opens a single file called `CHKSUM_DECOMP` and by all checksum results are printed to that file unless overridden by the `unit` optional argument to the checksum routines.

A useful technique for using the checksum routines is to create a separate routine (we usually used the name `CHECKALL()`) which computes checksums for a variety of array valued quantities in modelE. This routine should accept line and file information in its argument list and pass this down to the checksum calls contained within. One then liberally sprinkles calls to `CHECKALL()` throughout the code. After executing the model on differing numbers of processors, one may diff the `CHECKSUM_DECOMP` files among the runs to determine the first location at which a difference occurred. Additional calls to `CHECKALL()` can be added to refine the location further if necessary.

*API for CHECKSUM*() goes here.*

On some occasions the checksum routines discussed above do not provide sufficient information to determine the underlying problem. A finer grained tool for debugging the state is provide by the routine `LOG_PARALLEL()`. This procedure enables the developer to write out processor-local data to a distinct file on each processor named `LOG_<pe#>`. Again as with `CHECKSUM()`, default

compilation flags leave `LOG_PARALLEL` deactivated. Optional arguments exist for reporting scalar integers and reals and one-dimensional vectors of integers and reals. Although higher-dimensional optional arguments could easily be added, the developer is cautioned that `LOG_PARALLEL()` can easily produce prohibitively *large* amounts of formatted data.

*API for LOG_PARALLEL() goes here.*

### 7.1.2 Parallel Debuggers

It is sometimes useful to use parallel debuggers that have support MPI to track down bugs. Some HPC centers have such debuggers on their systems. At NCCS, we have the totalview debugger from Etnus on halem, the HP SC45 system. Instructions on how to use totalview on halem can be found at

**http://nccstag.gsfc.nasa.gov/halem/totalview.html**.

We also have the SGI Prodev Workshop Debugger known as cvd on our SGI Origin systems. Instructions on how to use cvd can be found at

**http://nccstag.gsfc.nasa.gov/sgio/cvd.html**.

## 7.2 Performance Measurement and Tuning

Performance optimization for parallel applications can be somewhat nonintuitive for developers familiar with optimization in a serial context. The primary reason for this is that the available resource of processing power is not regarded as being fixed. For example, as one attains reasonable efficiency on a certain number of processors for a given model, an even larger number of processors can often be exploited. A change from 90% to 95% efficiency may effectively double the performance of an application by virtue of this effect.

On the other side of this issue, parallel developers must bear in mind that although increased effeciencies permit larger numbers of processors to work effecitively on an application, there may be relatively little improvement in terms of capacity. A larger total throughput is typically attained by running on the smallest number of processors on which a given application will fit in memory (and hence usually has a very high parallel efficiency). Usually a compromise is reached between throughput and time-to-solution.

### 7.2.1 Amdahl's Law

To be more precise about the impact of a potential optimization, it is useful to understand Amdahl's Law [1] for parallel efficiency. Amdahl's law neglects various aspects of parallel overhead related to communication, but captures the essence that for any realistic application there is some essential work that is essentially serial in nature. For example, the overhead of setting up a loop is nearly independent of the number of iterations. At large iteration counts, this is negligible, but there *must* be a number of processors at which the loop count on any given processor is so low that this overhead is relevant.

Let us denote the fraction of an application which is effecively serial by $f$. We then further assume that the remaining workload can be scaled at 100% efficiency on an arbitrary number of processors $n_p$. If $T_s$ is the time to execute the application on one processor[7], then it is straightforward to derive the time $T(n_p)$ to execute the application on $n_p$ processors

$$T(n_p) = T_s \left( f + \frac{1-f}{n_p} \right),$$

with an asymptotic value at large numbers of processors $T \to T_s f$. The parallel efficiency $E(n_p) \equiv \frac{T_s}{n_p T(n_p)}$ is then given by

$$E(n_p) = \frac{1}{f n_p + (1-f)}.$$

Amdah's Law assumes that parallel overhead for operations such as halo exchanges and global summation is negligible, and is therefore an optimistic estimate of parallel efficiency. For most real applications, execution time actually begins to increase beyond a critical number of processors as parallel overhead becomes more significant than savings from applying more resources.

[7]It should be noted that the time to run on one processor is not necessarily the same as the time to run the serial variant of an algorithm. In most cases, these quantities should be quite close.

### 7.2.2 Load Imbalance and Barriers

### 7.2.3 TAU

# 8 Future Directions

In this section we list a variety of potential modifications to modelE that may be worthwhile in the future. Whether any given aspect is worth pursuing is dependent on the relative gain in capability and on the availability of resources to implement.

## 8.1 Using more Processors

Although the scalability of this implementation of MODELE represents a significant improvement over previous releases, further scalability would be desirable in many instances. [8] As discussed in an earlier section, the decision to implement a 1D decomposition for MODELE leads to a serious restriction on scalability At the time that this is written, the implementation requires at least 3 latitudes within each subdomain, restricting the number of useful processors to $n_p = n_j/3$.

### 8.1.1 Relaxing 1D Decomposition Restrictions.

There are two common approaches to expanding beyond the existing 1D decomposition. The most portable of these is to use an explicit 2D entirely within MPI. An attractive alternative that is somewhat portable, but more effective on some architectures is the so-called hybrid model where OpenMP is used to govern teams of threads within each MPI process.

### 8.1.2 2D Decomposition

To create a fully 2D decomposition within MODELE is largely an excercise of duplicating most of the transformation on the `J` coordinate into the `I` coordinate. The process would be somewhat more rapid due to the fact that many issues have already been isolated, but undoubtedly new exceptional

[8] This statement is somewhat ameliorated by the characteristics of GISS's workload which which is heavily biased towards throughput as opposed to time-to-solution. Increasing scalability will not significantly alter the total number of cases that can be simulated within a given set of computing resources.

cases would also be found. Any forecast for the improvement in scalability that could be achieved by such an effort is unfortunately rather speculative, but experience based upon other codes suggest that a 2X increase in scaling is not unreasonable for some architectures.

### 8.1.3 Hybrid MPI-OpenMP

The hybrid approach offers the possibility of exploiting larger numbers of processors for a minimal effort. Indeed, much of the existing OpenMP instrumentation within MODELE is already suitable for supporting this paradigm. A modest number of loops for which OpenMP is used to distribute over `J` indices should be modified to distribute over other coordinates. Depending on whether the MPI implementation on a given architecture is *thread-safe*, there may need to be additional work to isolate some explicit communication from OpenMP loops. As with the discussion of 2D domain decomposition, an accurate prediction of scalability in this case is difficult to make, but on some favorable architectures, such as SGI Origin, a significant gain is a reasonable expectation.

## 8.2 Alternative Algorithms

A handful of algorithms within MODELE are "unfriendly" to domain decomposition. For the current 1D decomposition, the most obvious example is tridiagonal solves across latitudes which ultimately require expensive global redistribution of data. In a 2D decomposition, polar filters via FFT's would have similar performance bottlenecks. Many models have found alternative numerical algorithms to overcome thes issues. E.g. using iterative/multigrid elliptic solvers in place of direct solvers, and using cubed-sphere grids to avoid the need for polar filters (and hence FFT's).

### 8.2.1 Asynchronous Communication

The current implemetation of MODELE uses so-call *synchronous* communication in which pairs of processors exchange data simultaneously. MPI provides *asynchronous* communication protocols that can significantly reduce communication latency in some conditions depending on the underlying architecture, and thereby increase parallel efficiency. To explore this with MODELE would involve splitting communication calls into two phases - an asynchronous send

and an asynchronous receive. Conservatively, a one man-month level of effort should be adequate to demonstrate this technique on MODELE and evaluate the impact.

### 8.3 Serial Optimization

Traditional serial optimization should not be discounted. Cache-based memory systems suggest a number of code transformations that may have long-term performance benefits to codes. Unfortunately, such optimizations remain an art, and are most cost effective when the expensive portions of a code are highly-concentrated as opposed to the more diffuse profile for MODELE. Less profound, but far easier is to explore compiler flags on current work platforms. Initial investigations on halem indicate that 10% performance improvement can be obtained by more aggressive optimization flags without altering the numerical answer.

## References


[1] G.M. Amdahl. Validity of the single processor approach to achieve large scale computing capabilities. In *AFIPS 1967 Spring Joint Comput. Conf.*, volume 30, pages 483–485, 1967.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Academic Press, San Diego, CA, 2001.

[3] S.-J. Lin. A finite-volume integration method for computing the pressure forces in general vertical coordinates. *Quart. J. Roy. Meteor. Soc.*, 123:1749–1762, 1997.

[4] S.-J. Lin. A Vertically Lagrangian finite-volume dynamical core for global models. *Mon. Wea. Rev.*, 132:2293–2307, 2004.

[5] S.-J. Lin and R.B. Rood. Multidimensional flux-form semi-lagrangian scheme. *Mon. Wea. Rev.*, 124:2046–2070, 1996.

[6] S.-J. Lin and R.B. Rood. An explicit flux-form semi-lagrangian shallow water model on the sphere. *Quart. J. Roy. Meteor. Soc.*, 123:2477–22498, 1997.

[7] G.A. Schmidt, R. Ruedy, J.E. Hansen, I. Aleinov, N. Bell, M. Bauer, S. Bauer, B. Cairns, V. Canuto, Y. Cheng, A. Del Genio, G. Faluvegi, A.D. Friend, T.M. Hall, Y. Hu, M. Kelley, N.Y. Kiang, D. Koch, A.A. Lacis, J. Lerner, K.K. Lo, R.L. Miller, L. Nazarenko, V. Oinas, Ja. Perlwitz, Ju. Perlwitz, D. Rind, A. Romanou, G.L. Russell, Mki. Sato, D.T. Shindell, P.H. Stone, S. Sun, N. Tausnev, D. Thresher, and M.-S. Yao. Present day atmospheric simulations using GISS ModelE: Comparison to in-situ, satellite and reanalysis data. *J. Climate*. in press.

[8] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1997.

# A List of Unusual Bounds Cases

```
ATMDYN.f (around line 662)
   Original: do 2035 j=3,jm-1
   Parallel: do 2035 j=max(J_0S,3), J_1S

ATMDYN.f (around line 1448)
   Original: DO J=3,JM-1
   Parallel: DO J=MAX(J_0S,3), J_1S

ATMDYN.f (around line 1518)
   Original: DO J=3,JM-1
   Parallel: DO J=MAX(J_0S,3), J_1S

CLOUDS2_DRV.f (around line 1193)
   Original: DO J=J5S,J5N
   Parallel: DO J=MAX(J_0,J5S),MIN(J_1,J5N)

CLOUDS_DRV.f (around line 1056)
   Original: DO J=J5S,J5N
   Parallel: DO J=MAX(J_0,J5S),MIN(J_1,J5N)

DIAG.f (around line 781)
   Original: DO J=J5S,J5N
   Parallel: DO J=MAX(J_0,J5S),MIN(J_1,J5N)

DIAG.f (around line 4077)
   Original: DO J=1+JM/2,JM
   Parallel: DO J=MAX(J_0,1+JM/2),MIN(J_1,JM)

DIAG.f (around line 4082)
   Original: DO J=1,JM/2
   Parallel: DO J=MAX(J_0,1),MIN(J_1,JM/2)

DIAG.f (around line 4088)
   Original: DO J=1,JM/2
   Parallel: DO J=MAX(J_0,1),MIN(J_1,JM/2)
```

```
ICEDYN_DRV.f (around line 885)
   Original: DO 120 J=2,JM-2
   Parallel: DO 120 J=J_0S,MIN(JM-2,J_1)

LAKES.f (around line 561)
   Original: DO J=1,JM
   Parallel: DO J=MAX(1,J_0-1),MIN(JM,J_1+1)

LAKES.f (around line 605)
   Original: DO J=2,JM-1
   Parallel: DO J=MAX(2,J_0H), MIN(JM-1,J_1H)

LAKES.f (around line 795)
   Original: DO JU=2,JM-1
   Parallel: j_start=Max(2,J_0H)
             j_stop =Min(JM-1,J_1H)
             DO JU=J_start,J_stop

LANDICE_DRV.f (around line 121)
   Original: DO J=JML,JMU
   Parallel: DO J=MAX(J_0,JML),MIN(J_1,JMU)

LANDICE_DRV.f (around line 134)
   Original: DO J=JML,JMU
   Parallel: DO J=MAX(J_0,JML),MIN(J_1,JMU)

RAD_DRV.f (around line 2129)
   Original: DO J=J5S,J5N
   Parallel: DO J=max(J_0,J5S), min(J_1,J5N)

STRAT_DIAG.f (around line 241)
   Original: DO J=3,JM-1
   Parallel: DO J=MAX(3,J_0S),J_1S

STRAT_DIAG.f (around line 427)
   Original: DO J=3,JM-1
   Parallel: DO J=MAX(3,J_0S),J_1S
```

```
TRACERS_DRV.f (around line 7064)
   Original: do j=31,35
   Parallel: do j=MAX(31,J_0),MIN(35,J_1)

TRACERS_DRV.f (around line 7081)
   Original: do j=33,39
   Parallel: do j=MAX(33,J_0),MIN(39,J_1)

TRACERS_DRV.f (around line 7098)
   Original: do j=29,34
   Parallel: do j=MAX(29,J_0),MIN(34,J_1)

TRACERS_DRV.f (around line 7115)
   Original: do j=28,32
   Parallel: do j=MAX(28,J_0),MIN(32,J_1)
```