ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-197

Runtime System Library for Parallel Finite Difference Models with Nesting

by

John Michalakes

Mathematics and Computer Science Division

Technical Memorandum No. 197

March 1997

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

A Runtime System Library for Parallel Finite Difference Models with Nesting

John Michalakes

Abstract

RSL is a parallel run-time system library for implementing regular-grid models with nesting on distributed memory parallel computers. RSL provides support for automatically decomposing multiple model domains and for redistributing work between processors at run time for dynamic load balancing. A unique feature of RSL is that processor subdomains need not be rectangular patches; rather, grid points are independently allocated to processors, allowing more precisely balanced allocation of work to processors. Communication mechanisms are tailored to the application: RSL provides an efficient high-level stencil exchange operation for updating subdomain ghost areas and interdomain communication to support two-way interaction between nest levels. RSL also provides run-time support for local iteration over subdomains, global-local index translation, and distributed I/O from ordinary Fortran record-blocked data sets. The interface to RSL supports Fortran77 and Fortran90. RSL has been used to parallelize the NCAR/Penn State Mesoscale Model (MM5).

1 Overview

RSL is a Fortran library and run-time system for efficient and straightforward implementation of nested regular-grid applications, such as regional weather models, on distributed memory parallel computers. These applications employ finite-difference approximation on rectangular coordinate grids. Domains may be a single rectangle or, more generally, unions of rectangles. RSL facilitates parallel implementation of these models by providing support for domain decomposition, stencil communication, interdomain communication, local iteration, index translation, distributed I/O, and dynamic load balancing.

Features of RSL include the following:

- Domain specification, decomposition, and remapping over processors
  - Rectangular or irregularly shaped (union of rectangles) domains.
  - Multiple, two-way interacting nested domains, independently decomposed over processors.
  - Pointwise processor decomposition allowing nonrectangular processor subdomains for more precisely balanced allocation of work to processors.

  - Automatic run-time decomposition and re-decomposition of domains using built-in or user-supplied mapping functions.
  - Run-time migration of state data between processors for dynamic load balancing.
  - Automatic run-time calculation of array size requirements on each processor, allowing models to dynamically allocate memory for local domain data structures.

- Communication
  - Stencil exchanges for updating ghost regions. These are defined at a high level that encapsulates details of the underlying message passing. Packing and unpacking of messages from domain data structures is handled automatically and transparently within RSL,
  - Automatic first-use compilation of stencil communication schedules. Stencils automatically recompile when a domain is remapped,
  - Message aggregation semantics that allows many fields to be communicated within a single stencil exchange, yielding greater code compactness and, on machines where latency is a concern, efficiency.
  - Broadcast-merges to exchange forcing and feedback data between nested domains.

- Computation
  - Run-time support for local iteration over decomposed dimensions on each processor.
  - Run-time support for global-local index translation on each processor.
  - M4 and CPP macro support for expressing parallel loops and indices. These may be used directly or targeted by a source translator such as the Fortran Loop and Index Converter (FLIC)[7], enabling a same-source approach to parallelizing existing codes with RSL.

- Distributed I/O
  - Single-reader/single-writer distributed I/O, in which the "monitor" processor, usually node zero, reads and writes the file system and communicates with the other processors to distribute or collect the data.
  - Ability to read and write ordinary Fortran record-blocked serial data sets, allowing the parallel code to use native data sets.

MPMM and MM90, parallel versions of the Penn State/NCAR Mesoscale Model MM5 were parallelized using RSL [2, 3, 4, 8]. At the present time, RSL is applicable only to models using explicit solvers. Support for global communication in the form of parallel transpose routines is under future work. The transposes may be implemented in a code outside the RSL framework by calling MPI directly. The interested reader may also wish to consider routines in the PETSc tools package[1].

Continuing RSL development focuses on the FLIC source translation software, which is based on a full Fortran-parser front-end and application specific back-end software, to

generate distributed-memory code mapped transparently onto the RSL library from original model source code [7]. This approach, which is virtually *directiveless*, will allow full and seamless integration of MPP capability into the officially maintained weather model, eliminating the need to support separate versions of the code for different architectures.

The purpose of this article is to provide an introduction to RSL and its concepts, presenting usage examples where necessary, but deferring detailed information on the library to a reference manual [1] Section 2 discusses related research efforts by other groups. Section 3 gives a brief overview of the features of the type of application RSL is designed to help parallelize. Section 4 discusses the issues involved in parallelizing a model and describes the approach one use with RSL. Section 5 walks through the parallelization of a simple relaxation code using RSL. Section 6 discusses the implementation of a MM5 using RSL. Section 7 presents conclusions and directions for future work.

2 Related Work

RSL is similar to efforts of a number of other groups, particularly Comlib [5], LAPRX [6], and NNT/SMS [9], in that it provides high-level, efficient mechanisms for performing data-parallel computations over multiple interrelated grids.

Like the LAPRX software abstractions, RSL is able to support dynamic load balancing, efficient intergrid communication, irregularly shaped logical domains, and irregularly shaped processor subdomains. However, since RSL is less general—tailored to multiple-grid finite-difference atmospheric models—it is smaller, simpler, and more conceptually familiar to a geophysical modeler. At the same time, it exploits fine-grained parallelism over grids, whereas LAPRX is restricted to coarse-grained parallelism over grids (and not within them).

RSL, like NNT/SMS and Comlib, is closely targeted to atmospheric models. In particular, like NNT/SMS, RSL offers stencil exchanges and interprocessor communication to support nesting. NNT/SMS puts more emphasis on parallel I/O, though it limits grids and processor subdomains to rectangular geometries. RSL, on the other hand, offers advanced features: pointwise decomposed, irregularly shaped processor subdomains, dynamic remapping of work to processors for load balancing, and support for irregularly shaped nests. Earlier concerns that RSL required more dramatic modifications to existing codes for column callability have been addressed in the current version, without sacrificing RSL's unique ability to efficiently support irregularly shaped processor decomposition.

3 Model Characteristics

Finite-difference models of dynamical systems are widespread in atmospheric and other sciences. The models typically consist of a two- or three-dimensional gridded domain representing the model state—velocity, temperature, and pressure, for example. Most generally, a domain is initialized and then integrated forward over a series of time steps. Boundary input and model output are performed periodically, as follows:

[1] A draft is in progress. Please see http://www.mcs.anl.gov/Projects/RSL for the most up to date version.

Domain definition and initialization.
Loop over time.
    If it is time, acquire new boundary data.
    Advance domain state by one time step.
    If it is time, perform model output.
End loop.

At the beginning of the simulation, the model domain is defined in terms of its size, shape, and allocation in memory, and the initial state of the model is input or otherwise obtained. A second source of model input, lateral boundary conditions, may input periodically over the course of the simulation. During each time step, the state of the model for the next time step is computed for each grid point by evaluating the state at the point and some stencil of nearest-neighbor grid points.

$$
\begin{array}{lcccccc}
X_{i,j}^{new} & = & & & c_1 X_{i+i,j} & + & \\
 & & c_2 X_{i,j-1} & + & c_3 X_{i,j} & + & c_4 X_{i,j+1} \\
 & & & + & c_5 X_{i-i,j} & &
\end{array}
$$

The exact shape and number of points in a stencil depend on the order of the finite-difference method and on the gridding scheme used. Interpolation will also involve a stencil.

3.1 Nesting

Accurate resolution of weather phenomena improves with scale-appropriate resolution. However, as fineness of resolution increases, so does computational cost because of the added number of grid points and the smaller time step. Nesting is used to increase resolution over portions of a domain. Nesting is accomplished by positioning a higher-resolution domain within a coarser domain and exchanging forcing and feedback data between the two:

Parent domain definition and initialization.
Nested domain definition and initialization.
Loop over time.
    Advance parent domain one time step.
    Transfer parent domain state data to force the nest.
    Loop over nest time steps.
        Advance nested domain one time step.
    End loop.
    Transfer nested domain state data back to parent domain.
    If it is time, perform model output for both parent and nest.
End loop.

The parent domain advances one time step; then data in the region of the nest is transferred from the parent to the nest. The model iterates over the smaller nested domain time steps, bringing it forward to the same time level as the parent. Finally, nested domain data is transferred back onto the region of the parent domain, and the next time step commences.

Nested domains may themselves have nests, allowing simulations to reach arbitrarily fine resolutions within the limits of the particular dynamics and physics in the model.

4 Parallelization

Parallelizing a model on a distributed-memory parallel computer involves defining, decomposing, and allocating memory for the model domains; iteration over decomposed dimensions; local-global index translation; interprocessor communication; load balancing, nesting; and I/O. RSL provides support for each of these tasks.

4.1 Domain Definition, Decomposition, and Allocation

Domains are defined by describing their size, shape, and parentage to RSL. Size and shape are specified by giving the number of rows and columns for rectangular domains. For irregularly shaped domains, size and shape are specified by giving the outline of the domain, that is, by listing the coordinates of the vertices of the irregularly shaped domain's enclosing polygon. A domain may be any nonzero size provided it is totally enclosed by its parent domain (in the case of nest), within the limits of physical memory. A nest is always defined as the child of a parent domain, and parentage remains fixed for the duration of the nest. Multiple nested domains may be defined within a parent. There must always be a top-level mother domain that is defined first and only once. The mother domain is always rectangular and has no parent.

Decomposition of a domain maps each grid cell of the domain to a processor. All domains in a model are defined over the same set of processors. Viewed another way, each processor has a piece of every domain in the model. RSL automatically decomposes domains when they are defined or remapped. RSL's default algorithm divides the domains into partitions with the number of points as close to equal as possible. Each point of the domain can be allocated independently, allowing irregularly shaped processor subdomains. Domains may be redecomposed at any point during a run. The user may specify alternative decomposition algorithms.

Allocation pertains not to the domain itself but rather to the two- and three-dimensional arrays that store the state and intermediate variables used in the model. For a given decomposition, the arrays associated with a domain require a certain amount of memory on each processor. RSL does not actually allocate the arrays associated with a domain. Rather, it makes the size information available to the program. This size information may be used to allocate memory dynamically or simply to provide a means for checking that static sizes are large enough for a decomposition.

4.2 Local Iteration and Computation

Since a processor computes only the points that are stored locally, a mechanism is needed for keeping track of a processor's local allocation in the parallel code. RSL assumes the responsibility for keeping track of the points that are local on each processor and for directing iteration over those points. A number of mechanisms are provided. RSL may actually

control the iteration by applying model routines that the user provides as functional pointers, or it may simply make the partition information available to control iteration that is specified explicitly in the user program. Macros are provided to facilitate the expression of decomposed loops using RSL. The macros may be programmed manually or generated automatically by using a special purpose preprocessor or precompiler, such as FLIC.

4.3 Local and Logical Index Correspondence

Under the single-address space memory model, the indices of a point in the logical domain are identical to its array indices, so that the indices may be used interchangeably. Decomposition and shrinking of local data structures on processors break this relationship: the index of a point in a local processor's memory is almost never the logical index of the point in the global domain. Therefore, the relationship between the local array indices and logical coordinates must be explicitly established and maintained.

RSL automatically computes and makes available to the program both sets of indices. The indices in local data structures are used whenever a local array is referenced in the code. No assumptions can be made by the program about the actual value of these local indices except that a point i is always adjacent to the points $i-1$ and $i+1$ in a given dimension. A corresponding set of global indices are used for determining the position of a point within the logical domain, for example, when testing for proximity with a boundary.

4.4 Interprocessor Communication

Model computations that involve data from neighboring cells or from cells that exist on another domain will require communication if the cells reside on a different processor. To avoid complicated, error-prone, and potentially less efficient message-passing code in the model, RSL provides high-level communication mechanisms for handling the types of data dependency found in finite-difference models with nests. The *stencil* provides intradomain communication for finite-differencing and interpolation. The *broadcast-merge* provides communication for exchanging data between domains for nesting.

*Intra*domain communication resolves the nearest-neighbor data dependencies associated with finite differencing and horizontal interpolation. The set of neighboring points that have data needed for a computation is called a stencil. Under RSL, stencils are defined by specifying the points of the stencil and the fields (model variables) that should be exchanged on each of the points. Stencils are used in *stencil exchanges*: transfers of data from remotely stored points into extra cells of the local array that have been allocated around the partition. This padding is known as the "halo" or "ghost" region of an array. RSL automatically determines the size and shape of the ghost region for each defined stencil. During a stencil exchange, the needed data is automatically buffered on the sender and unbuffered on the receiver; hence, each stencil exchange involves only one message sent and one message received for each processor pair in the exchange, minimizing the latency cost of the transfer.

*Inter*domain communication transfers the forcing or feedback data between a parent domain and a nest. At the time a nest is created, RSL establishes a link between each parent domain point and the points in the nest it overlays (Figure 1). The links are logical and do not depend upon on what processor a parent or nested domain point resides. Downward

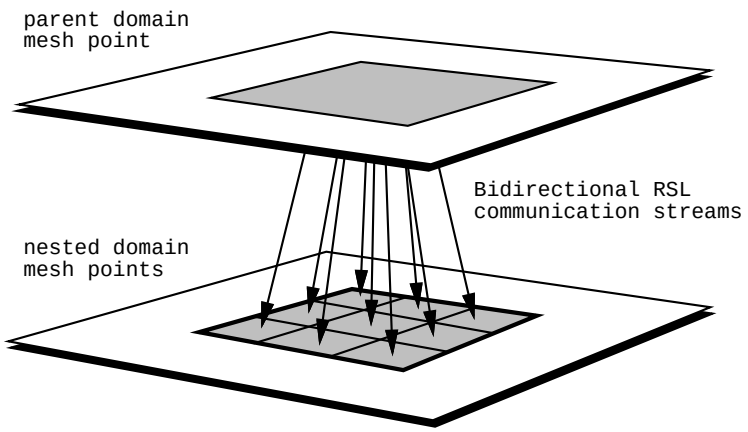

Figure 1: A parent domain cell and nine nested domain cells covering the same geography at different resolutions. The domains exchange data over communication streams.

forcing, from parent to nest, involves a logical broadcast from a parent domain point to the nest points that are linked to it. Upward forcing involves a merge along the same links but in the opposite direction.

Incidentally, RSL permits the ratio of nested to parent points to vary in each horizontal dimension (but always ≥ 1).

4.5 Load Balancing

Load imbalance occurs when some processors have more work to do than others. Processors that finish first idle, reducing performance relative to the ideal (in which all processors are kept busy). The ratio of actual performance to ideal performance is called the *efficiency*. Inefficiency from load imbalance may result from (1) an uneven initial distribution of domain points to processors–especially if the number of processors does not evenly divide the number of rows or columns; (2) reduced amounts of work in the boundary points of a domain; (3) dynamic conditions in the simulation itself that cause computations to be performed in some sections of the domain but not in others; or (4) different processor speeds or task loads in a heterogeneous or multiuser computing environment. RSL addresses this problem by supporting optimal decompositions of points to processors, whether or not the decomposition results in rectangular processor subdomains, and by providing a mechanism for distributing and redistributing domain cells between processors. Implementing irregularly shaped processor decompositions would be prohibitively complicated in an explicit message-passing code or using High Performance Fortran, which supports only regular decompositions of work to processors. However, RSL supports this automatically, transparently, and with little additional overhead.

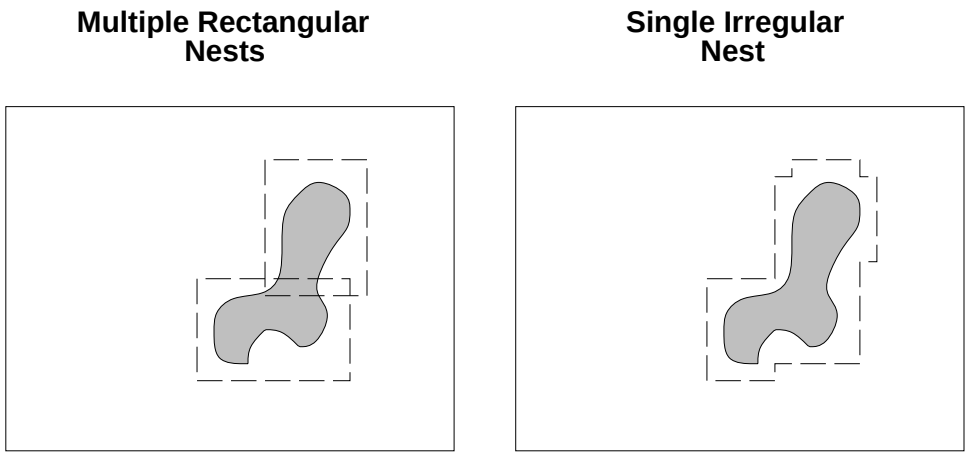

Figure 2: A single irregularly shaped nest fits a feature of interest in a simulation more closely and without additional code to handle the overlap region that occurs when two rectangular nests are used.

4.6 Irregularly Shaped Nests

Models that support nested domains may also allow multiple overlapping nests so that a user can overlay a number of rectangular nests to closely fit a feature of interest in the simulation, such as a weather front or a region of complicated terrain. RSL supports multiple domains on a nest level, but the user can avoid the need to writing complicated "overlap" code by specifying, instead, an irregularly shaped nested domain whose shape is a union of rectangles to fit the feature of interest (Figure 2). Control flow of the model is also simplified by eliminating nest overlapping in favor of irregularly shaped domains. The nesting hierarchy becomes strictly tree-shaped, since only parent-to-nest (not nest-to-nest) data dependency relationships need to be supported.

4.7 Input and Output

Reading data from a serial data set onto distributed domains and outputting distributed data to serial data sets requires communication between processors and may also introduce a serial bottleneck in the parallel code. RSL provides routines that read and write sequential Fortran data sets, automating the distribution of array elements to processors on input and the collection of array elements from processors on output. The parallel implementation of MM5, for example, is able to read and write serial MM5 data sets.

Although RSL manages the complicated task of decomposing serial input and recomposing serial output on the fly, the mechanism employed is currently "single reader, single writer"; that is, one processor reads and writes the data to files and sends and receives messages to the other processors. This aspect of the system is currently nonscalable and represents a disadvantage. However, since atmospheric codes such as MM5 generate output at a low frequency relative to the amount of computation that occurs between outputs, the single-reader, single-writer mechanism has not been a serious problem in the work with

MM5. Implementing a scalable yet portable solution to parallel I/O is an issue that will be addressed in future implementations of RSL.

5 Example: Simple Relaxation

The previous discussion described RSL's approach to parallelizing finite-difference weather codes with nesting. The purpose of this section is to introduce some of the main routines and discuss their usage in the context of a simple example. The size and complexity of a weather model make it unsuitable for illustration. Instead, a simple grid-boundary relaxation program is used. The basic algorithm is as follows.

Define an $m \times n$ *domain.*
Initialize the boundaries.
Do the following set of computations some number of times:
    Exchange stencil data.
    Do $j \leftarrow$ *1,* n
        Do $i \leftarrow$ *1,* m
            $New_{i,j} = (X_{i+1,j-1} + Xi + 1, j + \cdots)/8$
        End do.
    End do.
    Do $j \leftarrow$ *1,* n
        Do $i \leftarrow$ *1,* m
            $X_{i,j} = New_{i,j}$
        End do.
    End do.
End do.

Each iteration of the model computes the value at each point i, j as the average of its eight neighbors. Thus, when decomposed, some data necessary for the local computation is located on a different processor, and a stencil exchange must be defined and executed before each successive phase of the computation.

To simulate the interdomain communication in a model with a nested domain, the example has a nested grid. Also, the nest is defined with an irregular boundary to demonstrate this capability in RSL. The complete algorithm with the nested communication is as follows:

Define an $m \times n$ *mother domain.*
Define a finer resolution nest (arbitrary shape) in mother domain.
Initialize the boundaries of the mother domain.
Do the following until simulation is finished:
    Exchange stencil data on mother domain.
    Relax the mother domain.
    Transfer mother domain data onto nested boundary.
    Do the following several times (simulating a nested time step):
        Exchange stencil data on nested domain.

```
      program relaxmain
#include "rsl.inc"
      logical rsl_iammonitor
      external rsl_iammonitor
      ...
      call rsl_initialize
      if ( rsl_iammonitor() ) then
        read*, nproc_m, nproc_n
      end if
      call rsl_mon_bcast(nproc_m,4)
      call rsl_mon_bcast(nproc_n,4)
      call rsl_mesh(nproc_m, nproc_n)
```

Figure 3: RSL include file and initialization.

Relax the nested domain.
End do.
Transfer data from nest onto overlying cells of mother domain.
End do.

After each relaxation, mother domain cells transfer their values of X to the underlying nested domain points if those points lie on the boundary of the nest. This process is equivalent to the transfer of atmospheric variables to update the boundaries of a nested domain. The relaxation computation is performed several times on the nest; then, nested data is fed back onto the parent domain and the next major time step commences.

Main sections of the program are detailed below. The complete Fortran90 texts of the relaxation code are provided as appendixes.

5.1 Preliminaries

The main program is declared and the header file rsl.inc is included in Figure 3. Here, a CPP directive is used, but a Fortran INCLUDE also works. The RSL library must be initialized before it can be used. RSL_INITIALIZE partially initializes the run-time system so that certain basic functions may be used. Informational routines such as RSL_IAMMONITOR and the simple broadcast routine RSL_MON_BCAST are enabled, but not the rest of the system. This strategy allows configuration information to be read on processor zero (the "monitor") and distributed to the other processors. Initialization completes with a call to RSL_MESH, giving the values just read in for the numbers of processors decomposing the M and N dimensions. Configuration information supplied by Fortran Namelists is also input in this fashion at the beginning of a run.

5.2 Defining Domains

RSL must be given a description of the domains that it will be working with. Once described, a domain becomes active and an integer descriptor is returned for use when performing

```
C define the top level domain
      call rsl_mother_domain(
     $               did,              ! parent domain descriptor (output)
     $               RSL_8PT,          ! maximum stencil          (input)
     $               m, n,             ! global dimensions        (input)
     $               mloc, nloc )      ! local memory required    (output)

C define a nest
      xlist(1) =  4   ; ylist(1) =  6     ! outline of the nest, specified
      xlist(2) =  4   ; ylist(2) =  11    ! in coarse domain coordinates
      xlist(3) =  7   ; ylist(3) =  11
      xlist(4) =  10  ; ylist(4) =  14
      xlist(5) =  15  ; ylist(5) =  14
      xlist(6) =  15  ; ylist(6) =  9
      xlist(7) =  10  ; ylist(7) =  4
      xlist(8) =  6   ; ylist(8) =  4
      xlist(9) =  4   ; ylist(9) =  6     ! close the polygon (optional)
      npoints = 9
      call rsl_spawn_irreg_nest(
     $              nid,               ! nested domain descriptor (output)
     $              did,               ! parent domain descriptor (input)
     $              RSL_8PT,           ! max stencil              (input)
     $              xlist, ylist, 7,   ! xpts, ypts, and npts     (input)
     $              3, 3,              ! nesting ratio in m and n (input)
     $              2, 2,              ! domain trim factors      (input)
     $              mloc_n, nloc_n,    ! local memory size        (output)
     $              m_n, n_n )         ! global size              (output)
```

Figure 4: Definition of the top-level mother domain and an irregularly shaped nest.

computation, stencil exchanges, opening nests, remapping, and other operations on the domain. The top-level mother domain must be specified. Nests may be opened in any domain that is currently active at any time.

The mother domain. One domain must always be defined in a model. The code in Figure 4 declares this top-level mother domain for the relaxation program. M and n are integer variables giving the global number of rows and columns of the two-dimensional domain. Although nested domains may be irregularly shaped, the mother domain is always rectangular. The definition of the domain also results in its automatic decomposition. Here, the default decomposition algorithm in RSL is used. The decomposition generated for four processors is shown in Figure 5. The decomposition of the nested domain is shown in Figure 7.

On return, RSL_MOTHER_DOMAIN sets the integer arguments *mloc* and *nloc*, specifying the minimum local array size that will hold arrays associated with the decomposed domain. The values of *mloc* and *nloc* may differ on each processor. If the calling program uses dynamic memory allocation, such as is available in Fortran 90, the sizes can be used to allocate the model arrays. If not, the sizes may still be used to check that statically

```
domain=0, len_n=25, len_m=20
    2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
    2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
    0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 5: Decomposition of the mother domain.

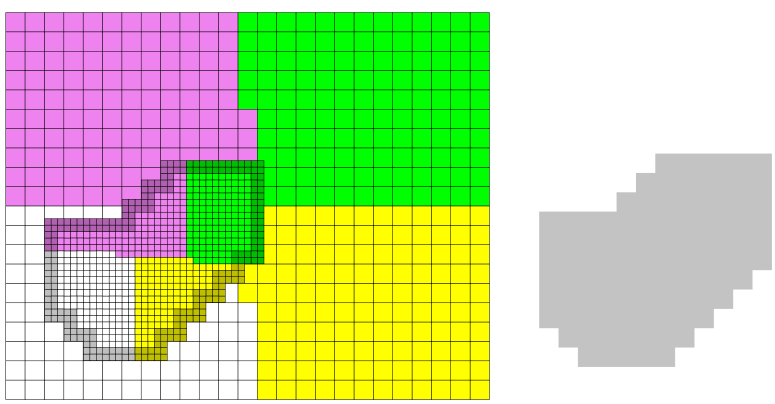

Figure 6: The parent domain and its nonrectangular nest as defined in the example relaxation code. The nest is trimmed two cells in each horizontal dimension (trimmed sections shown in grey).

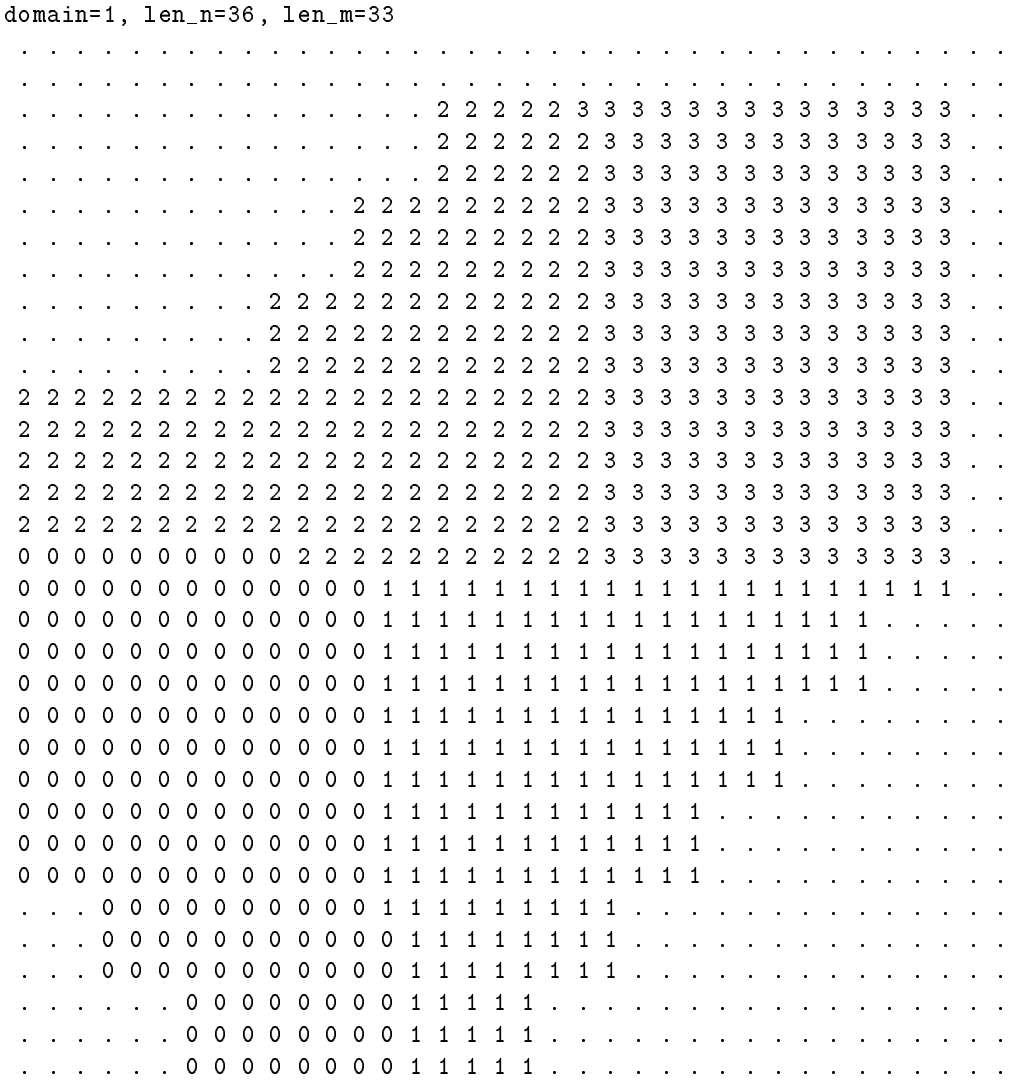

Figure 7: Decomposition of the irregularly shaped nest.

allocated arrays are large enough for the decomposition.

The value of RSL_8PT, defined in the rsl.inc, tells RSL what the largest stencil will be. In this case, an 8-point stencil is specified (a 9-point stencil if one counts the center point). On return, the integer variable *did* is set to the value of an RSL handle that refers to the domain just declared.

Defining a rectangular nest. Rectangularly shaped nests are easily specified in RSL by providing the number of rows and columns of the domain and by giving the position of the nest within its parent. RSL provides two routines for specifying regular nests. The first, RSL_SPAWN_REGULAR_NEST, specifies the dimensions of the nest in parent domain coordinates. The second, RSL_SPAWN_REGULAR_NEST1, specifies the dimensions in nest coordinates. The following example is from MM90, the Fortran90 parallel implementation of MM5:

```
      call rsl_spawn_regular_nest1(
     +             d%child(kid)%domdesc,
     +             d%domdesc,
     +             RSL_24PT,
     +             ipos,
     +             jpos,
     +             idim,
     +             jdim,
     +             irax, jrax,
     +             d%child(kid)%mloc, d%child(kid)%nloc,
     +             d%child(kid)%m,    d%child(kid)%n )
```

The first argument will be returned with the RSL descriptor for the nest (the reference here is through the Fortran90 derived data type for the parent). The second argument is the descriptor for the parent. RSL_24PT specifies the largest stencil that will be used on the domain. IPOS and JPOS are the parent domain coordinates of the southwest corner of the nest. IDIM and JDIM are the M and N dimensions of the nest.[2] IRAX and JRAX specify the nesting ratios in each dimension. The last four arguments return the minimum local array size in M, the minimum local array size in N, the effective global domain size in M, and the effective global domain size in N. The returned local memory requirements may be used to dynamically allocate local arrays or may provide a means to check that enough memory has been allocated for the given decomposition. The domain will have been automatically decomposed when the routine returns. Consult the RSL reference manual for additional information.

Defining an irregularly shaped nest. The relaxation problem in the example also specifies a nested domain. The call to RSL_SPAWN_IRREG_NEST in Figure 4 defines a nested domain and positions it within the coarse domain. The resolution of the nest is three times that of the coarse domain in each of the two horizontal dimensions, associating nine nested cells with each coarse domain cell in the region of the nest.

[2] M corresponds to the north-south dimension in MM5, N to the east-west dimension.

The integer argument *did* specifies the parent domain. On return, the integer argument *nid* is set to a handle that will be used when referring to the nest in subsequent calls to RSL.

The nest in the example is irregularly shaped. Its size and shape are specified by defining the outline of the enclosing polygon in parent domain coordinates, specified by *xlist* and *ylist*. The next argument is a count of the number of elements in each of the lists. Although the lists are hard-coded here, the outline coordinates could also come from a file or an interactive window that allowed the user to draw the nest's shape through a run-time user interface (not part of RSL).

Since the nesting ratio in each dimension is three, the dimensions of the nest in each dimension must be a multiple of three. The result, for example with staggered logical grids, may be iteration over parts of the nest that do not exist. Therefore, a number of cells may be trimmed from each dimension by specifying a *trim* in each dimension. Trims of 2 and 2 are specified here. The trim may be in the range of integers zero through the nesting ratio minus one. Cells are always trimmed from the high end of a dimension. The resulting parent and nested domains are shown in Figure 6.

As with the mother domain, the nest is automatically decomposed when it is instantiated. The routine returns the required memory size for the local partition (*mloc_n* and nloc_n) and the global size of the nest as if it were rectangular. In other words, *m_n* and *n_n* returned above specify the size of the rectangle that would totally enclose the nest, even though the nest's shape may not cover every point of that rectangle.

5.3 Defining Stencils

The relaxation computation uses the eight neighboring values of the variable X for each nonboundary cell of the domain. Since the horizontal dimensions of the domain are decomposed, some data needed for the computation will lie on other processors. Therefore, we define an 8-point stencil that can be used by the routine RSL_EXCH_STENCIL to exchange data prior to the computation. RSL stencils comprise messages, which are composed of one or more field descriptions. A stencil for the variable X is defined in Figure 8.

Building a stencil involves specifying the messages that will be required for each point of the stencil. Therefore, the messages are constructed first.

The call to RSL_CREATE_MESSAGE creates a temporary message descriptor that can be built up by adding field descriptions. The messages descriptor stays in effect until it is used in the description of the stencil, after which point the message descriptor becomes undefined. The call to RSL_BUILD_MESSAGE adds the two-dimensional array X as a field of the message. Other fields can be added to the message by repeated calls to RSL_BUILD_MESSAGE.

The first argument to RSL_BUILD_MESSAGE is the message descriptor. This is followed by the type specifier, in this case RSL_REAL (defined in rsl.inc). The field itself is specified as the third argument. The fourth argument is the dimensionality of X. The last three arguments, DECOM, GLEN, and LLEN, are integer arrays that specify how the dimensions are decomposed, the global or logical length of the dimensions, and the local declared lengths of the dimensions, respectively. The arrays are set at the top of the figure.

```
      decomp(1) = RSL_M                ! minor dim is m
      decomp(2) = RSL_N                ! major dim is n
      llen(1) = mloc                   ! local memory size in m
      llen(2) = nloc                   ! local memory size in n
      glen(1) = m                      ! global domain size in m
      glen(2) = n                      ! global domain size in n

c define the message to be sent on the stencil points
      call rsl_create_message( mesg )          ! create a message handle
      call rsl_build_message(
                     mesg,             ! add X to the message
                     RSL_REAL,         ! type of X
                     X,                ! X
                     2,                ! how many dimensions
                     decomp,           ! how decomposed
                     glen,             ! global dimensions
                     llen )            ! local dimensions

c create and define the stencil
      points(1) =    mesg
      points(2) =           mesg
      points(3) =                 mesg
      points(4) =    mesg
      points(5) =                 mesg
      points(6) =    mesg
      points(7) =           mesg
      points(8) =                 mesg
      call rsl_create_stencil( sten )
      call rsl_describe_stencil( did, sten, RSL_8PT, points )
```

Figure 8: Building a stencil description out of messages.

The lowest numbered index is most minor.

Once the messages for the stencil are built (in this case there is only one), the messages are assigned to the stencil points by first assigning them to an integer array POINTS whose length is equal to the number of stencil points, then using the array to describe the stencil with a call to RSL_DESCRIBE_STENCIL. The stencil descriptor being described must first have been created with a call to RSL_CREATE_STENCIL.

Stencil points are numbered from top to bottom and left to right. Thus, for an 8-point stencil, index 1 in the *points* array is the descriptor for the northwest message. Index 8 is the descriptor for the southeast message. The center point of the stencil is never specified. This example shows the same message being assigned to each point of the stencil. However, different messages may be assigned to different stencil points. If a stencil point is not used, the corresponding array element should be assigned the value RSL_INVALID, instead.

RSL_DESCRIBE_STENCIL describes a stencil and associates it with a domain. Stencils are specific to the domain for which they are described. Therefore, a stencil must be described for each new domain, for example, by including the stencil definitions in a sub-

```
C DECLARATION SECTION
      RSL_RUN_DECL
      ...
C EXECUTABLE
      RSL_INIT_RUNVARS(did)
      ...

      RSL_DO_N(j)
        RSL_DO_M(i)
          X(i,j) = 10.0
        RSL_ENDDO
      RSL_ENDDO

      RSL_DO_N(j,2,n-1)
        RSL_DO_M(i,2,m-1)
          X(i,j) = 0.0
        RSL_ENDDO
      RSL_ENDDO
```

Figure 9: Initializing the mother domain.

routine. RSL compiles communication schedules for stencils at the point of first use after a domain is defined or remapped.

5.4 Initializing the Mother Domain

The first computational part of the program initializes X. Boundary cells are given initial values of 10.0. Interior cells are set to 0.0. The code is shown in Figure 9.

The first loop sets all cells to 10.0; the second loop sets the interior to 0.0. The macro RSL_INIT_RUNVARS initializes the looping macros RSL_DO_M and RSL_DO_N for the domain specified by the integer descriptor DID. Arguments to the macros are the name of the loop variable and, if the loop is over a subrange of the logical dimension, the starting and ending global indices. Otherwise, only the loop variable is specified. Another way to handle boundary points separately from the interior is to code a single loop with a conditional in the body. This is done in the relaxation computation later in the example. RSL_RUN_DECL in the declaration section declares the data structures used when the loop macros are expanded.

The loop macros are defined in RSL in the file LoopMacros.m4, which may be expanded by using M4 on the command line or in a UNIX Make file:

```
m4 Loopmacros.m4 file.f
```

Often, subroutines are written to include iteration over one dimension but not the other; rather, the statement expressing the major iteration may be in a routine further up the call tree. This is handled in the called routine by including the comment

```
C define(INSIDE_MLOOP)
```

or

```
C define(INSIDE_NLOOP)
```

For additional information on RSL macros and their expansions, see the RSL reference manual. Macro insertion may be automated using FLIC.

5.5 Computation on a Domain

Computation of one relaxation step for either domain, mother or nest, involves iterating over the local subdomain on each processor and computing the average value of the eight neighbors, except on the boundaries, which are held fixed. The code is shown in Figure 10.

As in the previous example, RSL_INIT_RUNVARS initializes the loop macros. This step is not strictly necessary, since the first call to the macro for a domain initializes the looping structures for every routine with the RSL_RUN_DECL macro in its declaration section. The impact on efficiency is minimal, however, and this is the safest approach during code development.

Since the routine will be called on both rectangular and irregularly shaped domains, the boundary test must be handled accordingly. RSL provides support for handling irregularly shaped boundaries using conditionals within the parallel loops. RSL_GET_BDY_LARRAY initializes a three-dimensional integer array, BDYINFO, with boundary proximity information. The first two dimensions correspond to the horizontal dimensions of the domain and are indexed by the local indices I and J. The third dimension indexes the proximity information by kind. The values for this index are defined in the rsl.inc include file:

Normal grid boundary information (dot, in Arakawa B)	
RSL_MLOW	Distance to MLOW (south) boundary
RSL_MHIGH	Distance to MHIGH (north) boundary
RSL_NLOW	Distance to NLOW (west) boundary
RSL_NHIGH	Distance to NHIGH (east) boundary
RSL_DBDY	Distance to closest boundary
RSL_CLOSEST	Closest boundary
Cross grid boundary information (cross, in Arakawa B)	
RSL_MLOW_X	Distance to MLOW (south) cross boundary
RSL_MHIGH_X	Distance to MHIGH (north) cross boundary
RSL_NLOW_X	Distance to NLOW (west) cross boundary
RSL_NHIGH_X	Distance to NHIGH (east) cross boundary
RSL_DBDY_X	Distance to closest boundary
RSL_CLOSEST_X	Closest cross boundary

Before the relaxation, a stencil exchange is performed. RSL_EXCH_STENCIL is called with ID set to the identifier of the domain, DID or NID. STEN is set to the stencil descriptor defined for the domain. The exchange ensures that the data for every point needed in the average calculation will be on-processor, either locally or in the ghost region.

```
C DECLARATION SECTION
      RSL_RUN_DECL
      ...
C EXECUTABLE
      RSL_INIT_RUNVARS(did)
      call rsl_get_bdy_larray( id, bdyinfo )

      call rsl_exch_stencil( id, sten )

      RSL_DO_N(j)
        RSL_DO_M(i)
          if ( bdyinfo(i,j,RSL_DBDY) .eq. 1 ) then
             New(i,j) = X(i,j)
          else
             New(i,j) =  (
     $          X(i+1,j-1)   +  X(i+1,j)  +  X(i+1,j+1) +
     $          X(i,j-1)     +               X(i,j+1)   +
     $          X(i-1,j-1)   +  X(i-1,j)  +  X(i-1,j+1)
     $                ) / 8.0
          endif
        RSL_ENDDO
       RSL_ENDDO

      RSL_DO_N(j)
        RSL_DO_M(i)
          X(i,j) = New(i,j)
        RSL_ENDDO
      RSL_ENDDO
```

Figure 10: One relaxation step on a domain.

Following the boundary calculation in the first set of nested loops, the variable X is updated in a second set of loops. Note that unlike the the first loop body, the second loop body for the update has no horizontal dependencies. Therefore, no prior stencil exchange is required.

5.6 Forcing the Nest

Data is exchanged from a parent domain to a nest in three phases. The first phase involves packing the data on the parent domain, sending the data, and unpacking the data on the nest.

The packing in Figure 11 is implemented as a conditional loop that performs one iteration for each point in the nested domain, specified by *nid*. Each time RSL_TO_CHILD_INFO is called, it returns with the global coordinates of the nested point, NIG and NJG; the local and global indices of the forcing parent domain point, I,J, PIG, and PJG; and the index of the nested point, CM and CN, associated with the parent domain point. When no nested points remain to be processed, RSL_TO_CHILD_INFO returns a value of -1 in the integer argument RETVAL.

```
 call rsl_to_child_info(
$                         did,      ! parent    (input)
$                         nid,      ! nest      (input)
$                         4,        ! bytes from each point  (input)
$                         i, j,     ! local indices in parent (output)
$                         pig, pjg, ! global indices in parent (output)
$                         cm, cn,   ! which child of parent cell (output)
$                         nig, njg, ! global indices in nest (output)
$                         retval )  ! result (1 if a valid point)
 do while ( retval .eq. 1 )
   call rsl_get_bdy_gpt( nid, bdyinfo, nig, njg )
   if ( bdyinfo(RSL_DBDY) .eq. 1 ) then
     call rsl_to_child_msg(
$                           4,        ! size of data in bytes (input)
$                           X(i,j) )  ! data being added to message (input)
   endif
   call rsl_to_child_info( did, nid, 4,
$                           i, j, pig, pjg, cm, cn,
$                           nig, njg, retval )
 enddo

 call rsl_bcast_msgs
```

Figure 11: Forcing the nest boundary from the parent domain: packing the data from the parent and sending it between processors.

The call to RSL_GET_BDY_GPT ("get boundary information for a globally indexed point") is used to determine boundary proximity for the nested point indexed by NIG and NJG. If the nested point is on a boundary, RSL_TO_CHILD_MSG is called to pack the value of `X(i,j)`, the parent copy of the variable X, into the message destined for the point indexed by NIG and NJG on the nest. In this example, only one variable is being sent; however, RSL_TO_CHILD_MSG may be called as many times as necessary to pack fields for the nested point. After the completion of the packing loop, the messages are sent between processors with the call to RSL_BCAST_MSGS.

In Phase 2 of the broadcast, data from the parent domain cells is unpacked onto the nest. Again, a packing loop is used, now with calls to RSL_FROM_PARENT_INFO (Figure 12). On each iteration of the loop, RSL_FROM_PARENT_INFO returns the local and global indices of a nested point, I, J, NIG, and NJG, and the global indices of the sending parent cell, PIG and PJG. When no points remain, RSL_FROM_PARENT_INFO returns a value of -1 in RETVAL. RSL_FROM_PARENT_MSG unpacks the data into the nest's copy of the array X.

The upward forcing of nested data to the parent domain at the end of the set of nested iterations is similar to the downward forcing described here.

```
 call rsl_from_parent_info(
$                         i, j,     ! local indices in nest (output)
$                         nig, njg, ! global indices in nest (output)
$                         cm, cn,   ! which child of parent cell (output)
$                         pig, pjg, ! global indices of parent (output)
$                         retval )  ! result (1 if a valid point)
 do while ( retval .eq. 1 )
   call rsl_from_parent_msg(
$                           4,       ! size of data in bytes (input)
$                           X(i,j))  ! data being extracted (output)
   call rsl_from_parent_info(
$                           i, j, nig, njg, cm, cn,
$                           pig, pjg, retval )
 enddo
```

Figure 12: Forcing the nest boundary from the parent domain: unpacking onto the nest.

5.7 Dynamic Load Balancing

Domains are automatically decomposed when they are defined. However, the application may redefine the decomposition at any time. A remapping is done for a single domain and has no effect on other domains in the simulation.

The simplest remapping situation is if the data structures for a domain have not yet been allocated and initialized. In this case, the remapping is accomplished by using RSL_FDECOMPOSE:

```
rsl_fdecompose( d, fcn, nproc_m, nproc_n, info, mloc, nloc )
```

The first argument is the integer RSL domain handle. The FCN argument is a user-supplied function that RSL will use to determine a new processor assignment for each point in the domain. NPROC_M and NPROC_N are the numbers of processors decomposing m and n. The INFO argument may be used to pass information, such as per-processor timing information, to FCN for its internal use. MLOC and NLOC are integers that, on return, contain the minimum local array size for the newly decomposed domain. It is only slightly more difficult to remap a domain after arrays have been allocated and filled with distributed state data. This movement is accomplished by using a specially defined RSL state message and the routine RSL_REMAP_STATE (Figure 13).

RSL uses a state message to know how to pack and unpack fields when moving points between processors. The variable DSTATE in the figure is a Fortran90 structure containing pointers to X and other dynamically allocated model state arrays. The variable TMP is an identical structure that is used to hold the new state while the remapping from the orignal state is underway. Once the remapping is complete, the new state in TMP becomes the state stored in DSTATE by means of Fortran90 pointer assignments in a user-supplied routine, MOVE_STATE.

Fortran90 is used to simplify the management of the user data structures involved in the example remapping. Pointers, dynamically allocatable arrays, and derived types (struc-

```
c define array X as a member of the state message, used in remapping
      call rsl_create_message( mesg )
      call rsl_build_message( mesg,RSL_REAL,dstate%X,2,decomp,glen,llen )
        ... add other fields to message ...
      call rsl_describe_state( d, mesg )

c generate new decomposition; mloc and nloc return with new array sizes
      rsl_fdecompose( d, mapping, nproc_m, nproc_n, info, mloc, nloc )

c allocate new data structures for the domain; store in temporary structure
      call allocate_domain( tmp, m, n, mloc, nloc )  ! user defined

c now defined a new state message for the new X (associated with tmp)
      call rsl_create_message( mesg )
      call rsl_build_message( mesg,RSL_REAL,tmp%X,2,decomp,glen,llen )
        ... add other fields to message ...
      call rsl_describe_state( d, mesg )

c do the remapping
      call rsl_remap_state( d )

c replace the old model state with the new model state
      call move_state( tmp, dstate )    ! user defined
```

Figure 13: Dynamic redecomposition of a domain over processors during a model run.

tures) obviously simplify remapping. However, since RSL_REMAP_STATE is able to remap the domain arrays in-place, the remapping can be accomplished within statically allocated data structures under Fortran77. The mapping function, FCN, must be constrained to generate only those mappings that will fit in the statically allocated data structures on each processor.

5.8 Main Loop and Model Output

The main loop of the example is analogous to the time loop in a weather model. On each iteration, the domain and its nests are advanced one time step. The code is show in Figure 14.

Output is generated periodically by the calls to RSL_WRITE. The first argument is the Fortran unit number of the data set. This is followed by the data description flag IO2D, defined in the rsl header file rsl.inc. IO2D specifies that the array being written is two-dimensional. Implicit is the assumption that both dimensions are horizontal and decomposed and that the minor dimension of the array corresponds to the minor dimension of the domain. That is, m in the domain and m in the array X are both minor. The next argument is the X array itself, followed by the domain descriptor. RSL_REAL is a tag specifying the type of the array. The arrays GLEN and LLEN specify the global and local sizes of the arrays. The local array size is given by the elements of LLEN, the global domain size by GLEN.

```
do t = 0, iter-1
   <relax coarse domain>
   <forcing coarse->nest>
   do tnest = 1,3
     <relax nested domain>
   enddo
   <forcing nest->coarse>
   if ((mod(t,5).eq.0)) then
     call rsl_write(18,IO2D,Xparent,did,RSL_REAL,glen,llen)
     call rsl_write(19,IO2D,Xnest,nid,RSL_REAL,glen_n,llen_n)
   endif
endif
```

Figure 14: Main loop of relaxation. For each iteration, one set of calculations is performed over the parent domain, the nest is forced, then three sets of calculations are performed on the nest. Data from the nest is fed back onto the parent, and the cycle repeats. Output is generated every five iterations of the outer loop.

The relaxation code described in this section was run for fifty steps on multiple processors of an IBM SP2, generating output every five steps. The output data after the full fifty steps on the parent and nest domains is plotted in Figure 15.

6 Parallel MM5

MM5 is a numerical weather prediction model used for forecasting, climate prediction, and other atmospheric simulations for domains ranging from several thousand kilometers down to several hundred. Domains are uniform rectangular grids representing three-dimensional regions of the atmosphere. The horizontal coordinate system is equally spaced geographically, and the model uses the Arakawa-B gridding scheme. The vertical coordinate system is σ surfaces, with layers distributed more closely nearer the surface (23 layers in the current model). For this implementation, atmospheric dynamics is nonhydrostatic and uses finite-difference approximation. Physics includes the Blackadar high-resolution planetary boundary layer scheme, the Grell cumulus scheme, explicit moisture with treatment of mixed-phase processes (ice), shallow convection, dry convective adjustment, and the Dudhia long- and short-wave radiation scheme [3].

MM5 was first converted to column-callable form and then parallelized using the first generation of RSL. MPMM, as the parallel version is called (for Massively Parallel Mesoscale Model), was validated and benchmarked on the IBM SP1 and SP2. It has since been ported to the Intel Paragon, Cray T3D, Fujutsu AP1000, Silicon Graphics Power Challenge, and networks of workstations using MPI. Performance of 1.2 Gflops has been generated for a single domain problem running on 64 SP1 processors. Additional information regarding the parallel MM5 is available in [8] and on the World Wide Web at the location `http://www.mcs.anl.gov/Projects/mpmm/index.html`.

Parallelizing MM5 provided a number of insights and suggested improvements that have been incorporated into the latest version of RSL. These include explicit loop constructs that

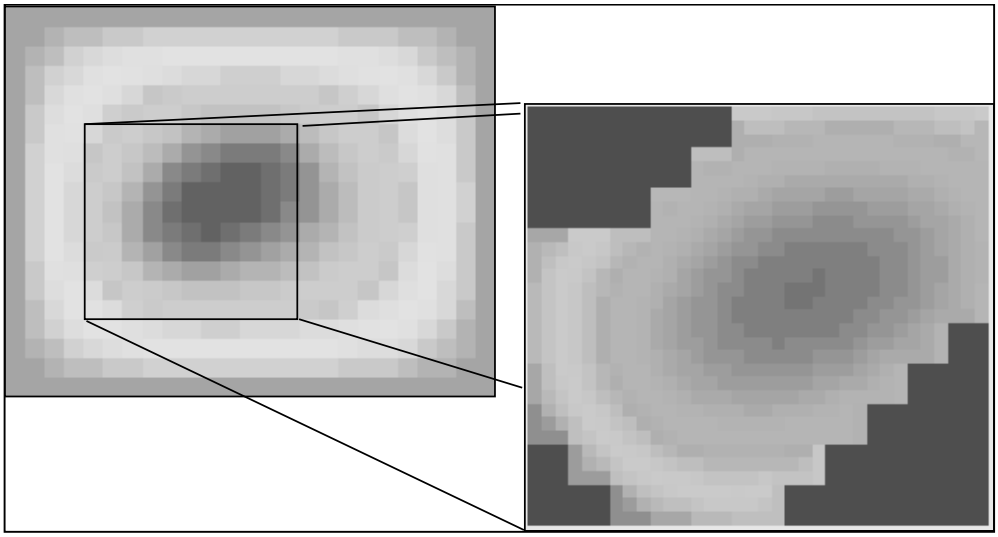

Figure 15: Example problem output from coarse domain (left) and nest running in parallel after 50 iterations of the main loop, plotted as 2D density plots. The value of each point is represented by shades of grey. The position of the nested domain in relation to the coarse domain is shown in outline. In each step on the coarse domain, the irregularly shaped boundaries of the nest are set to the values of the overlying coarse domain points. Then the nest performs several relaxation steps of its own. Finally, the values of the nested points are copied back onto the coarse domain points that overlay them.

allow relaxation of the column-callable requirement, support for dynamic memory allocation at the option of the programmer, removal from RSL of the need to explicitly compile stencils and broadcast-merges before use (they are now compiled automatically on first use), and the automatic and default decomposition of domains.

7 Conclusions and Future Work

RSL provides a system whereby an existing or new weather model can be implemented on a parallel machine with support for dynamic, fine-grained parallel decomposition of multiple nested domains and with high-level communication routines that hide and efficiently implement message passing. Improvements to the original version of RSL have also reduced the number of changes to an existing code, simplifying initial implementation and ongoing maintenance and effectively removing a source of architecture-dependent coding from the model.

New features and enhancements to the system will include

- ports to other parallel machines,
- scalable and portable parallel I/O,
- hybrid parallelism,
- fully Fortran90 implementation, and
- automatic dependency analysis and source translation.

Porting to other parallel machines such as the Cray T3E is already simplified because RSL can be implemented atop the MPI standard message-passing layer. The I/O mechanism of RSL is currently a potential obstacle to scalability and must be addressed in a portable manner.

Hybrid parallelism is the mixing of shared and distributed-memory approaches in a single architecture: a set of distributed memories representing the computational nodes of a message-passing computer but on each of which multiple processing elements operate in shared-memory mode. The semantics embodied in the computational and communication structures already in RSL will support this feature if the underlying mechanism is available.

A fully Fortran90 implementation has the advantage of incorporating derived types and dynamic memory allocation directly into RSL, allowing it to provide additional levels of support for managing multiple parallel domains in separate processor memories. One might, for example, construct must of the underlying structure of a parallel model simply by specifying to the run-time system a list of variables and their logical dimensions, giving the run-time system complete control over their allocation and reallocation, further relieving the programmer of the task of explicitly managing this (admittedly, this feature would be more useful for a new code than an existing code).

The long-range benefit to ceding this kind of control to a run-time system is that it hides more and more of the implementation details and makes the top-level description of

the program increasingly architecturally independent. The scientific community will benefit significantly if a single expression of a weather model can be made to run equally well on vector, RISC, shared-memory, and distributed-memory parallel architectures. This goal will prove elusive in the foreseeable future for the general class of scientific applications precisely because of their ambitious scope. However, a focused approach that addressed the needs and requirements of a particular type of application—in this case, grid-based weather and related codes—has a good chance of success in the near term. Elements of a total system will include source translation and run-time system technology augmenting basic compiler technology. Therefore, the run-time system library approach of RSL represents a step in the direction toward architecture independence.

Acknowledgments

Appendix

The following is a Fortran90 implementation of the example code in Section 5. Each model domains is represented as a Fortran90 derived type, and the nesting hierarchy is implemented as pointers between these structures. The main computational part of the code is a recursive descent of this tree for each time step. The number of processors decomposing the code and the size of all data model data structures are defined at run-time and dynamically allocated. The data for Figure 15 was generated by using this program on four processors of an IBM SP2 computer.

Implementation note: the IBM Fortran compiler is already F90 compliant and supports recursion. However, stack allocation of local variables may not be the default at every installation. Therefore, the code should be compiled with `-qnosave`.

```
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC FORTRAN 90 MODULE DEFINITION FOR A 'DOMAIN'
CC
CC User program definition of domain data structure and
CC manipulation routines.
CC
      module domains_module
        integer :: maxkids              ! maximum number of children
        parameter ( maxkids = 5 )
C
C Domainstruct is the principal domain data structure in the user
C program.  It contains the size of the domain in local
C memory and logically.  Also pointers to parent and child domains,
C if any.  Also the RSL domain handle.  Also, the domain state
C arrays themselves (unallocated, so virtually no storage expended
C unless the domain is actually used).
C
        type domainstruct
C
```

```
C This section has information about the domain.
C
          type( domainstruct ), pointer ::
     $           parent,        ! parent domain
     $           child(:)       ! children
          integer nkids         ! number of (active) children
          integer nestlevel     ! nestlevel of this domain
          integer domdesc       ! RSL domain handle
          logical active        ! flag
          integer m, mloc       ! global and local dimensions in m
          integer n, nloc       ! global and local dimensions in n
          integer sten          ! stencil descriptor
C
C This section has the state arrays for the domain.
C
          real, pointer    :: X(:,:)
        endtype domainstruct
C
C Declaration of the top-level domain (all others are children of this one)
C
        type ( domainstruct ), target :: mother
C
C Domain manipulation routines
C
        contains
C
C Initialize a domain data structure
C
        subroutine init_domain( d )
          type(domainstruct) :: d
          d%active = .false.
          d%m = 0
          d%n = 0
        end subroutine
C
C Allocate the fields of the domain (including state arrays).  Once
C this is called, the domain will take more than nominal storage.
C
        subroutine allocate_domain( d, m, n, mloc, nloc )
          type(domainstruct) :: d       ! domain structure
          integer m, n, mloc, nloc      ! global and local sizes
          integer k                     ! child index

          if ( d%active ) then
            write(0,*) 'allocate_domain: domain already active.'
            stop
          endif
          write(0,*)'allocate_domain ',d%domdesc,m,n,mloc,nloc
C
C Create and initialize structures for future children.   Note: only
C the child domain structures and not the child state arrays are
C being allocated here -- thus, the children require only nominal
C storage until actually activated and allocated.
C
```

```
          allocate (d%child(maxkids))
          do k = 1, maxkids
            call init_domain( d%child(k) )
          enddo
C
C Set the size information about this domain
C
          d%m = m
          d%n = n
          d%mloc = mloc
          d%nloc = nloc
          d%active = .true.
          d%nkids = 0
C
C Allocate storage for state arrays.
C
          allocate (d%X(mloc,nloc))
        end subroutine
      end module
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC MAIN PROGRAM
CC
CC This is the main routine for a Fortran-90/RSL implementation of
CC a simple relaxation problem on a grid, with an irregularly shaped
CC nested domain.
CC
CC The bulk of the main routine, here, is involved with defining
CC and allocating the problem, which can be done entirely at run-time
CC by exploiting modern features of F90 such as derived data types
CC and dynamic memory allocation.
CC
CC The model itself is executed in one call to another routine,
CC ITERATE_MODEL
CC at the bottom of this main program.
CC
      program relaxmain
      use domains_module
      implicit none
      type(domainstruct), pointer :: d
#include "rsl.inc"
      integer m,n,mloc,nloc,did         ! coarse domain params
      integer m_n,n_n,mloc_n,nloc_n,nid ! nested domain params
      integer mtrim, ntrim              ! trim for the nest
      integer xlist(20), ylist(20), npoints   ! outline of nest
      integer iter                      ! number of iterations to perform
      integer k                         ! child index
      integer nproc_m, nproc_n          ! number of procs in m, n
C
C Initialize RSL
C
      call rsl_initialize
```

```
C
C Input run-time problem configuration information
C
      if ( rsl_iammonitor() ) then
        read(7,*)m,n              ! mother domain size specif. at run time
        read(7,*)iter             ! number of mother domain iterations
        read(7,*)nproc_m, nproc_n ! number of processors specif. at run time
      endif
      call rsl_mon_bcast( m, 4 )
      call rsl_mon_bcast( n, 4 )
      call rsl_mon_bcast( iter, 4 )
      call rsl_mon_bcast( nproc_m, 4 )
      call rsl_mon_bcast( nproc_n, 4 )
      call rsl_mesh(nproc_m, nproc_n)
      call rsl_error_dup()

C
C Mother domain.  Note that the RSL routine is called before the
C domain is allocated.  This allows the local state arrays to be
C allocated only as large as necessary (using the size information
C returned by rsl_mother_domain in mloc and nloc).
C
      call init_domain(mother)
C
      call rsl_mother_domain(
     $                       did,               ! output: RSL domain handle
     $                       RSL_8PT,           ! input: max stencil
     $                       m, n,              ! input: global size
     $                       mloc, nloc )       ! output: local size

      call show_domain_decomp( did )
C
      call allocate_domain( mother, m, n, mloc, nloc )
      mother%domdesc = did                      ! store RSL handle in domain
      mother%nestlevel = 1                      ! nest level of mother is 1
C
C Specify the outline of the nest in coarse domain coordinates
C
      xlist(1) =  4   ; ylist(1) =  6
      xlist(2) =  4   ; ylist(2) =  11
      xlist(3) =  7   ; ylist(3) =  11
      xlist(4) =  10  ; ylist(4) =  14
      xlist(5) =  15  ; ylist(5) =  14
      xlist(6) =  15  ; ylist(6) =  9
      xlist(7) =  10  ; ylist(7) =  4
      xlist(8) =  6   ; ylist(8) =  4
      xlist(9) =  4   ; ylist(9) =  6
      npoints = 9
C
C Specify the "trim" for the nest
C
      mtrim = 2
      ntrim = 2
C
```

```
C Spawn an irregular nest using the outline and trim information
C specified above.
C
      call rsl_spawn_irreg_nest(
     $              nid,                         ! output: nest handle
     $              mother%domdesc,              ! input: parent handle
     $              RSL_8PT,                     ! input: max stencil
     $              xlist, ylist, npoints,       ! input: domain outline
     $              3, 3,                        ! input: nesting ratios
     $              mtrim, ntrim,                ! input: trim effective size
     $              mloc_n, nloc_n,              ! output: local memory size
     $              m_n, n_n )                   ! output: global size

      call show_domain_decomp( nid )
C
C Allocate using local size information returned by RSL.
C
      call allocate_domain( mother%child(1), m_n, n_n, mloc_n, nloc_n )
C
C Set fields in domain structures associating nest with parent
C
      mother%nkids = 1                           ! mother has one nest
      mother%child(1)%parent    => mother        ! back pointer to parent
      mother%child(1)%domdesc   =  nid           ! store RSL handle
      mother%child(1)%nestlevel =  mother%nestlevel+1
C
C Define the stencil communications on the mother and all subnests
C
      call define_data( mother )                 ! user routine: recursive
C
C Initialize the interior of the mother and the boundary cells
C
      call init_grid( mother, mloc, nloc, 1 )
C
C Initialize the nests with data from the mother.  (There is only
C one nest in this example, but this code allows for more).
C
      do k = 1, mother%nkids
        call initial_nest_data( mother, mother%child(k) )
      enddo
C
C Write initial state of the model
C
      call output_domains( mother )              ! recursive
C
C Execute 'iter' iterations of the simulation.  The following call
C executes on the mother and recursively on all nests.
C
      call iterate_model( mother, 1, iter )      ! main time loop (recursive)
C
      stop
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

```
CC
CC ITERATE_MODEL
CC
CC Main computational routine -- loop over time, iteration over mother
CC and all subnests, and control of inter-domain data exchanges.
CC
      recursive subroutine iterate_model( d, iter1, itern )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d           ! input: domain
      integer iter1, itern              ! input: starting and ending steps

      integer t                         ! local: time step on this domain
      integer k                         ! local: child index


      do t = iter1, itern
        call relax_grid( d, d%mloc, d%nloc, 1 )         ! compute this domain
        do k = 1, d%nkids                               ! for each nest...
          call force_domain( d, d%child(k) )            !   force
          call iterate_model( d%child(k), 1, 3 )        !   RECURSIVE CALL
          call merge_domain( d, d%child(k) )            !   feedback
        enddo
        if ( d%nestlevel .eq. 1 .and.
     $      ((mod(t-1, 5) .eq. 0) .or. (t .eq. itern))
     $                                             ) then
          call output_domains( d )                      ! if time, output
        endif
      enddo
      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC RELAX_GRID
CC
CC This is the main computational routine that is called for one step
CC on a domain.  The new value for each point is computed as the
CC average of the values of 8 neighbors.  Prior to the computation,
CC a stencil exchange is performed to ensure that off-processor data
CC will be available for the computation.
CC
      subroutine relax_grid( d, ilen, jlen, klen )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d           ! input: domain being computed
      integer ilen,                     ! input: local array sizes in m
     $        jlen,                     ! input: local array sizes in n
     $        klen                      ! input: local array sizes in vertical
      RSL_RUN_DECL                      ! macro: declares looping information
      integer m, n, i, j                ! local: misc info
      real New( ilen, jlen )            ! local: temporary array
```

```
                                    ! local: proximity information
      integer bdyinfo(ilen,jlen,DOT_BDY_INFO_LEN)
      real, pointer :: X(:,:)           ! local: state array pointer
C
C Macro to initialize RSL loop constructs
C
      RSL_INIT_RUNVARS(d%domdesc)
C
C Get boundary proximity information for each cell from RSL.
C
      call rsl_get_bdy_larray( d%domdesc, bdyinfo, DOT_BDY_INFO_LEN )
C
C Exchange data with other processors on this domain using the stencil
C information defined for and stored with the domain.
C
      call rsl_exch_stencil( d%domdesc, d%sten )
C
C Set pointer to domain state array.  Size info.
C
      X => d%X
      m = d%m
      n = d%n
C
C Main loop over horizontal dimensions of partition of array that
C is stored on local processor.  If a boundary cell, hold fixed,
C otherwise compute average.  Boundary cells are those with a boundary
C proximity of zero (i.e. they are zero cells away from a boundary).
C
      RSL_DO_N(j)
        RSL_DO_M(i)
          if ( bdyinfo(i,j,RSL_DBDY) .eq. 1 ) then
             New(i,j) = X(i,j)
          else
             New(i,j) =  (
     $          X(i+1,j-1)   +  X(i+1,j)  +  X(i+1,j+1) +
     $          X(i,j-1)     +               X(i,j+1)   +
     $          X(i-1,j-1)   +  X(i-1,j)  +  X(i-1,j+1)
     $                 ) / 8.0
          endif
        RSL_ENDDO
      RSL_ENDDO
C
C Update X.
C
      RSL_DO_N(j)
        RSL_DO_M(i)
          X(i,j) = New(i,j)
        RSL_ENDDO
      RSL_ENDDO
C
      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

```
CC
CC OUTPUT_DOMAINS
CC
CC Called initially and periodically.  Outputs state of model on
CC to separate files for each nest level.
CC
      recursive subroutine output_domains( d )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d                     ! input: domain
      integer k                                   ! local: child index
      integer glen(2), llen(2)                    ! local: size arrays
C
      glen(1) = d%m
      glen(2) = d%n
      llen(1) = d%mloc
      llen(2) = d%nloc
C
C Output top level domain
C
      call rsl_write( 18+d%nestlevel-1,           ! Fortran unit for output
     $                IO2D_IJ,                    ! describe record
     $                d%X,                        ! data for record
     $                d%domdesc,                  ! domain descriptor
     $                RSL_REAL,                   ! type of each element
     $                glen, llen )                ! size info
C
C Foreach nest, output it and its subnests recursively.
C
      do k = 1, d%nkids
        call output_domains( d%child(k) )         ! recurse
      enddo
      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC INITIAL_NEST_DATA
CC
CC Set the cells in the nest to values transferred down from the parent.
CC This is called to initialize the nest.
CC
      subroutine initial_nest_data ( d, nst )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d, nst                ! input: parent and nest
      integer pi, pj, pig, pjg                    ! local: parent indices
      integer ni, nj, nig, njg                    ! local: nest indices
      integer m, n, i, j, msize                   ! local: misc variables
      integer cm, cn                              ! local: relative nest index
      integer retval                              ! local: return value
      real, pointer :: X(:,:)                     ! local: pointer to state array
```

```
C
C Point to parent's state arrays.
C
      X => d%X
C
C Build a message for each point on the nest using data from the
C overlying cell in the parent domain.  Loop goes until we have handled
C data from all the parent domain points on this processor.
C
      call rsl_to_child_info( d%domdesc, nst%domdesc, 4,
     $                        i, j, pig, pjg, cm, cn, nig, njg, retval )
      do while ( retval .eq. 1 )
        call rsl_to_child_msg( 4, X(i,j) )
        call rsl_to_child_info( d%domdesc, nst%domdesc, 4,
     $                          i, j, pig, pjg, cm, cn, nig, njg, retval )
      enddo
C
C Exchange the data using RSL inter-domain communication.
C
      call rsl_bcast_msgs
C
C Now, point to nest state data
C
      X => nst%X
C
C Unpack the message on each point of the nest.  Loop goes until we
C have unpacked all the nested domain points that are local to this
C processor.
C
      call rsl_from_parent_info( i, j, nig, njg, cm, cn, pig, pjg, retval )
      do while ( retval .eq. 1 )
        call rsl_from_parent_msg( 4, X(i,j) )
        call rsl_from_parent_info( i, j, nig, njg, cm, cn, pig, pjg, retval )
      enddo
C
      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC FORCE_DOMAIN
CC
CC Similar to init_domain, except this is called at
CC each step to force only the boundaries of the nest
CC (not the entire domain, as in init_domain).
CC
      subroutine force_domain ( d, nst )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d, nst              ! input: parent and nest
      integer pi, pj, pig, pjg                  ! local: parent indices
      integer ni, nj, nig, njg                  ! local: nest indices
      integer m, n, i, j, msize                 ! local: misc variables
```

```
      integer cm, cn                            ! local: relative nest index
      integer retval                            ! local: return value
      real, pointer :: X(:,:)                   ! local: pointer to state array
      integer bdyinfo(DOT_BDY_INFO_LEN)         ! local: boundary proximity
C
C Point to parent's state arrays.
C
      X => d%X
C
C Build a message for ONLY THOSE POINTS on the nest that are on a boundary.
C The call to rsl_get_bdy_gpt gets the proximity information for a nested
C point.  The information is then used to decide whether or not data should
C be packed for that point.  Note: RSL will automatically size the messages
C between processors to exchange only that data that is packed.
C
      call rsl_to_child_info( d%domdesc, nst%domdesc, 4,
     $                        i, j, pig, pjg, cm, cn, nig, njg, retval )
      do while ( retval .eq. 1 )
        call rsl_get_bdy_gpt( nst%domdesc, bdyinfo,
     $                        DOT_BDY_INFO_LEN,nig, njg )
        if ( bdyinfo(RSL_DBDY) .eq. 1 ) then
          call rsl_to_child_msg( 4, X(i,j) )
        endif
        call rsl_to_child_info( d%domdesc, nst%domdesc, 4,
     $                          i, j, pig, pjg, cm, cn, nig, njg, retval )
      enddo
C
C Exchange the data using RSL inter-domain communication.
C
      call rsl_bcast_msgs
C
C Now, point to nest state data
C
      X => nst%X
C
C Unpack the message on each point of the nest.  Because the first phase
C of the exchange packed only data for the boundary points, this loop
C will iterate only over those.
C
      call rsl_from_parent_info( i, j, nig, njg, cm, cn, pig, pjg, retval )
      do while ( retval .eq. 1 )
        call rsl_from_parent_msg( 4, X(i,j) )
        call rsl_from_parent_info( i, j, nig, njg, cm, cn, pig, pjg, retval )
      enddo

      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC MERGE_DOMAIN
CC
CC This implements the feedback of data from the nest to the parent.
CC The structure is similar to FORCE_DOMAIN, except that the flow
```

```
CC of information is in the opposite direction.  Data is returned only
CC for the center nest point under each coarse domain point (cm=2, cn=2).
CC
      subroutine merge_domain ( d, nst )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d, nst              ! input: parent and nest
      integer pi, pj, pig, pjg                  ! local: parent indices
      integer ni, nj, nig, njg                  ! local: nest indices
      integer m, n, i, j, msize                 ! local: misc variables
      integer cm, cn                            ! local: relative nest index
      integer retval                            ! local: return value
      real, pointer :: X(:,:)                   ! local: pointer to state array
C
      X =>nst%X
      call rsl_to_parent_info( d%domdesc, nst%domdesc, 4,
     $                         i, j, nig, njg, cm, cn, pig, pjg, retval )
      do while ( retval .eq. 1 )
        if ( cm .eq. 1 .and. cn .eq. 1 ) then
          call rsl_to_parent_msg( 4, X(i,j) )
        endif
        call rsl_to_parent_info( d%domdesc, nst%domdesc, 4,
     $                           i, j, nig, njg, cm, cn, pig, pjg, retval )
      enddo
C
      call rsl_merge_msgs
C
      X => d%X
      call rsl_from_child_info( i, j, pig, pjg, cm, cn, nig, njg, retval )
      do while ( retval .eq. 1 )
        if ( cm .eq. 2 .and. cn .eq. 2 ) then
          call rsl_from_child_msg( 4, X(i,j) )
        endif
        call rsl_from_child_info( i, j, pig, pjg, cm, cn, nig, njg, retval )
      enddo
C
      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC INIT_GRID
CC
CC This is a computational routine whose purpose it is to assign
CC a domain with initial values.  Boundary cells receive a nonzero
CC initial value.  The interior receives a zero value.
CC
      subroutine init_grid( d, ilen, jlen, klen )
      use domains_module
      implicit none
#include "rsl.inc"
      type(domainstruct) :: d
      integer ilen, jlen, klen
```

```
      RSL_RUN_DECL
      integer bdyinfo(ilen,jlen,DOT_BDY_INFO_LEN)
      integer m, n, i, j
      integer cn, cm
      real, pointer :: X(:,:)
C
      RSL_INIT_RUNVARS(d%domdesc)
      call rsl_get_bdy_larray( d%domdesc, bdyinfo, DOT_BDY_INFO_LEN )
C
      X    => d%X
      m = d%m
      n = d%n
C
      RSL_DO_N(j)
        RSL_DO_M(i)
          if ( bdyinfo(i,j,RSL_DBDY) .eq. 1 ) then
              X(i,j) = 10.0
          else
              X(i,j) = 0.0
          endif
        RSL_ENDDO
      RSL_ENDDO
C
      return
      end
CC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CC
CC DEFINE_DATA
CC
CC This is called only once, prior to the first step of the model, and
CC is used to define the stencils on all the domains in the simulation.
CC Stencils, though identical, must be assigned individually for each
CC domain.  This routine does that recursively for the domain 'd', and
CC all nests under 'd'.
CC
      recursive subroutine define_data( d )
      use domains_module
#include "rsl.inc"
      type(domainstruct) :: d           ! input: domain
      integer decomp(3)                 ! local: how dimensions are decomposed
      integer llen(3)                   ! local: local size in each dim.
      integer glen(3)                   ! local: global size in each dim.
      integer mesg                      ! local: a message definition
      integer messages(8)               ! local: message for each stencil pt.
      integer k                         ! local: child index
C
      decomp(1) = RSL_NORTHSOUTH        ! m is decomposed by n/s processors
      decomp(2) = RSL_EASTWEST          ! n is decomposed by e/w processors
      glen(1) = d%m                     ! global sizes set
      glen(2) = d%n
      llen(1) = d%mloc                  ! local sizes set
      llen(2) = d%nloc
```

```
C
C Create and build a message descriptor containing the state array X.
C
      call rsl_create_message( mesg )
      call rsl_build_message( mesg,RSL_REAL,d%X,2,decomp,glen,llen )
C
C Create and build a stencil with the message on each of the 8 pts.
C
      call rsl_create_stencil( d%sten )
      messages(1) =    mesg
      messages(2) =           mesg
      messages(3) =                 mesg
      messages(4) =    mesg
      messages(5) =                 mesg
      messages(6) =    mesg
      messages(7) =           mesg
      messages(8) =                 mesg
      call rsl_describe_stencil( d%domdesc, d%sten, RSL_8PT, messages )

C
C Define the stencils for all the child domains of this domains
C
      do k = 1, d%nkids
        call define_data( d%child(k) )  ! RECURSION
      enddo
C
      return
      end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
```

References

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997. To appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P634-0197).

[2] I. Foster and J. Michalakes, *MPMM: A Massively Parallel Mesoscale Model*, in Parallel Supercomputing in Atmospheric Science, G.-R. Hoffmann and T. Kauranne, eds., World Scientific, River Edge, NJ 07661, 1993, pp. 354–363.

[3] G. A. Grell, J. Dudhia, and D. R. Stauffer, *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5)*, Tech. Rep. NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado, June 1994.

[4] P. L. Haagenson, J. Dudhia, G. A. Grell, and D. R. Stauffer, *The Penn State/NCAR Mesoscale Model (MM5), Source Code Documentation*, Tech. Rep.

NCAR/TN-328+STR, National Center for Atmospheric Research, Boulder, Colorado, March 1994.

[5] R. Hempel and H. Ritzdorf, *The GMD communications library for grid-oriented problems*, Tech. Rep. GMD-0589, German National Research Center for Information Technology, 1991.

[6] S. R. Kohn and S. B. Baden, *A Parallel Software Infrastructure for Structured Adaptive Mesh Methods*, in Proceedings of Supercomputing '95, IEEE Computer Society Press, 1996.

[7] J. Michalakes, *FLIC: A Translator for Same-source Parallel Implementation of Regular Grid Applications*, Tech. Rep. ANL/MCS-TM-223, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, March 1997.

[8] J. Michalakes, T. Canfield, R. Nanjundiah, and S. Hammond, *Parallel Implementation, Validation, and Performance of MM5*, in Coming of Age: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Procesors in Meteorology, World Scientific, River Edge, NJ, 1995, pp. 266–276.

[9] B. Rodriguez, L. Hart, and T. Henderson, *A Library for the Portable Parallelization of Operational Weather Forecast Models*, in Coming of Age: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Procesors in Meteorology, World Scientific, River Edge, NJ, 1995, pp. 148–161.