Abstractions to Separate Concerns in Semi-Regular Grids

Andrew Stone
Colorado State University
Computer Science Department
1873 Campus Delivery
Fort Collins, CO 80523
stonea@cs.colostate.edu

Michelle Mills Strout
Colorado State University
Computer Science Department
1873 Campus Delivery
Fort Collins, CO 80523
mstrout@cs.colostate.edu

ABSTRACT

In various applications including atmospheric and ocean simulation programs, stencil computations occur on semi-regular grids where subdomains of the grid are regular (e.g., can be stored in an array) but boundaries between sub-domains connect in an irregular fashion. Implementations of stencils on semi-regular grids often have grid topology details tangled with stencil computation code. This tangling of details makes updating stencil code difficult as it requires the programmer to have full knowledge of the current grid topology. Existing libraries and tools for separating the concerns of stencil computations from grid connectivity have not dealt with semi-regular grids and instead have focused on purely regular grids with possible periodicity or purely irregular grids. In this paper we introduce programming abstractions for the class of semi-regular grids and describe a prototype Fortran 90+ library called GridLib that implements these abstractions. Implementing these abstractions requires solving issues involving nodes in the grid with a non-standard number of neighbors and determining the communication schedule given an orthogonal specification of the grid decomposition. We present solutions to these issues that work within the context of grids used in atmospheric and ocean simulations. We also show that to maintain the performance while still providing this separation of concerns, it is necessary for a source-to-source translator to perform inlining between user code and the GridLib run-time library code. We present performance results for a stencil computation extracted from the Parallel Ocean Program.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'13, June 10–14, 2013, Eugene, Oregon, USA.
Copyright 2013 ACM 978-1-4503-2130-3/13/06 ...$15.00.

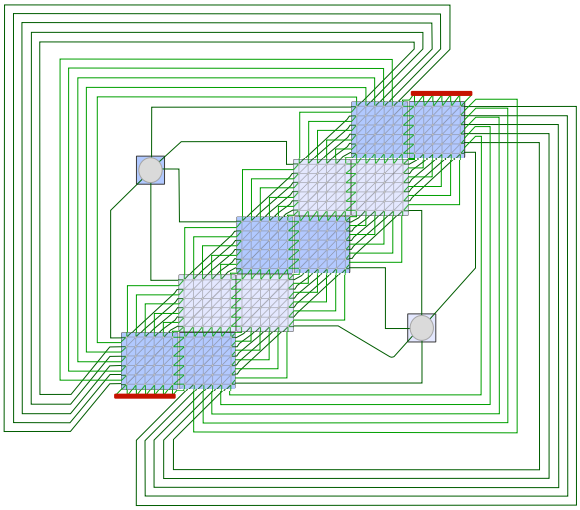


Figure 1: Connectivity of the Icosahedral grid. Each circular point in the figure represents a piece of data; the two large circles in the top-left and bottom-right corners represent data nodes for the north and south poles. Each of the blue rectangular regions represents a collection of data that is stored in an array. We illustrate six by six nodes in each non-polar array but in practice these arrays contain several thousands of nodes. The lines between nodes show connectivity. The lines going into and out of the red bars are connected.

Keywords

Programming models; Domain-specific languages; Compilers; High-performance computing; Parallel programming; Earth simulation; Atmospheric science

1. INTRODUCTION

Applications that simulate the Earth's atmosphere, land, and oceans are used to study problems in Geophysics, Climatology, Oceanography, and related fields. These simulation applications often require large amounts of computing and programmer resources. One of the main hurdles programmers face when developing Earth simulations is implementing globe discretization details in an efficient manner. The need for efficiency leads to a distributed memory parallel implementation and semi-regular grids, which are a patchwork quilt of regular arrays that connect in irregular ways. This causes the parallelization and grid connectivity details to

```
x2(:,:,:) = zero

! Iterate through subgrids, local blocks of data,
! vertical, and horizontal nodes.
DO g=1,10
 DO blk=lclBlk(g),lclBlk(g+1)
  DO j = 2,jm-1
   DO i = 2,im-1
     x2(blk,i,j) = area_inv(blk,i,j) * &
       ((x1(blk,i+1,j  )-x1(blk,i,j)) *
           weight(1,blk,i,j) + &
        (x1(blk,i+1,j+1)-x1(blk,i,j)) *
           weight(2,blk,i,j) + &
            ...
   END DO
  END DO
 END DO

 ! UPDATE NORTH POLE
 i = 2; j = jm; blk=1
 x2(blk,i,j) = area_inv(blk,i,j) *
               pweight(blk,i,j) * ...

 ! UPDATE SOUTH POLE
 i = im; j = 2
 x2(blk,i,j) = area_inv(blk,i,j) *
               pweight(blk,i,j) * ...
END DO
```

Figure 2: Fortran source code for stencil in SWM proxy application (extracted from GCRM).

```
real function x2(x1, i, j)
  ! ** Input parameters and local variables: **
  real, intent(inout) :: array(:, :)
  integer, intent(in) :: i, j

  x2 = &
    area_inv(i,j) * &
    ((x1(i+1,j  )-x1(i,j)) * weight(1,i,j) + &
     (x1(i+1,j+1)-x1(i,j)) * weight(2,i,j) + &
       ...
end function
```

Figure 3: Fortran source code for a simple stencil. The function x2 calculates the stencil for position (i,j) reading values from x1.

permeate application code. This paper introduces abstractions that enable an orthogonal specification of the stencil algorithm, the grid connectivity, and the grid decomposition for parallelization.

Figure 1 illustrates the semi-regular connectivity pattern of the icosahedral grid [17, 31], which is used in the Global Cloud Resolution Model (GCRM) [5]. As can be seen in the figure, the grid consist of twelve subgrids. Ten of these subgrids store data for pairs of triangular faces of an icosahedron and two of these subgrids store data for the north and south poles of the Earth.

Once the Earth is discretized and stored, applications iteratively perform operations on the discretized data to simulate some physical change over time. These operations typically take the form of a stencil computation. Stencil computations iterate over every point in a grid updating each value using values from neighboring nodes.

Figure 2 shows an example stencil from the Shallow Water Model (SWM) proxy application for GCRM. The code is executed in each process of an SPMD MPI program, where each process is responsible for computing over a subset of the blocks within the ten main subgrids.

Each block includes an extra layer (or halo) of array elements in all directions so that the stencil computation in the for loop can be the same at every location on the inside of the array except for the possible north and south pole nodes, which have their own specialized code.

As detailed on the icosahedral grid webpage [4], the software architects of GCRM have taken great care to map large chunks of the icosahedral grid to regular arrays and to keep the amount of special code needed for the stencil computations to a minimum. To handle nodes in the mesh that have five neighbors instead of the typical six, nodes with five neighbors have one of their neighboring halo values filled with a zero. In this manner the same six neighbor stencil can be used throughout the whole subdomain. The north and south pole singularities are mapped into the halo of one subdomain each and their five neighbors are placed so that they do not interfere with the rest of the subdomain. However, the north and south pole still require extra code as can be seen in Figure 2.

Even though the specialized code for the poles reflects a lack of separation of concerns between stencil code and the irregular grid connectivity, we do not show any communication code in Figure 2; this is because GCRM cleanly separates communication from stencil code. Nevertheless, the communication code that is in GCRM is highly tangled with information about the grid connectivity. This tangling of concerns leads to two main issues. (1) Whenever switching an application from one grid to another for scientific modeling reasons, most of the code in the application must be rewritten. The Parallel Ocean Program (POP) [19] provides an example of this. Between versions 1.4.3 and 2.0 the application was significantly rewritten to change from the dipole grid to the tripole grid. (2) The communication code, which is a significant portion of the full application, is currently written by hand for the specific grid and parallel distribution. In the 14,813 line SWM proxy application for GCRM, initializing and conducting communication involves 1,891 lines of code. In CGPOP [30], a 3,214 line miniapp of POP, initializing and conducting communication involves about 1,147 out of 3,214 lines of code. Our lines-of-code values were measured using the SLOCCount tool [2].

Ideally, programmers should be able to specify grid connectivity, stencil code, and parallel distribution orthogonally without having to write any specialized communication code. Figure 3 shows an idealized version of the stencil in the GCRM. Several libraries abstract parallelization details from programmers to enable easier implementation of certain types of code. Stencil libraries and generators such as Patus [11], Mint [32], and Physis [25], can be used to specify a stencil and its parallelization details orthogonally. Mesh based solver libraries such as Liszt [13], also allow for an orthogonalization of algorithm and grid. However, existing programming language work has not focused on the semi-regular grids seen in applications like GCRM or the Parallel Ocean Program. Additionally, some of these approaches would require rewriting full simulation applications in different languages, which is problematic.

To address these issues we introduce a set of abstractions for the orthogonal specification of semi-regular grid connectivity, stencil computations, and grid distribution for parallelism. With these abstractions we are able to maintain a clean separation of grid topology from stencil algorithm and parallel distribution and use a generic communication algo-

Construct a subgrid of size n by m:

```
subgrid_new(n,m)
```

Construct a border map between rectangular region (`x1_t`, `y1_t`, `x2_t`, `y2_t`) in the halo of subgrid `sg_t` (target) to rectangular region (`x1_s`, `y1_s`, `x2_s`, `y2_s`) in the halo of subgrid `sg_s` (source):

```
bmap_new(x1_t, y1_t, x2_t, y2_t, sg_t,
         x1_s, y2_s, x2_s, y2_s, sg_s)
```

Construct a new grid:

```
grid_new()
```

Add subgrid sg to the a grid g:

```
grid_addSubgrid(g, sg)
```

Add border map b to grid g:

```
grid_addBMap(g, b)
```

Figure 4: Interface for specifying semi-regular grid connectivity in GridLib library

rithm that will work for any semi-regular grid. We implement these abstractions in a Fortran library called GridLib.

A library based approach has the advantage that it can be integrated into existing Fortran programs and work with existing Fortran debugging and development tools. A disadvantage of a library based approach is that it introduces overhead. To alleviate this potential disadvantage we create a source-to-source translation tool called GridWeaver. GridWeaver inlines library calls and user-defined stencil class, removing overhead, and lets us produce code that performs competitively against hand-written code. We evaluate our tool by comparing performance against a benchmark stencil based on the Parallel Ocean Program.

We introduce our semi-regular grid abstractions in Section 2. In Section 3 we formalize these abstractions and describe how they are implemented; we also present an algorithm that calculates what communication must occur for a semi-regular grid. In Section 4 we discuss the need for a translation tool in order to maintain performance. We discuss related work in section 6 and conclude in Section 7.

2. SEMI-REGULAR GRID ABSTRACTIONS

In this section we describe abstractions that can be used to specify and operate on semi-regular grids. We also describe how to use these abstractions via functions calls to our GridLib Fortran library.

To perform a parallel stencil computation in GridLib we require users to specify three things: (1) grid connectivity, (2) grid decomposition, and (3) the stencil algorithm. The following subsections describe the relevant abstractions and library functions for each of these three requirements.

2.1 Specifying Connectivity

There are three main types of abstractions we use to specify connectivity of semi-regular grids: subgrids, border mappings, and grid objects. Subgrids represent a two-dimensional index space where a grid exhibits regular connectivity, border mappings represent connectivity between subgrids, and grids aggregate the subgrids and border mappings together to form a single object.

In Figure 4 we present the interface for defining connectivity in GridLib. Although GridLib is a Fortran 90 library we use an object-oriented approach in its design. The functions `subgrid_new`, `bmap_new`, and `grid_new` construct new subgrid, border-mapping, and grid objects respectively.

When constructing a new subgrid the user specifies the width and height of the subgrid. It is not necessary that all subgrids in a grid have the same width and height. For example, the connectivity of the Icosahedral grid depicted in Figure 1 consists of ten N by M subgrids representing the sides of an icosahedron and two one by one subgrids that represent the north and south poles.

Our current interface only supports subgrids that are two-dimensional, although our approach could be generalized to handle greater dimensionality. It should be noted that this limitation does not prevent stencils from operating on higher dimensional data: for example, an array of values, rather than a scalar, could be associated at each point of the grid. This limitation does prevent a grid from having differing connectivity along the higher dimensions or being decomposed along these dimensions; however, for the CG-POP and SWM applications two-dimensional subgrids are sufficient. In other Earth science applications there might be a column of data associated with each cell or node in the two-dimensional subgrid, but the connectivity between subgrids is still two dimensional in nature

In GridLib, subgrids are connected to one another via border mappings. Border mappings are based on the idea of specifying a layer of nodes, called a halo, that surrounds a subgrid and then indicating connectivity by specifying a target and source rectangular region. The target region lies in the halo of a given subgrid, the source region lies along the border of another (or potentially the same) subgrid. The mapping specifies that the region occupied by the target should be filled with values from the source region. For example, in Figure 5a the border mapping `bmap_new(3,1, 3,4, sg1, 1,1, 1,4, sg2)` indicates that the target rectangle in the halo of the left hand subgrid should be filled with the rectangle from the border of the right hand subgrid. With border mappings we are able to model the complex connectivity patterns seen along array borders in semiregular grids (see Figures 5b and 5c for more examples).

Typically the halo stores copies of data from neighboring subgrids. Once a halo's nodes are populated (copied) a stencil computation may be performed on a given subgrid without requiring access to any values outside the subgrid. In practice, subgrids are further divided into blocks that contain their own halos so that a finer granularity of parallelism can be achieved; however, for specification purposes the user only need think about a halo of nodes existing around the subgrid. In this paper and in our current implementation of GridLib we limit halos to be of single element depth. An implication of this is that stencil operations that work on the grid are required to be compact (only accessing nearest neighbor values). Future work will focus on relaxing this assumption.

Figure 5 illustrates different patterns of mapping that occur in Earth grids In this figure we also illustrate how each pattern is specified with border mappings and provide code that uses the interface in Figure 4 to specify the mapping. One thing to note is that the rectangular regions specified by the mappings have an orientation. In Figure 5 this orientation is illustrated using small triangles in one of the four

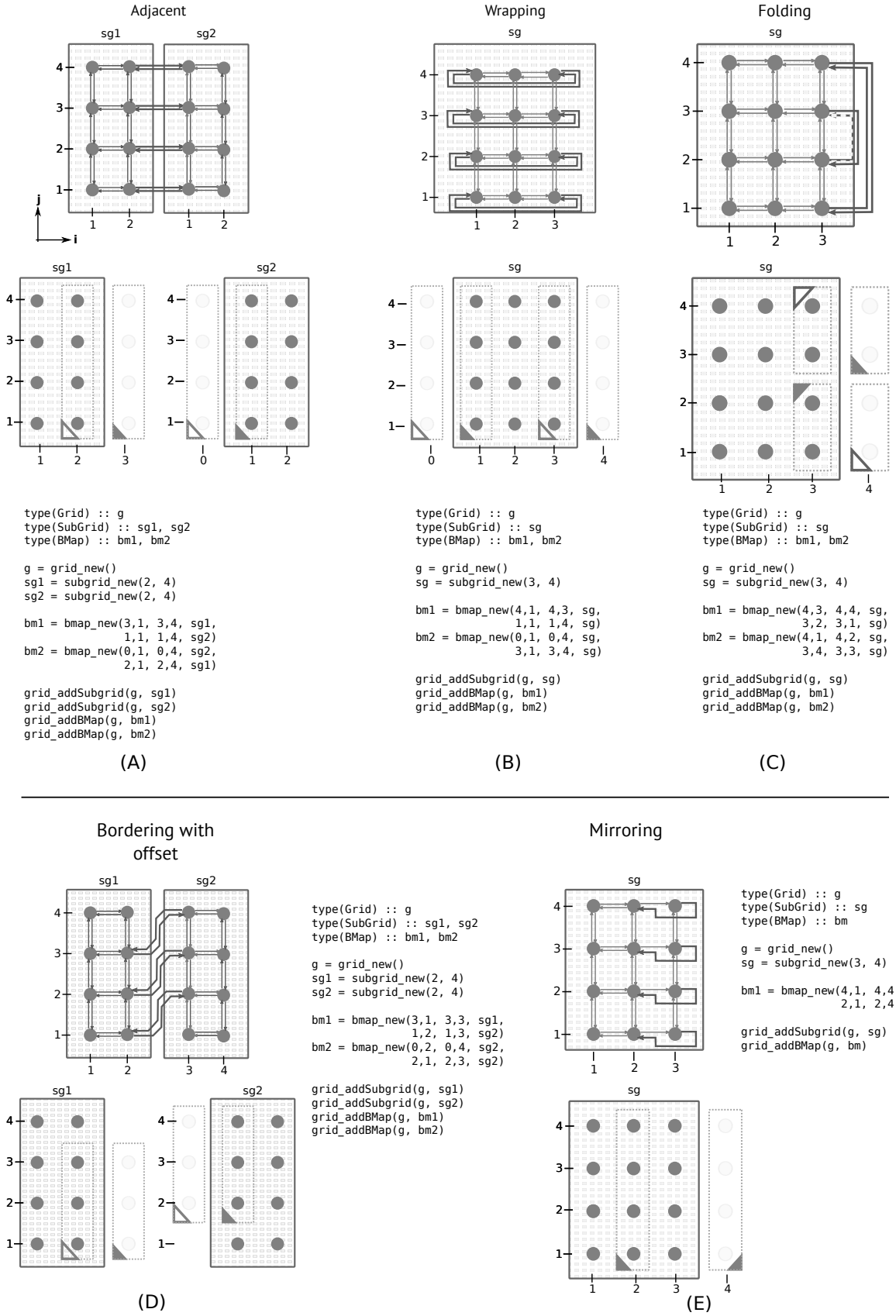


Figure 5: Examples of border connectivity patterns. In each subfigure we include an illustration of the connectivity pattern between nodes in the subgrid. Below each connectivity pattern illustration we show how the connectivity is represented using border mappings (the dashed rectangles). Below (in a,b, and c) or next (in d and e) to that we show code using the GridLib interface from Figure 4 that specifies the connectivity.

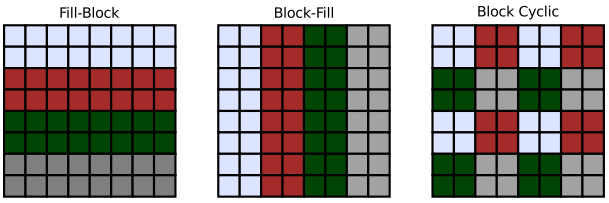


Figure 6: GridLib includes functions for decomposing an array into blocks in the typical block and cyclic combinations. Each color in this figure represents a different process.

corners of each source and target rectangle. The first coordinate of the specified target region corresponds to the first coordinate of the specified source region. Using the notation of the parameters passed to the `bmap_new` function in Figure 4, target point (`x1_t`, `y1_t`) corresponds to source point (`x1_s`, `y1_s`), target point (`x1_t`, `y1_t`) corresponds to source point (`x1_s`, `y1_s`), and `x1_s`, `y1_s`, `x1_t`, `y1_t` are not necessarily greater than `x2_s`, `y2_s`, `x2_t`, `y2_t` respectively.

The connectivity illustrated in a, b, c, and d in Figure 5 are seen in actual grids. The adjacency pattern in Figure 5a is seen in the cubed-sphere [28] and icosahedral grids [17, 31]. The wrapping pattern in 5b appears in toroidal and dipole grids [29]. The folding pattern in 5c appears in the tripole grid and the offset pattern seen in 5d appears in the icosahedral grid [26]. We include the mirroring pattern in 5e since it is seen in array-based languages such as ZPL [10].

2.2 Specifying Decomposition

Once a grid's connectivity is specified it is possible to assign values to each of its nodes. If a stencil computation is to run in serial then an array can be allocated for each subgrid and there is no need for further decomposition. On the other hand, if the stencil is to run in parallel then the grid must be decomposed into chunks that are distributed to each processing elements. In a shared memory environment this ownership identifies what nodes a processor is responsible for updating when conducting the stencil; in a distributed memory environment this ownership identifies both what nodes a processor is responsible for updating as well as what nodes that processor is responsible for storing.

A naive decomposition would be to assign ownership of one entire subgrid to each processing element. This decomposition does not scale since large machines contain several thousands more processing elements than there are subgrids; the most complex grid we have studied, the Icosahedral grid, consists of only twelve subgrids.

To address this concern GridLib provides several functions for decomposing a grid into blocks. Each of these functions will decompose the subgrids into equally sized blocks and assign ownership according to some pattern. In Figure 6 we illustrate three different decomposition patterns; these patterns are based on the array decomposition features of High Performance Fortran [23]. Users may also manually assign or modify the decomposition.

In line 45 of Figure 7 we use the `new_block_fill` function to specify a block-fill decomposition with blocks of size 64 by 64 on the tripole grid. In lines 9 through 28 we use the grid connectivity functions described in Section 2.1 to define the connectivity of the tripole grid.

```
1  module Stencils; contains
2      real function fivePtAvgStencil(A, i, j)
3          fivePtAvgStencil = 0.2 * &
4              (A(i, j) + A(i-1, j) + &
5              (i+1, j) + A(i, j-1) + A(i, j+1))
6      end function
7  end module
8  program StencilOnTripole
9      type(SubGrid) :: sg
10     type(BMap) :: w2e, e2w, fold1, fold2
11     type(Grid) :: g
12     type(Decomposition) :: dcmp
13     type(Data) :: data_in, data_out
14     integer :: N, M
15
16     ! Create subgrid
17     N = 2048; M = 2048
18     call subgrid_new(sg, N, M)
19
20     ! Create mappings for tripole connectivity
21     call bmap_new(w2e,    0, 1,     0, M,   sg,
22                         N+1, 1,   N+1, M,   sg);
23     call bmap_new(e2w, N+1, 1,   N+1, M,   sg,
24                           1, 1,     1, M,   sg);
25     call bmap_new(fold1, 1, M+1,  N/2,M+1, sg
26                          N, M,     N/2+1,M,    sg)
27     call bmap_new(fold2, N/2+1, M+1,  N, M+1, sg,
28                            N/2, M,     1, M,    sg)
29
30     ! Create a tripole grid
31     call grid_new(g)
32     call grid_addSubgrid(sg)
33     call grid_addBMaps(w2e, e2w, fold1, fold2);
34
35     ! Specify a decomposition and input data
36     dcmp = decomposition_new_block_fill(g, 64, 64)
37     data_in = data_new_from_file("input.dat", dcmp)
38     data_out = data_new(dcmp)
39
40     ! Perform stencil operation
41     data_out =
42         data_apply(data_in, fivePtAvgStencil)
43 end program StencilOnTripole
```

Figure 7: GridLib code for stencil on tripole grid. The `decomposition_new_block_fill` function applies a block-fill decomposition (illustrated in Figure 6) to grid g where each block is 64 by 64 nodes. The call to `data_new_from_file` reads in a data object from `"input.dat"` and stores the data using the previously defined decomposition. The `data_apply` function performs the stencil operation specified in the fivePtAvgStencil function using `data_in` for input values and `data_out` for output values.

2.3 Specifying Algorithm

Once decomposition is specified values may be assigned to grid node and stencil operations can be used to update these values. Figure 7 shows a complete example program that uses GridLib to perform a stencil operation on the tripole grid [26]. From a GridLib user's perspective there are two things that must be specified in order to perform a stencil operation: (1) instantiating data objects and (2) applying a stencil function.

Data objects map values to grid nodes. GridLib includes functions for reading and writing data objects to and from files. The functions that construct data objects are passed a decomposition object that specifies how to distribute the data. The `data_apply` function applies a stencil operation to every point in a data object. This function is passed (1) a data object that is referenced when grabbing values for a stencil operation and (2) a reference to a function that

performs the stencil operation for a given point. We apply this function in lines 41 through 42 in Figure 7.

The function that performs the stencil operation is specified by the user. This stencil function is passed the coordinates of a given point (`i`, `j`) and a function `A` that is used to retrieve the values of nodes neighboring (`i`, `j`). Despite the fact that we implement A as a function we encourage users to think of it as though it what an array. In this fashion the stencil code appears similar to what would be in the loop body of a serial stencil implementation on a purely regular grid. Note the similarity in the simplicity of the specification of the `fivePtAvgStencil` function and the code in Figure 3.

3. IMPLEMENTING ABSTRACTIONS

In the previous section we introduced abstractions for specifying the connectivity of semi-regular grids. These abstractions, which we have implemented in the GridLib library, enable a separate specification of grid topology from stencil code. In this section we describe issues that emerge when implementing these abstractions. Specifically we look at: (1) how to handle stencil nodes that have less than the typical number of neighbors, and (2) how to automate communication.

3.1 Handling Non-Standard Stencil Nodes

Due to the fact that subgrids represent regular portions of a grid, all elements in a subgrid that are not along one of its four borders will have the same connectivity pattern. Nodes along a border have at most the same number of neighbors. For example, the poles seen in Figure 1 have five neighbors while most other nodes in the grid have six.

When a stencil computation attempts to access a non-existent neighbor GridLib will return a value of zero. The user is responsible for writing the stencil computation so that it computes the correct value given this behavior. Since most stencils are written as linear combinations where the value of each neighboring node is multiplied by some coefficient the product for a non-existent neighbor will also be zero. We have found this behavior correctly implements the stencils in GCRM and CGPOP across all nodes. To ensure this behavior, prior to conducting communication, we initialize all values in a halo to be zero.

3.2 Communication Planning for Halos

We have found one of the most complicated portions of implementing semi-regular grid computations to be handling communication. Communication code is typically long and specialized for the grid used: in SWM (a 14,813 line proxy application of GCRM) communication takes 1,891 lines of code; in CGPOP [30] (a 3,214 line miniapp of POP) communication takes 1,147 lines of code.

Communication in these applications and in GridLib is cleanly separated by making use of halo regions around blocks of data. However, in SWM and CGPOP this code is written to be specific to the grid used. In GridLib we have implemented a generic communication algorithm that will populate block halos for any semi-regular grid. Once the halo is populated a stencil computation may update the values within a block without conducting any additional communication.

In this section we introduce an algorithm, used by GridLib, that will properly populate halo values for a semi-regular grid. To do this we begin by formalizing the semi-regular grid abstractions introduced in Section 2. The way GridLib stores and constrains its abstractions is based on these formalisms. We write the communication algorithm (shown in Algorithm 1) in terms of these formalisms.

3.2.1 Formalisms for semi-regular grid abstractions

We define a subgrid σ as an index space

$$\sigma = \{s, \langle i, j\rangle \mid s, i, j \in \mathbb{Z} \wedge 1 \leq i \leq N \wedge 1 \leq j \leq M\}, \quad (1)$$

where N and M are bounds on the index space, and s is an integer used to uniquely identify the subgrid. Thus all nodes in the same σ have the same value for s.

All subgrids have an associated halo and border-point index space. The halo of a subgrid σ of size N by M with identifier s is $h(\sigma)$ where

$$h(\sigma) = \{s, \langle i, j\rangle \mid s, i, j \in \mathbb{Z} \wedge 0 \leq i \leq N+1 \wedge 0 \leq j \leq M+1\} - \sigma \quad (2)$$

Border mappings are used to determine how nodes in a subgrid σ_1's halo $h(\sigma_1)$ map to nodes that lie within subgrid σ_2. Note that border mappings can apply to the same subgrid (that is σ_1 may equal σ_2). A border mapping β (equation 7) is defined in terms of two rectangular regions ρ_{tgt} and ρ_{src} (equations 5 and 6) that respectively lie in $h(\sigma_1)$ and σ_2 (equations 3 and 4) where

$$(s_1, p_i, p_j), (s_1, q_i, q_j) \in h(\sigma_1) \quad (3)$$
$$(s_2, v_i, v_j), (s_2, w_i, w_j) \in \sigma_2 \quad (4)$$
$$\rho_{\text{tgt}} = (s_1,\ p_i, p_j,\ q_i, q_j) \quad (5)$$
$$\rho_{\text{src}} = (s_2,\ v_i, v_j,\ w_i, w_j) \quad (6)$$
$$\beta = (\rho_{\text{tgt}}, \rho_{\text{src}})\,. \quad (7)$$

It is also necessary that ρ_{tgt} and ρ_{src} to have the same size.

When mapping nodes to ρ_{tgt} from ρ_{tgt} we may need to deal with regions that are oriented differently. To translate a point from one orientation to another we multiply it by an orientation matrix. The orientation function o returns the matrix for a given rectangle. We define o as

$$o(\rho) = \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } q_i \geq p_i \wedge q_j \geq p_j \\ \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } q_i \leq p_i \wedge q_j \geq p_j \\ \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} & \text{if } q_i \leq p_i \wedge q_j \leq p_j \\ \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} & \text{if } q_i \geq p_i \wedge q_j \leq p_j \end{cases} \quad (8)$$

For a given border mapping β there is a border mapping function $m : \beta \times \mathbb{Z}^3 \to \mathbb{Z}^3$ where

$$m(\beta, s_1, \langle i, j\rangle) = \left(s_2, \langle i', j'\rangle\right) \quad (9)$$

such that

$$\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} v_i \\ v_j \end{pmatrix} + o(\rho_h) o(\rho_b) \left(\begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} p_i \\ p_j \end{pmatrix} \right). \quad (10)$$

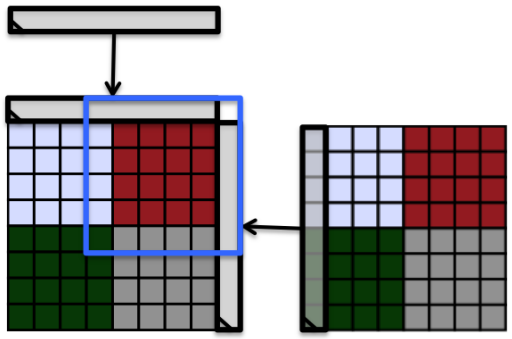


Figure 8: Halo of a block in a sub-grid. When populating the red block's halo some values come from neighboring blocks in the same sub-grid, other come from outside the subgrid.

This mapping function takes a point in a halo $(s_1, \langle i, j\rangle)$ and determines the location $(s_2, \langle i', j'\rangle)$ of the value that should be copied into the halo point.

We define a grid as a tuple $\gamma = (S, B)$ where S is a set of subgrids and B is a set of border mappings. All subgrids $\sigma \in S$ have a unique id s. A grid defines a global index space g such that

$$g(\gamma) = \bigcup_{\sigma \in S} \sigma. \tag{11}$$

The abstraction we use for assigning grid nodes to processes is the decomposition. Our decomposition abstraction consists of a set of blocks. We define a block θ as a tuple:

$$\theta = (b, p, s, \langle u_i, u_j\rangle, \langle v_i, v_j\rangle), \tag{12}$$

where $\langle u_i, u_j\rangle$ to $\langle v_i, v_j\rangle$ is a rectangular region in subgrid s. All blocks are given a unique identifier b and are assigned to a process p.

We define B to be a set of blocks. To simplify later abstractions and their implementations we constrain all blocks to be the same size and require that the union of blocks in B span the space $g(\gamma)$ of some grid γ.

Now that we have defined formalisms for grid connectivity and data decomposition we can describe the formalisms used to perform a stencil operation.

For a given grid γ we may define one or more data objects δ over that grid. Each point in the grid's index space $x \in g(\gamma)$ has an assigned value $\delta(x)$. The function δ has the signature

$$\delta(x) : \mathbb{Z}^3 \rightarrow \mathbb{R}. \tag{13}$$

To store data we use a three dimensional array where two of the dimensions correspond to values in the two-dimensional block and the third dimension corresponds to a block's local ID.

3.2.2 Algorithm and formalisms for communication

Around each of the stored blocks we include a single-node depth halo. The halo stores a copy of data from surrounding blocks. Once the halo is populated stencil operations can be applied independently to all nodes in each block. The GridLib library is currently limited to computations that have a single layer of halo depth. Due to the complex connectivity patterns that exist between grids handling larger

Input: A Grid G and distribution D
Output: A communication plan for each (MPI) rank r
$\pi(r) = (\mu_r(r), \mu_s(r))$.

let μ_r(r) be receiving messages for rank r
let μ_s(r) be sending messages for rank s
let B be the set of border mappings

! Determine the communication that occurs to populate
! each block's halo.
foreach *block* b_{tgt} *in the grid* **do**
  s = the subgrid b resides in
  r_r = the rank that owns block b_{tgt}
  b_{i1} = global i coordinate of block b_{tgt}
  b_{j1} = global j coordinate of block b_{tgt}
  b_{i2} = global i extent of block b_{tgt}
  b_{j2} = global j extent of block b_{tgt}

  ! Construct a region that includes all internal and
  ! halo nodes for block b_{tgt}
  $\rho_{\text{blkHalo}} = (s, \langle b_{i1} - 1, b_{j1} - 1\rangle, \langle b_{i2} + 1, b_{j2} + 1\rangle)$

  ! Determine the blocks that b_{tgt} *communicates with*
  O = set of blocks overlapping with ρ_{blkHalo}

  ! For each of the neighboring blocks create a message
  ! for the region in $\rho_{blkHalo}$ *that it overlaps with.*
  foreach $b_{src} \in O$ **do**
    r_{tgt} = rank that owns block b_{tgt}
    r_{src} = rank that owns block b_{src}
    ρ_{blk} = region spanned by block b_{src}
    ρ_o = region of ρ_{blkHalo} intersecting with ρ_{blk}

    if ρ_o *is contained in* s **then**
      ! Create a message within the subgrid
      $\mu_r(r_r) = \mu_r(r_r) \cup (b_{\text{tgt}},\ \text{tr}(\rho_o, b_{\text{tgt}}),\ r_{\text{src}})$
      $\mu_s(r_s) = \mu_s(r_s) \cup (b_{\text{src}},\ \text{tr}(\rho_o, b_{\text{src}}),\ r_{\text{tgt}})$
    else
      ! Create a message for each portion of
      ! ρ_o *that is in a border mapping*
      foreach $\beta \in B$ **do**
        if ρ_o *intersects with* β **then**
          ρ_m = intersection of β and ρ_o
          $\mu_r(r_r) = \mu_r(r_r) \cup$
            $(b_{\text{tgt}},\ \text{tr}(\text{rm}(\rho_m, \beta), b_{\text{tgt}}),\ r_s)$
          $\mu_s(r_s) = \mu_s(r_s) \cup$
            $(b_{\text{src}},\ \text{tr}(\text{rm}(\rho_m, \beta), b_{\text{src}}),\ r_r)$
        end
      end
    end
  end
end

Algorithm 1: Algorithm to generate a communication plan. The function tr(ρ, b) returns a region that translates the global indices used in ρ into the local indices used in block b. The function rm(ρ, β) returns a region where each of the four coordinates of ρ are passed through the border mapping function m.

halo depths is a more difficult and the algorithms presented in this paper do not easily scale to multi-depth halos. In Figure 8 we illustrate the halo region for a single block in a subgrid. Notice that some of the values needed to populate the halo come from blocks within the same subgrid and other values come from values on a different subgrid.

Prior to conducting communication we initialize values in the halo to zero. In this manner if an (n+1)-point stencil operation is applied to a point with less than n neighbors it will return zeroes for the missing neighbors. We have found this behavior correctly implements the stencils in GCRM and CGPOP across all nodes.

To populate the halo we conduct communication GridLib uses an abstraction internal to itself called a communication plan. This object is used to guide a message passing communication between processes. In this communication plan we identify each process with a rank r.

Each rank has a communication plan π, which is a tuple

$$\pi_r = (\mu_r, \mu_s), \tag{14}$$

where μ_r is the set of messages to receive and μ_s is the set of messages to send. A message μ is a tuple

$$\mu = (b, r, \rho), \tag{15}$$

where r is a rank that the message is received from if $\mu \in M_s$ or sent to if $\mu \in M_r$, b is the block to place data into if $\mu \in M_s$ or send data from if $\mu \in M_s$, and ρ is a rectangular region in block b. In Algorithm 1 we present a method for generating a communication plan object.

In our GridLib implementation we then populate a halo by simply posting MPI receive calls for each message in $\mu_r(r)$ on a given rank r, and post MPI sending calls for each message in $\mu_s(r)$ and copying the values read in from the received calls into the specified region of the specified block's halo. We perform this update halo operation each time a stencil is performed.

The algorithm will iterate through each block in the grid, calculate what region of a subgrid its halo covers, then determine what neighboring blocks overlap with this halo. If any portion of the block's halo extends beyond the boundary of a subgrid it will iterate through border mappings to determine if any of them intersect with the halo. After determining what blocks are neighboring a given block we construct messages in $\mu_r(r)$ and $\mu_s(r)$ to copy the neighboring data into the halo.

4. GRIDWEAVER TRANSLATION TOOL

We have developed a library called GridLib, which implements the abstractions introduced in this paper. The GridLib library enables a separate specification of semi-regular grids topology details from stencil algorithms. This specification enables users to specify a parallel stencil computation without requiring the user to write any parallel code.

A disadvantage of a library based approach is that libraries necessarily introduce overhead. In GridLib this overhead is introduced due to the fact that whenever a stencil computation is performed the user-defined function that encapsulates the stencil operation will be called for every point in the grid. Furthermore, within this function another function will be called whenever accessing the point's value or the value of any of its neighbors.

For example, the function `fivePtAvgStencil` will be called once for every point in the grid, and, although it syntactically appears like an array, the symbol `A` is actually function that returns the value associated with some grid point from data-structures internal to GridLib.

To address this overhead we have developed a code generation tool called GridWeaver that preprocesses programs written with GridLib andn inlines GridLib calls. To parse and translate Fortran programs we use the Rose source-to-source translation framework [27]. The name GridWeaver comes from both the notion of tying several subgrids together to form a single grid and the idea of weaving the GridLib library code into a user's program. An analogy can be drawn between this weaving process and the weaving of advice to joinpoints done by Aspect Oriented Programming [22] compilers.

More specifically, GridWeaver replaces calls to `data_apply` with a loop nest that will iterate over all locally owned nodes and will embed the contents of the stencil function within this loop nest. Gridweaver then replaces calls to the function `A` with code that directly accesses data from the array GridLib uses internally to store grid data. The inlining optimization is not performed automatically by Fortran compilers due to the fact that the stencil operation is specified outside of GridLib by its user but is invoked internally.

5. EVALUATION RESULTS

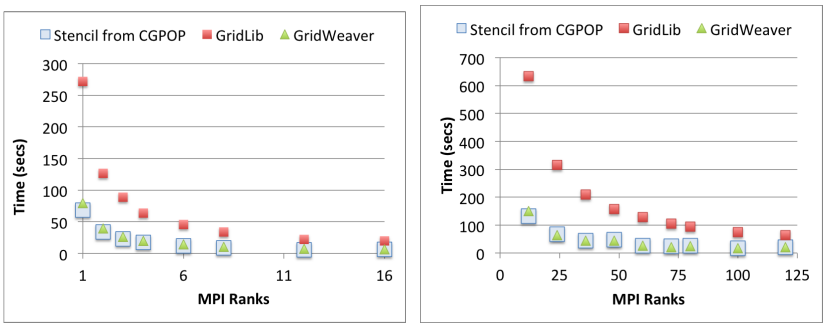
In Figure 9 we show a performance comparison of GridLib and GridWeaver against a stencil computation extracted from CGPOP. The extracted stencil is representative of CGPOP. In experiments run on a 2.8 GHZ Intel Core2 Machine CGPOP's kernel ran at performance of 2.35 GFLOP/s while the extracted stencil performed at 2.41 GFLOP/s. We performed this stencil for 100 iterations on a 3600 by 3600 dipole-grid with a row-blocked distribution.

We performed our scalability experiments on two machines at Colorado State University: Bacon, a 16 core, 2.13 GHz, Intel XeonE7 machine with 128 GB of memory; and the ISTeC Cray HPC system, a Cray XT6m with 1248 cores across 52 compute-node, and 1.6 TB of memory. There are 24 cores per node on the ISTeC Cray. This evaluation demonstrates that (1) GridLib introduces library overhead that reduces performance, particularly when run on small number of cores, and (2) GridWeaver is able to produce code that performs similarly to hand-written code. The GridWeaver tool is able to produce the efficient code by inlining the GridLib library calls as described in Section 4.

6. RELATED WORK

Several tools have been developed to aid with implementing stencil computations on structured or unstructured grids. These tools differ in terms of what types of grids they operate on, how grids and stencils are specified, and what platform and languages they target. For example, the framework from Kamil et al. [21], Mint [32], Patus [12], Physis [25], and Pochoir [24] work on regular grids, while Liszt [14], MPAS [3], and OP2 [15] work on irregular grids. Some grids, such as the torus, are mostly regular but include simple wrap-around (periodic) boundaries. Many allow periodic boundaries [11, 32, 25] but are unable to handle the other types of boundaries illustrated in Figure 5. Parallel array languages such as ZPL [10] also have the ability to express periodic boundaries.

The Chombo toolkit has the ability to work with grids that are composed of multiple subgrids. In Chombo a por-



(a) Performance on multicore (b) Performance on cluster

Figure 9: Performance comparison of GridLib and GridWeaver against the stencil extracted from CGPOP

tion of a regular grid may be adaptively refined to form an embedded, finer grained, subgrid. In such a case the coarser, enclosing, grid will copy data from the neighbor points in the finer, enclosed, grid; an averaging operator is used to translate between the finer and coarser data [6]. Chombo also the ability to allocate subgrids next one another through its Mapped Multiblock Grids abstraction. In this abstraction a series of grids in a two-dimensional domain is mapped to a three-dimensional range. Chombo will automatically determine how to interpolate and communicate values between subgrids that are adjacent in the three-dimensional range [1]. GridLib uses a different approach by having the user specify connectivity between subgrids directly

The Overture system also has the ability to work on grids that are composed of several subgrids. In Overture [7] users specify a geometry; Overture then discretizes that geometry with overlapping regular grids. As part of the discretization process connections between grids are determined and communication plans between grids are developed. It is unclear how one might adapt established semi-regular grids such as the icosahedral grid, cubed sphere, and tripole would be specified within the Overture framework. overture, More generally, to address software engineering concerns seen in Earth Science computations, researchers have developed component based software frameworks such as Cactus [16] and the Earth System Modeling Framework
(ESMF) [18]. These packages address the issues raised when composing different components of an Earth simulation application to one another (for example: how to compose a given sea model with an atmosphere model), and how to convert data from one gridded model to another (the regridding problem). However, these frameworks do not introduce to address the data-layout problem seen for a grid within a given component, which we address with this work.

The concept of using an abstraction to specify an array's decomposition is not unique to the GridLib and GridWeaver tools. The high-performance Fortran language [23] allows users to specify regular, block, cyclic, and block-cyclic distributions along each of an arrays dimensions. HPCS languages such as Chapel [8, 9] includes a first class distribution object that enable a separate specification of an array from its decomposition.

Other work has used the term semi-regular grid to refer to a regular grid that is then refined in places using a triangular mesh [33].

The GridWeaver code generator replaces calls to `data_apply` with a loop nest that iterates over grid nodes then embeds the body of the stencil function within the loop nest. Embedding the stencil function is similar to how iterators are inlined in Chapel and other languages [20].

7. CONCLUSIONS

In this paper we introduce abstractions for representing semi-regular grids. Semi-regular grids are used in Earth simulation programs and can be stored using one or more arrays. If these arrays are expanded to include a halo region then stencil computations can be applied without requiring any communication code within the stencil. We present a communication creation algorithm that determines how to populate the halos when the grid and the block distribution for the grid have been specified with orthogonal abstractions Communication plans are metadata used to direct send and receive calls. Using these abstractions and this communication plan generation algorithm GridLib enables users to specify grid connectivity separately from algorithm and alleviates the need to write communication code.

GridLib is a Fortran library. A Fortran library based approach is convenient in that it does not require users to rewrite their code if it is already in Fortran and in the fact that it will operate with existing Fortran debugging tools. A disadvantage of a library based approach is overhead introduced by the library. To alleviate the overhead we introduce a tool called GridWeaver that replaces the library calls with more efficient code. Existing libraries and tools for stencils have not focused on the class of semi-regular grids.

Current limitations of our approach include that it assumes that grid connectivity only differs along two dimensions and that halo size is one. Even with these limitations all of the stencils in CGPOP are expressible and all of the stencils except for one in SWM are expressible The abstractions could be relaxed to remove the first assumption without too much difficulty; however, we believe relaxing the second assumption will require more work due the fact that outgoing edges of subgrids often change directions. We intend our future work to focus on generalizing our abstractions to enable larger halos.

Acknowledgements

We would like to especially thank John Dennis and Ross Heikes for their help with the CGPOP and SWM proxy applications. The authors gratefully acknowledge the use of the ISTeC Cray at Colorado State University. This project is supported by a Department of Energy Early Career Grant DE-SC0003956.

8. REFERENCES

[1] Chombo: Mapped multiblock grids. `https://commons.lbl.gov/display/chombo/Mapped+Multiblock+Grids`.

[2] Sloccount tool website. `http://www.dwheeler.com/sloccount/`.

[3] MPAS: Model for prediction across scales. `http://mpas.sourceforge.net/`, December 2011.

[4] Spherical geodesic grids: A new approach to modeling the climate. `http://kiwi.atmos.colostate.edu/BUGS/geodesic/`, July 2012.

[5] Website: Design and testing of a global cloud resolving model. `http://kiwi.atmos.colostate.edu/gcrm/`, July 2012.

[6] M. Barad, P. Colella, D. T. Graves, T. J. Ligocki, D. Modiano, P. O. Schwartz, and B. V. S. and. EBChombo software package for cartesian grid, embedded boundary applications. Technical report, Lawrence Berkeley National Laboratory, February 2000.

[7] D. Brown, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for solving partial differential equations. In Y. Ishikawa, R. Oldehoeft, J. Reynders, and M. Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 177–184. Springer Berlin Heidelberg, 1997.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[9] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi. User-defined distributions and layouts in chapel: philosophy and framework. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[10] B. L. Chamberlain, S. eun Choi, S. J. Deitz, and L. Snyder. The high-level parallel language zpl improves productivity and performance. In *In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[11] M. Christen, O. Schenk, and H. Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Computer Science - Research and Development*, 26(3-4):205–210, Apr. 2011.

[12] M. Christen, O. Schenk, and H. Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Comput. Sci.*, 26:205–210, June 2011.

[13] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1—-9:12, New York, NY, USA, 2011. ACM.

[14] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[15] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. Performance Analysis and Optimization of the OP2 Framework on Many-Core Architectures. *The Computer Journal*, 55(2):168–180, July 2011.

[16] T. Goodale, G. Allen, J. Lanfermann, Gerd Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Proceedings of the 5th international conference on High performance computing for computational science*, pages 197—-227, 2003.

[17] R. Heikes and D. A. Randall. Numerical Integration of the Shallow-Water Equations on a Twisted Icosahedral Grid. Part II. A Detailed Description of the Grid and an Analysis of Numerical Accuracy. *Monthly Weather Review*, 123(6):1881–1887, 1995.

[18] C. Hill, C. DeLuca, M. Suarez, and A. Da Silva. The architecture of the earth system modeling framework. *Computing in Science & Engineering*, 6(1):18–28, Jan. 2004.

[19] P. Jones. Parallel Ocean Program (POP) user guide. Technical Report LACC 99-18, Los Alamos National Laboratory, March 2003.

[20] M. Joyner, B. L. Chamberlain, and S. J. Deitz. Iterators in chapel. In *Eleventh International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2006)*, 2006.

[21] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, and E. W. Bethel. A generalized framework for auto-tuning stencil computations. In *In Proceedings of the Cray User Group Conference*, 2009.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[23] C. Koelbel, D. B. Loveman, R. S. Schreiber, L. S. J. Guy, and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, September 1996.

[24] C.-k. Luk, B. C. Kuszmaul, and C. E. Leiserson. The Pochoir Stencil Compiler. *Artificial Intelligence*, 36:117–128, 2011.

[25] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[26] R. J. Murray. Explicit Generation of Orthogonal Grids for Ocean Models. *Journal of Computational Physics*, 126(2):251–273, 1996.

[27] D. Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France*, volume 10 of *Parallel Processing Letters*. Springer Verlag, 2000.

[28] R. Sadourny. Conservative Finite-Difference Approximations of the Primitive Equations on Quasi-Uniform Spherical Grids. *Monthly Weather Review*, 100(2):136–144, Feb. 1972.

[29] R. D. Smith and S. Kortas. Curvilinear coordinates for global ocean models. Technical report, Los Alamos National Laboratory, LA-UR-95-1146, 1995.

[30] A. Stone, J. Dennis, and M. M. Strout. The cgpop miniapp, version 1.0. Technical Report Technical Report CS-11-103, Colorado State University, July 1 2011.

[31] H. Tomita, M. Tsugawa, M. Satoh, and K. Goto. Shallow Water Model on a Modified Icosahedral Geodesic Grid by Using Spring Dynamics. *Journal of Computational Physics*, 174(2):579–613, 2001.

[32] D. Unat, X. Cai, and S. B. Baden. Mint. In *Proceedings of the international conference on Supercomputing - ICS '11*, page 214, New York, New York, USA, 2011. ACM Press.

[33] Z. J. Wood, P. Schröder, D. Breen, and M. Desbrun. Semi-regular mesh extraction from volumes. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 275–282, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.