DISSERTATION

SEPARATING IMPLEMENTATION CONCERNS IN STENCIL COMPUTATIONS FOR SEMIREGULAR GRIDS

Submitted by

Andrew Stone

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2013

Doctoral Committee:

Advisor: Michelle Mills Strout

Daniel Massey
Shrideep Palickara
David Randell

ABSTRACT

SEPARATING IMPLEMENTATION CONCERNS IN STENCIL COMPUTATIONS FOR SEMIREGULAR GRIDS

TIn atmospheric and ocean simulation programs, stencil computations occur on semiregular grids where subdomains of the grid are regular (i.e. stored in an array), but boundaries between subdomains connect in an irregular fashion. Implementations of stencils on semiregular grids often have grid connectivity details tangled with stencil computation code. When grid connectivity concerns tangle with stencil code it becomes difficult for programmers to modify the code. This is because any change made will have to account for grid connectivity. In this dissertation we introduce programming abstractions for the class of semiregular grids and describe a prototype Fortran 90+ library called GridWeaver that implements these abstractions. Implementing these abstractions requires determining the communication schedule given an orthogonal specification of the grid decomposition and solving nodes in the grid with a non-standard number of neighbors. 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 a separation of concerns, it is necessary for a source-to-source translator to perform inlining between user code and the GridWeaver runtime library code. We present performance results for stencil computations extracted from the Parallel Ocean Program and Global Cloud-Resolving Model.

ACKNOWLEDGEMENTS

Foremost I would like to thank my advisor Michelle Strout for her patience and guidance throughout my years as a graduate student at Colorado State University. I would also like to thank Dan Quinlan, Brad Chamberlain, and John Dennis for their guidance as internship mentors at Lawrence Livermore National Lab, Cray, and the National Center of Atmospheric Research.

John Dennis is also the originator and a collaborator on the CGPOP miniapp, which we use as an evaluation benchmark in this dissertation. Ross Heikes deserves acknowledgement for supplying and instructing me on the use of the Shallow Water Model code (another evaluation benchmark).

I would to acknowledge support for our research from Grant DE-SC0003956 from the US Department of Energy, and for support from ISTeC for access to their Cray machine at Cray at Colorado State University. The National Center for Atmospheric research provided us with access to their Yellowstone cluster through the "Automating efficient and scalable stencil implementation" project grant, and the San Diego Supercomputer Center provided access and support to the Gordon Cluster through a XSede Startup Allocation for the "Abstractions for Earth Science Simulations" project.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Scientists use applications that simulate Earth's atmosphere, land, and oceans to study problems in Geophysics [7], Climatology [10], Oceanography [52, 94], and related fields [133]. These applications help scientists gain insight into how environmental systems function and to make predictions about these systems' future conditions.

Earth science applications simulate environmental systems by solving partial differential equations (PDEs). These equations model a system's changing physical state over a span of time. Partial differential equations are often solved using implicit and explicit methods, which are algorithmically enacted using stencil computations [127, 146].

Stencil computations solve PDEs by iteratively updating values stored in cells. Each update calculates a weighted average of a cell and its neighboring values. Cells represent a discretized region of some physical object, and in Earth science applications cell values represent physical properties such as temperature, pressure, and wind speed.

In Figure 1.1 we illustrate an application of a stencil computation on a discretization of the Earth. The illustrated discretization partitions Earth into a finite number of hexagonal

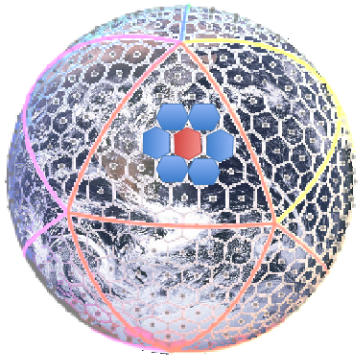


Figure 1.1: Illustration of the Earth discretized into hexagonal and pentagonal cells to form an icosahedral grid. The blue cells are immediate neighbors to the red cell.

and pentagonal cells. Sets of interconnected cells are referred to as *grids*, and the illustration in Figure 1.1 shows a three-dimensional view of an icosahedral grid. In the illustrated stencil application, the value for the red cell updates using values from the surrounding blue cells.

Scientists that use Earth simulation applications want fast and accurate results; thus these applications are compute-intensive, operate on large datasets, and are often executed on supercomputers [92]. One of the main hurdles programmers encounter when implementing efficient Earth-simulating applications is how to address globe discretization details. A common discretization strategy is to store grid-cell values in arrays rather than graph data structures. This improves performance since a stencil operation can access a cell's neighboring values directly, in the array, rather than indirectly through an adjacency list. Many Earth science grids consist of a series of regular subgrids, stored in arrays, that connect to each other in a patchwork fashion. The regular subgrids connect to each other to form a single, larger, *semiregular grid*.

When implementing semiregular-grid stencil-computations programmers often address discretization and parallelization details in a manner that specializes, permeates, and obfuscates code. We include an example of this in Section 1.1. In this dissertation, we aim to address these issues by introducing abstractions that enable orthogonal specifications of stencil algorithms from grid-connectivity and -decomposition.

1.1 Research problem

In this section we discuss an example that demonstrates how discretization and parallelization details permeate code. In Figure 1.2 we illustrate the semiregular connectivity of the icosahedral grid [86, 148]. The Global Cloud Resolution Model (GCRM) [5], an atmospheric circulation simulator, uses this grid. As drawn in Figure 1.2, the grid consist of twelve subgrids: ten that store data for pairs of triangular faces of an icosahedron, and two that store data for Earth's north and south poles.

Figure 1.3 shows an example stencil from the Shallow Water Model code (SWM): a proxy application of GCRM. We received this proxy application from David Randall and Ross

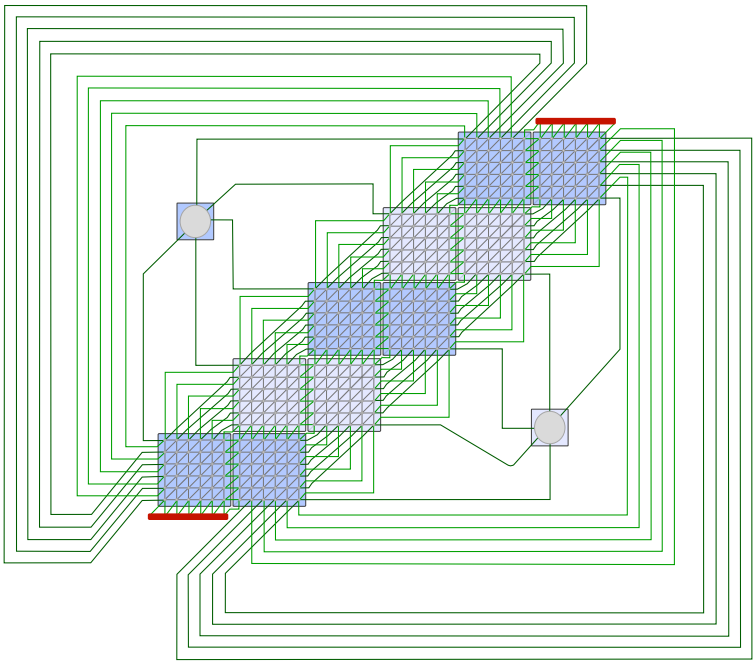


Figure 1.2: Connectivity of an icosahedral grid. Each circular point in the figure represents a piece of data; the two large circles in the top-left and bottom-right corners represent data nodes for the north and south poles. Each of the blue rectangular regions represents a collection of data that stored in an array. We illustrate six by six nodes in each non-polar array but in practice these arrays contain thousands of nodes. The lines between nodes show connectivity. The lines going into and out of the red bars connect to each other.

Heikes at Colorado State University's Atmospheric Science Department [5]. We discuss proxy applications in greater depth in Section 2.4. The example in Figure 1.3 executes a stencil operation using a single program, multiple-data (SPMD) style of computation. In this style, each processor is responsible for updating a set of locally owned values. SWM stores these locally owned values in blocks. Specifically, SWM partitions the icosahedral grid into a finite number of blocks, which are then distributed across processors. In lines 5 and 6 of Figure 1.3, the example iterates over a processor's locally owned blocks, and in lines 7 and 8 it iterates over the nodes contained in a block. In lines 9 through 14 SWM performs the stencil operation. The stencil operation stores its result in the `x2` array and reads values from the `x1` and `weight` arrays.

When SWM allocates the memory for the `x1`, `x2`, and `weight` arrays, it adds an extra layer of elements around each block called a halo. The halo stores copies of values from

```
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 1.3: 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 1.4: Fortran source code for a simple stencil. The function x2 calculates the stencil for position (i,j) reading values from x1.

neighboring blocks, and ensures that the same stencil operation legally applies across all non-polar nodes. The poles have their own specialized code in lines 19 through 27. Note that details about parallelism (such as SWM's use of blocks and halos) and grid connectivity (such as the fact that SWM splits the grid into ten subgrids) appear in the loop-nest-header lines (5-8).

Despite the tangling shown in Figure 1.3, the software architects of GCRM have taken steps to keep the amount of special code needed to a minimum [6]. For example, to handle nodes that have five neighbors instead of the typical six, GCRM fills halo values for missing sixth neighbors with a zero. In this manner, GCRM is able to apply the same six-neighbor stencil throughout the whole subdomain and only requires specialized code for the poles.

Furthermore, by using halos, GCRM's architects have cleanly separated communication and stencil code. Nevertheless, the communication code that is in GCRM is highly specialized for icosahedral-grid connectivity. This implies that if GCRM's developers wanted to change the underlying grid used, they would have to rewrite this communication code.

In the 14,822 line-of-code Shallow-Water Model (SWM), initializing and conducting communication involves 1,010 lines of code (6.8%). In the 3,214 line-of-code CGPOP application [135], which is a proxy application for the Parallel Ocean Program (POP), initializing and conducting communication involves 1,226 lines of code[1] (38%). CGPOP is one of the contributions of this dissertation and was initially presented in [135] and [137]. We discuss CGPOP depth in Section 2.3.

If scientists wanted to change the underlying grid used in either of these proxy applications (or the applications they model), they would have to rewrite the application's communication code. One reason why scientists may want to change grids for scientific modeling reasons; for example, POP's developers rewrote its communication code between versions 1.4.3 and 2.0

[1]We measured these lines-of-code values using the SLOCCount tool [23].

to use the tripole- rather than dipole-grid. Compared to dipole grids, tripole grids have a more uniform cell size across regions representing Earth's oceans [97, 129].

Ideally, programmers should be able to modify the grid an application uses without rewriting and specializing communication code or having grid details tangle with stencil code. In Figure 1.3 we show an example stencil from SWM that includes specialized code for the poles; in Figure 1.4 we show an idealized version of the stencil that does not include any specialized code.

Several libraries, languages, and tools help programmers to separately specify stencil algorithm from implementation concerns [4, 55, 56, 69, 80, 81, 96, 108, 112, 150]; however, related work has not focused on semiregular grids (such as those seen in POP and GCRM). Additionally, some of these approaches require rewriting full simulation applications in new languages. In this dissertation we introduce a programming model, library, and source-to-source code transformation tool for implementing stencil computations on semiregular grids.

1.2 Approach

To address the problems discussed in Section 1.1, we introduce a new programming model for stencil computations. We call this programming model GridWeaver, and it contains abstractions that programmers can use to orthogonally specify semiregular-grid connectivity, stencil computation, and data decomposition. We implement these abstractions in a Fortran library and use MPI for parallelization. From a user's perspective, GridWeaver parallelizes code as specified with a decomposition abstraction and automatically conducts communication.

GridWeaver takes an *active library* approach [61] by including a source-to-source translation tool that inlines GridWeaver function calls. Library-based approaches have the advantage of integrating easily with existing Fortran programs, but the disadvantage of introducing library overhead. With an active library approach we remove this potential disadvantage.

1.3 Research contributions

In this dissertation we describe the GridWeaver programming model and introduce algorithms that are necessary for GridWeaver to perform efficiently and automatically parallelize code. We also describe how to use proxy applications to evaluate and compare programming-models, and introduce CGPOP: a proxy application of the parallel ocean program. In chapter 8 We use CGPOP and SWM to evaluate GridWeaver. The specific contributions we make in this dissertation are as follows:

- **GridWeaver programming model** — a series of abstractions to describe semiregular grid connectivity, specify stencil algorithms, and address parallelization concerns. (see Chapters 5 and 6). Specifically, we introduce the following abstractions for semiregular grids: subgrids, grids, distributions, stencil functions, and data-objects.

- **Communication plan generation algorithms** — algorithms that automatically determine how to communicate data between processes to populate halos (see Sections 5.2.2 and Section 6.4). A communication plan is a collection of metadata describing what messages to send and receive between processes in a distributed memory system. Communication is necessary in stencil computations so that when a process updates its local data it has access to neighboring remote values. Automatically generating communication plans enables GridWeaver to perform communication without the user having to writing any communication code themselves.

- **GridWeaver active library** — a library containing functions to specify and conduct semiregular-grid computations and a source-to-source translation tool that replaces function calls with more efficient code (see Chapters 5, 6 9, and Appendix B).

- **Communication avoiding optimization for semiregular grids** — we describe how to apply a communication avoiding optimization to semiregular grids (see Section 7.2).

- **Terminology for describing and comparing programming models** — we introduce a set of terms for classifying how features in programming-models separate implementation details from algorithms (Section 3.3). We use these terms to describe and compare programming models for stencil computations in Chapter 4.

- **CGPOP** — a proxy application for the Parallel Ocean Program (Section 2.3). We developed this proxy application in collaboration with John Dennis at the National Center for Atmospheric Research (NCAR). We describe what proxy applications are in Section 2.2, and we use CGPOP to evaluate GridWeaver in Chapter 8.

- **CGPOP and SWM case studies** — evaluation of GridWeaver applied to the CGPOP and SWM proxy applications (see Chapter 8). We also demonstrate how to change CGPOP to use the tripole rather than dipole grid. The developers of the Parallel Ocean Program manually performed this change between versions 1.4.3 and 2.0.

1.4 Dissertation organization

To describe and defend our contributions, in the remaining chapters of this dissertation, we provide background material, describe the approach we take to separate implementation concerns for semiregular-grid stencil-computations, and evaluate our approach as implemented in the GridWeaver active library. Specifically, we organize this dissertation as follows:

- **Chapter 2: Earth-science grids and proxy applications** — this chapter gives background material on grids used in Earth science applications, discusses a methodology for using proxy applications to evaluate programmability, introduces the CGPOP proxy application, and gives an overview of the SWM proxy application. We use CGPOP and SWM to evaluate GridWeaver in Chapter 8. Chapter 2 includes work that we presented in [135], [137], and [140].

- **Chapter 3: Background on programming models** — in this chapter we introduce terminology to describe how, and qualitatively evaluate to what extent, programming

models separate algorithms from implementation concerns. We use this terminology in Chapter 4 to discuss related work. Chapter 3 includes work presented in [103].

- **Chapter 4: Programming models for stencil computations** — using the terminology from Chapter 4 we discuss and compare related programming models, languages, and tools. Chapter 4 also discusses the motivation behind these tools and the tradeoffs involved when deciding whether to store grid data in array-based versus graph-based data structures.

- **Chapter 5: Semiregular-grid abstractions** — this chapter introduces abstractions that programmers can use to describe stencils computations and semiregular-grid connectivity and decomposition. The chapter also describes how we implement these abstractions into GridWeaver and introduces a communication plan generation algorithm. The communication plan generation algorithm automatically determines what communication is necessary to populate halos on distributed-memory clusters. The algorithm we introduce in this chapter only produces correct results for compact stencils, which are stencil computations that update each cell only using values from immediate neighbors. This chapter includes work presented in [136].

- **Chapter 6: Addressing non-compact stencils** — this chapter generalizes the abstractions and communication algorithm from Chapter 5 to work with non-compact stencils. Non-compact stencils access values that are not immediate neighbors. SWM uses a non-compact stencil in its horizontal-advection operation. Non-compact stencils are also required to conduct a communication avoiding optimization that we discuss in Chapter 7.

- **Chapter 7: Performance optimizations** — this chapter discusses optimizations that the GridWeaver source-to-source translation tool can automatically perform to improve application performance. These optimizations include inlining library calls and communication avoiding with overlapping tiles.

- **Chapter 8: CGPOP and SWM case studies** — in this chapter we evaluate GridWeaver with case studies using the CGPOP and SWM proxy applications. For CGPOP we describe how to transition from CGPOP's native dipole grid to the tripole grid. For SWM we describe how to model an icosahedral grid and how to model SWM's non-compact horizontal-advection stencil.

- **Chapter 9: The GridWeaver active library** — in this chapter we document GridWeaver and its software architecture, compilation process, and invocation process. This chapter also includes a tutorial on using GridWeaver to construct and compute Conway's Game-of-Life [78] on an icosahedral grid.

- **Chapter 10: Current limitations and future work** — this chapter discusses current limitations in our approach. We also propose approaches that future work could take to overcome these limitations. Possible future work includes integrating GridWeaver to work with additional types of topologies and hardware architectures, adding a visualization tool, and automatically extracting semiregular-grid connectivity from existing irregular grids.

- **Chapter 11: Conclusions** — we end this dissertation with concluding remarks.

Chapter 2

Earth-science grids and proxy applications

We focus this dissertation on stencil computations applied to semiregular Earth grids. In existing Earth science codes, grid connectivity and parallelization details tangle with algorithm code; GridWeaver aims to resolve this by introducing new programming abstractions that separate these concerns.

In this chapter we: give background material about semiregular Earth grids (Section 2.1), discuss how to use proxy applications to evaluate programmability (Section 2.2), introduce CGPOP: a proxy application of the Parallel Ocean Program (POP) (Section 2.3), and provide an overview of SWM: a proxy application of the shallow water model component of the Global Cloud Resolution Model (GCRM) (Section 2.4). In Chapter 8 we evaluate GridWeaver when using it to rewrite the stencil computations in CGPOP and SWM.

2.1 Earth science grids

Scientists and researchers have developed different types of grids for various types of Earth simulations [97, 129, 121]. Most of these grids are discretizations of a spherical shell. The shell can consist of multiple layers with indices around the shell forming its horizontal axis, and indices through each level of the shell forming its vertical axis.

Within a given level of elevation are a finite number of horizontal indices. Indices may be in Cartesian, polar, or curvilinear lat/lon coordinates. Since lat/lon is a common format for storing Meteorological data, one common type of grid is the dipole lat/lon grid.

In the dipole lat/lon grid, cells represent a partition of the Earth's surface referenced by evenly spaced degrees of latitude and longitude. Since the length of a degree of longitude in

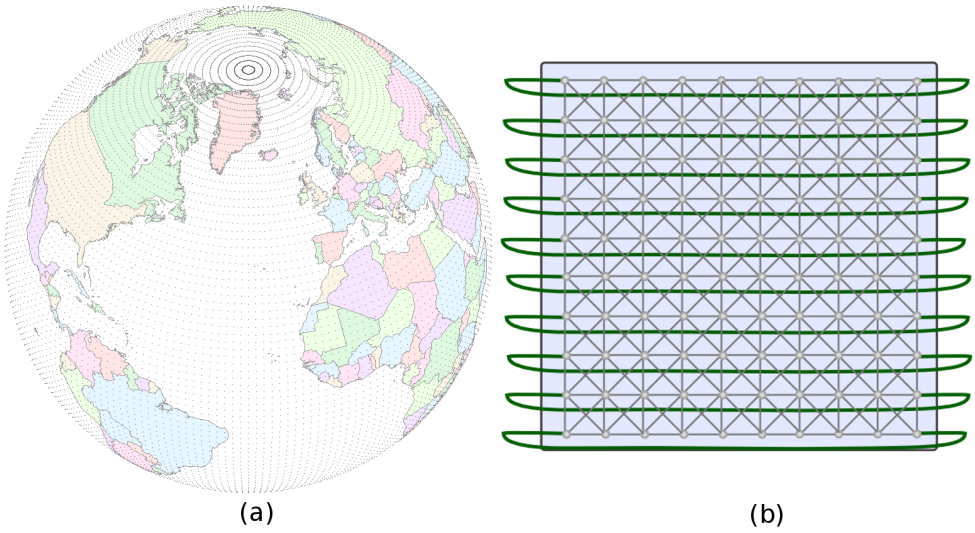


Figure 2.1: (a) Lat/lon grid (picture from [14]). Notice that even though there is an equal amount of space between each parallel (line of latitude), and that within a parallel each point is equally spaced, the amount of space between points varies between parallels. (b) eight-point dipole connectivity of the lat/lon grid.

kilometers is dependent on its distance from the equator, cell sizes in a lat/lon grid vary. We illustrate a lat/lon grid and its dipole connectivity in Figure 2.1.

To address issues of non-uniform cell size, scientists and researchers have developed the tripole [129], cubed sphere [121], and icosahedral grids [123, 24, 86]. We illustrate these three grid types and their connectivities in Figure 2.2.

One commonality between the three types of grid illustrated in Figure 2.2 is that topologically they consist of a finite number of regular regions whose borders connect to each other in an irregular fashion. We call grids that exhibit this behavior: *s*emiregular. When grids are semiregular, arrays can store their regular regions; however, complex communication code is necessary to update and coordinate values along the borders of the regular subgrids.

Storing values in arrays instead of graphs improves application performance because stencil computations can access neighboring values directly rather than indirectly through an adjacency list. We discuss this performance impact in greater detail in Section 4.2. The GridWeaver tool aims to maintain the performance benefits of storing data in arrays while removing the need for programmers to write complicated communication code.

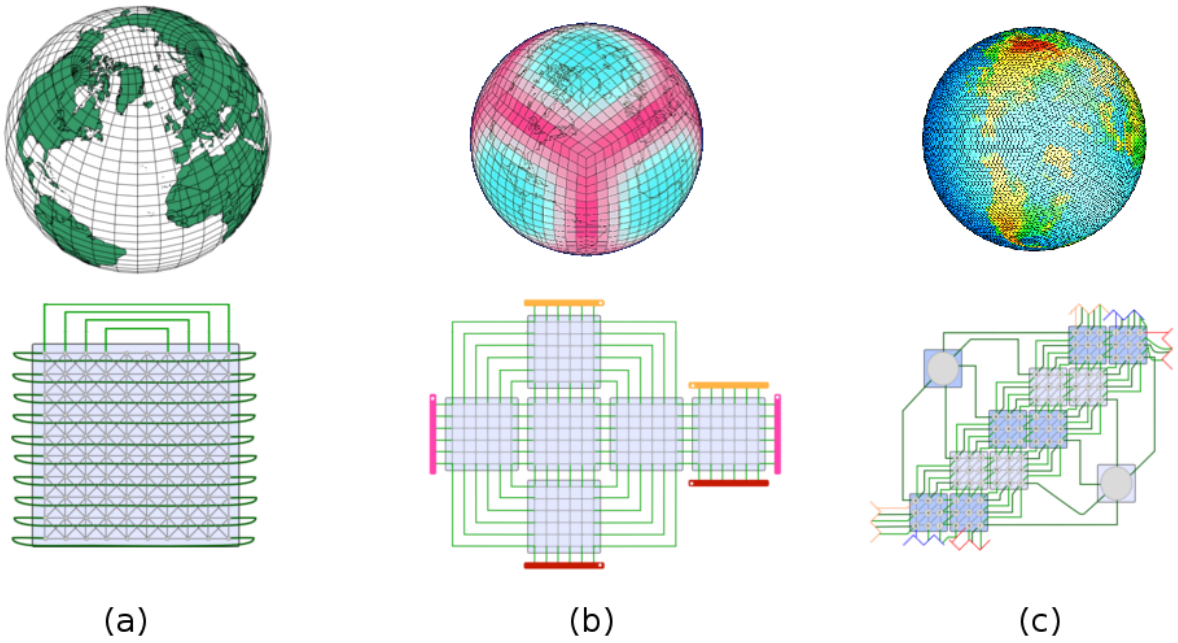


Figure 2.2: Different types of Earth science grids: (a) tripole, (b) cubed sphere, and (c) geodesic/icosahedral. The Picture of tripole grid is from [116], picture of cubed sphere grid is from [11], and the picture of the icosahedral grid is from [25].

2.2 Proxy applications

GridWeaver is a domain-specific programming model for semiregular-grid stencil-computations. Like more general programming models that target parallel codes, GridWeaver aims to improve programmability; however, accurately and objectively evaluating programmability is difficult [145, 110, 142, 111].

One problem with evaluating programmability is finding adequate benchmarks. Ideally, researchers could evaluate programming models by reimplementing large real-world applications and demonstrating a programmability improvement. Pragmatically, this is not a feasible approach since rewriting large scientific applications may involve rewriting hundreds-of-thousands of lines-of-code and require domain-specific knowledge.

In [87], Heroux et al. suggests using miniapps (a type of proxy application) to compare programming models in terms of programmability as well as performance. Proxy applications are applications on the order of a few thousand lines of code that include a simple build

system and model a larger application. In this section we refer to these larger applications as the *modeled applications*.

The concept of a proxy application being a *performance proxy* is common and well understood. To be a performance proxy, an application should accurately model the performance bottleneck of the full application at the range of cores that the full application typically targets. Researchers can use a performance proxy to evaluate what impact a new compiler, language, or programming model would have on the performance of the proxy's modeled application.

Heroux's work by proposing specific criteria that researchers can use to qualitatively evaluate and describe how well an application serves as a *programmability proxy* [135, 137]. Researchers can use a programmability proxy to evaluate what impacts a new compiler, language, or programming model would have on the programmability of the proxy's modeled application. Specifically, we argue that to be a programmability proxy an application should: (1) include an important algorithm, (2) have a design that enables changes made in the proxy application to be incrementally made in the modeled application, and (3) include representative implementation details of the modeled application. The true measure of how well an application acts as a programmability proxy is whether application developers are willing to use results from the proxy to evaluate possible changes in the full application. We believe that this is more likely to be the case when a programmability proxy satisfies these three requirements, which we term: *code importance*, *incrementality*, and *representativeness*.

To satisfy the *code importance* requirement, a proxy application must contain code that is both central to the full application's result and is common enough that studying it will yield broadly applicable results. Other researchers have worked to identify and classify algorithms that are important, such as the Motif's in the Berkeley view [35] and the algorithm-layer patterns in the OPL Pattern Language [98]).

To satisfy the *incrementality* requirement, a proxy application must define a clean interface between the full application and itself. This is to say — there exists some set of components (i.e., classes, functions, data structures) common to both the proxy application and the

modeled application; in the modeled application, these components should not be tightly coupled with any component outside the set: any interaction to components outside the set should occur through the interface. This ensures that changes made in the proxy application are possible to make in the full application, and that the application's developers could make these changes in the same way and without requiring large rewrites of code outside of what the proxy application models. This ensures that evaluations made on the proxy application are applicable to the modeled application. One way of evaluating incrementality is to reintegrate an unchanged proxy application back into its modeled application and measure the number of modified lines-of-code. If this number is large than it is unlikely that an applications' developers could incrementally change their modeled application in the same way that researchers changed the proxy application.

To satisfy the *representativeness* requirement, a proxy application should contain implementation details (i.e. parallelization strategy, data distribution, communication schedule, optimizations, etc. [103]) that affect the modeled application. In the absence of satisfying this requirement, researchers risk making conclusions that are applicable to the simplified proxy but not applicable to the modeled application. Satisfying this requirement involves a tradeoff between including representative details and providing a small, modular miniapp; thus we recommend that proxy-application developers consult with model-application developers to determine what implementation details the model-application developers consider the most important.

Proxy application developers should also consider that their choices regarding what implementation details to include can impact whether the proxy is able to provide a fair platform for comparing programming models. For instance, a proxy application that requires a two-sided communication programming model would not make a fair benchmark to evaluate programming models based on one-sided communication.

2.3 The CGPOP miniapp

In Chapter 8 we use the CGPOP and SWM proxy applications to evaluate GridWeaver. The creation of CGPOP is one of the contributions of this dissertation; We developed CGPOP in collaboration with John Dennis at the National Center for Atmospheric Research (NCAR). In this section we describe the CGPOP miniapp.

Specifically, CGPOP is a proxy application for version 1.4.3 of the Parallel Ocean Program (POP). POP is an important multi-agency code primarily developed at Los Alamos National Laboratory [94]. POP is a global ocean modeling program and a component of the Community Earth System Model (CESM) [10]. CGPOP encapsulates the performance bottleneck of POP, which is its conjugate gradient solver. The CGPOP miniapp contains about 3000 source lines of Fortran90 and MPI code, whereas POP contains about 71,000 lines of code. Along with John Denis, our collaborator at NCAR, we have used CGPOP to evaluate using different subdomain data structures, computation/communication overlap approaches, and programming models such as Coarray Fortran and one-sided MPI in the Parallel Ocean Program [137].

2.3.1 Extracting POP's main kernel

Our first step when constructing CGPOP was to identify the most performance impacting portion of code in POP. To understand the performance characteristics of POP, we profiled the scalability of three main parts of the application: the barotropic component, the baroclinic solver, and the 3D-update step. The barotropic component updates two-dimensional surface-pressure data and uses a single-production version of a preconditioned-conjugate-gradient algorithm. The POP solver is very sensitive to network latency and OS noise at large core counts [76]. The baroclinic component solves the equations of motion for the ocean model and involves *pleasantly parallel* calculations that stream data through the memory hierarchy to the CPU. The 3D-update step updates three-dimensional prognostic variables such as temperature and velocity at the end of each timestep.

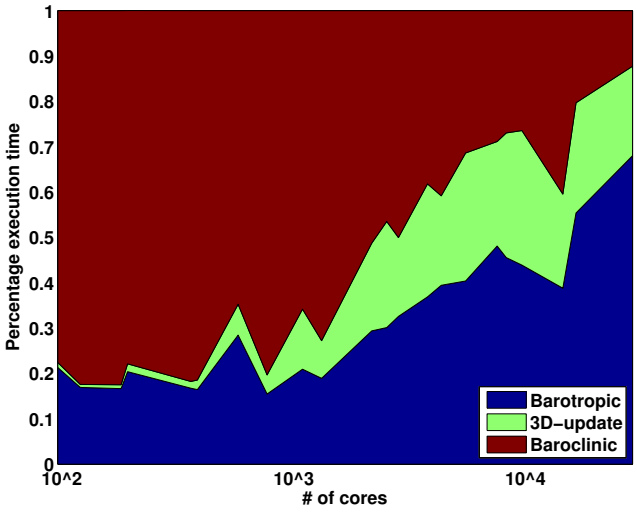


Figure 2.3: Relative cost of the different components of POP when executing a 0.1° dataset on Kraken: a Cray XT5 [18].

In Figure 2.3 we provide a breakdown of the relative computational cost of POP's three main components. To gather the data in this figure we executed POP with a 0.1° dataset on Kraken: a 99,072 core Cray XT5 supercomputer located at the National Institute for Computational Sciences [18]. The 0.1° dataset has an average of 0.1° of latitude and longitude separating neighboring grid points on Earth's equator.

While the relative cost of the baroclinic component is dominant at fewer than 1000 cores, the relative cost of the communication intensive sections of the code (barotropic and the 3D-update step) dominate at greater than 1000 cores. Thus we decided to have CGPOP contain the conjugate gradient solver from POP's barotropic component.

2.3.2 Required behavior of CGPOP

Broadly, we define the CGPOP proxy application in terms of its input/output behavior and the algorithm it conducts. We illustrate this behavior in Figure 2.5 and list pseudocode for the CG algorithm in Figure 2.4. The CGPOP application takes an intermediate state files produced by the `cginit` domain-decomposition generator.

The `cginit` domain-decomposition generator takes a single input file, that contains: (1) a dipole grid (stored as a 3600×2400 two-dimensional array), (2) the expected output of

```
x = function CGPOP-solver(A, x_0, b, M^{-1})
!----------------------------------
! Compute initial residual
!----------------------------------
s = A x_o
r = b - s
rr0 = (GlobalSum(r * r))^{1/2}

UpdateHalo(r)

!----------------------------------
! Single pass of CG algorithm
!----------------------------------
z = M^{-1} r
s = z
q = A s

UpdateHalo(q)

{ρ, σ} = GlobalSum({r * z, s * q})

!----------------------------------
! Calculate coefficient
!----------------------------------
α = ρ/σ

!----------------------------------
! Compute solution and residual
!----------------------------------
x = x + α s
r = r - α q

do for 124 iterations:
  !----------------------------------
  ! Apply preconditioner
  !----------------------------------
  z = M^{-1} r
  az = A z

  UpdateHalo(az)

  {ρ', δ, γ} =
     GlobalSum({r * z, r * r, az * z})

  !----------------------------------
  ! Calculate updated coefficients
  !----------------------------------
  β = ρ'/ρ
  σ = δ - β^2 σ
  α = ρ/σ
  ρ = ρ'

  !----------------------------------
  ! Compute solution and residual
  !----------------------------------
  x = x + α(z + β s)
  r = r - α(az + β q)
  s = z + β s
  q = az + β s
```

Figure 2.4: CGPOP's preconditioned conjugate gradient algorithm.

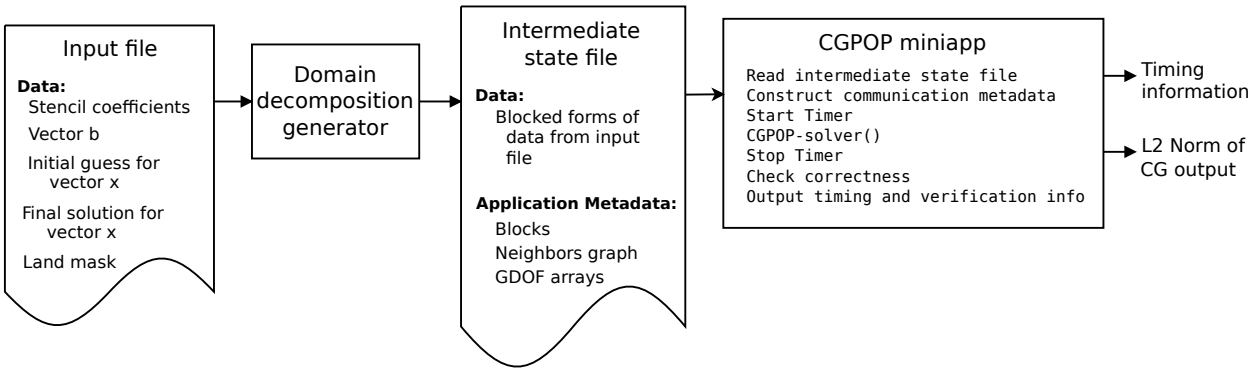


Figure 2.5: Software architecture of the CGPOP Miniapp

the POP conjugate gradient computation, (3) stencil coefficients, (4) a mask to indicate if a grid point is ocean or land, (5) the initial guess, and (6) solutions for the x and b vectors in Figure 2.4. The data stored in the input file is in NetCDF format [126, 20]. The CGPOP website [8] includes a link to download this input file.

The domain-decomposition generator breaks the 3600×2400 global domain into subdomain blocks and outputs these blocks into an intermediate state file (also known as a tile file). Users can configure the block sizes the generator outputs. By default the generator outputs block sizes of 180x120, 120x80, 90x60, 60x40, 48x32, 36x24, 24x16, and 18x12. Different block sizes are better suited for different numbers of processors. Generally, smaller block-sizes work better with more processors.

In addition to the data component of the intermediate state file, there is metadata that describe the relationship between blocks. There is a set of block information records, a graph of neighbors, and integer arrays that correspond to the global degree of freedom (GDOF) for every point in each block. GDOF values are identifiers that are unique for each grid point in the global domain. The block information records identify the location of each rectangular block within the global domain in terms of two-dimensional indices in the global domain.

CGPOP uses data stored in intermediate state files as input for its conjugate gradient algorithm; it outputs information about timing and correctness. CGPOP's correctness test calculates the difference between an L2 norm for x as calculated by CGPOP against a precomputed answer, stored in the input file, calculated by POP.

We list pseudocode for CPOP's conjugate gradient algorithm in Figure 2.4. This algorithm includes a single inner product [66] that iteratively solves the vector x in the equation $Ax = b$. CGPOP reads the matrix A, the right-hand-side vector b, a diagonal pre-conditioner vector M^{-1}, and a guess vector x_0 from an intermediate state file and passes these values to the `CGPOP-solver` function. The final surface pressure vector x is the output of `CGPOP-solver` function.

The `CGPOP-solver` function performs several linear algebra computations, two communication steps before conducting the iterative CG algorithm, and one communication

step for each timestep within CG. The `GlobalSum` function performs a three-word vector reduction, while the `UpdateHalo` function performs a boundary exchange between neighboring subdomains. The `UpdateHalo` function takes an array distributed across processes as specified by the blocks data structure in the intermediate state file. The asterisk $*$ indicates a dot product between two vectors involving all entries in the local subdomain, and the `GlobalSum` function completes the full dot product.

2.3.3 Establishing CGPOP as a performance proxy

One of the characterizations of a miniapp proposed by Heroux et al. [87] is that a miniapp should be on the order of a thousand lines-of-code and include a simple build process to enable easy porting. These size and simplicity requirements are critical because POP's developers' wanted a miniapp to enable benchmarking in immature environments. CGPOP satisfies these requirements in that the miniapp is approximately 3000 lines-of-code and we have shown that the build process correctly works on a number of platforms (see Table 2.1). Additionally, to be of use to scientific application developers, a miniapp should serve as a performance proxy for the target full application. Hereoux also argues that to be a *performance proxy*, a miniapp should accurately model the performance bottleneck of a full application at the range of cores that the full application typically targets. In this subsection, we show that CGPOP is a performance proxy for POP by showing that the scalability of CGPOP's base MPI implementation is similar to that of POP.

How scalable an application is depends on the machine and compiler used. To ensure that the performance behavior of CGPOP matches POP we examined scalability across three different platforms: Hopper, a Cray XE6 located at the National Energy Research Supercomputing Center (NERSC); Frost, a BlueGene/L located at the National Center for Atmospheric Research (NCAR); Lynx, a Cray XT5 also located at NCAR; and Kraken, a Cray XT5 located at the National Institute for Computational Science (NICS). We list technical information about these platforms in Table 2.1. The compilers we used in our examination were PGI Fortran, Cray Fortran, and XL Fortran. We present our results in

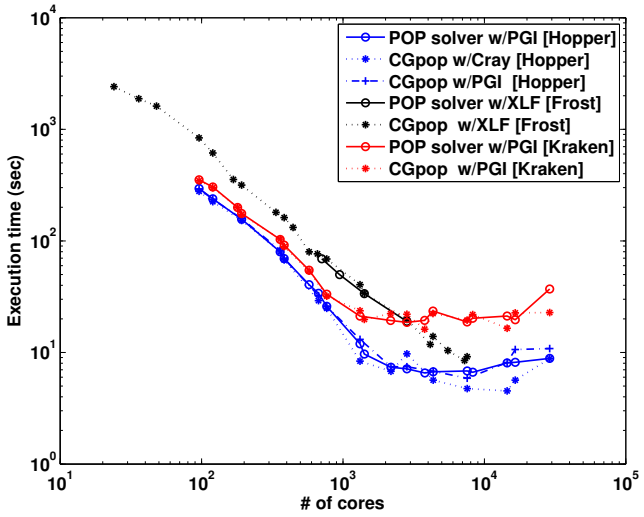


Figure 2.6: Execution time in seconds for 1 day of the barotropic component of the POP 0.1° benchmark and the MPI-2S version of the CGPOP miniapp on three different compute platforms

Figure 2.6; note that the similarly colored lines for POP and CGPOP are close to one another, which indicates that the POP and CGPOP's scalability behavior matches when the machine and compiler used matches.

For Kraken, the execution time of the barotropic section of POP and CGPOP matches well for all core counts. Note that considerable variability in execution time on Kraken for configurations with greater than 1,000 cores. This is an expected result of running an OS noise sensitive application on a shared production supercomputer. Additionally the loss of scalability in the CGPOP miniapp at large core counts on Kraken is a reflection of number of factors including the cost of global reduction within the solver. At greater than 4,000 cores on Kraken, approximately 50% of the execution time is due to a three-word global reduction.

On Frost the execution time for CGPOP and POP are also in close agreement. Not surprisingly the execution time of the CGPOP continues to scale on Frost, which is a Blue Gene/L system and therefore provides hardware support for global reductions.

On Hopper, execution times for the POP barotropic solver and CGPOP agree (within system variability) whether using the Portland Group (PGI) or Cray Compiler. Interestingly,

Table 2.1: Description of compute platforms used to evaluate CGPOP

System				
Name	Kraken [18]	Hopper [13]	Lynx [17]	Frost [19]
Company	Cray	Cray	Cray	IBM
System Type	XT5	XE6	XT5	BG/L
# of cores	99,072	153,408	912	8192
Processor				
CPU	Opteron	Opteron	Opteron	PPC440
Mhz	2600	2100	2200	700
Peak Gflops/core	10.4	8.4	8.8	2.8
cores/node	12	24	12	2
Memory Hierarchy				
L1 data-cache	64 KB	64 KB	64 KB	32 KB
L2 cache	512 KB	512 KB	512 KB	2 KB
L3 cache	6 MB (shared)	12 MB (shared)	6 MB (shared)	4 MB (shared)
Network				
topology	3D torus	3D torus	2D torus	3D torus
# of Links/per node	6	6	4	6
Bandwidth/link	9.6 GB/s	26.6 GB/s	9.6 GB/s	0.18 GB/s

the execution time for CGPOP using the Cray compiler appears to be depending on setting the following environment variable:

```
export HUGETLB_DEFAULT_PAGE_SIZE=512K
```

This environment variable enables the use of 512 KByte pages. By default the Cray compiler uses 2 MByte large pages while the PGI compiler uses the much small 4 KByte pages. Preliminary performance timings on Hopper indicated that use of the Cray compiler had a significant negative impact on scalability at large core counts. Subsequent tests with 512 KByte pages revealed significantly smaller execution time for CGPOP. While this suggests that reducing the page size positively impacts performance by a factor of 5, the performance timings for the two different page sizes were not measured at the same time. Further because our access has been during the pre-acceptance phase of Hopper, which potentially involves modifications to the system OS, other system issues may be at play. We intend to closely work with Cray and the technical staff at NERSC to resolve these issues. Regardless of the outcome, this experience clearly demonstrates the utility of the CGPOP miniapp at helping to identify potential scalability issues within compilers.

2.3.4 Establishing CGPOP as a programmability proxy

In this section we argue that CGPOP [135] is a programmability proxy for POP [94]. We do this by examining how CGPOP addresses the criteria listed in Section 2.2.

First, CGPOP includes *important code*. The conjugate gradient solver within POP is the performance bottleneck when executing the application on more than a thousand of cores, and Conjugate Gradient is a broadly used algorithm for solving systems of linear equations [88, 33, 37, 132].

Secondly, CGPOP fulfills the *incrementality* requirement. We consider a miniapp to be incremental if it is possible to incorporate a different version of the miniapp back into the full application. We consider incrementality threatened when reintegrating an implementation of the CGPOP miniapp back into the full POP application requires editing code outside of the Conjugate Gradient solver routine. We do not believe this to be a concern for CGPOP miniapp due to the fact that the difference between the base and studied CGPOP implementations predominantly lie in the construction of communication metadata and a subroutine (`UpdateHalo`) that performs the communication. Because communication metadata is solely used by the `UpdateHalo` routine and the fact that this routine is exclusively used by the Conjugate Gradient algorithm, changes to this metadata or the `UpdateHalo` routine does not invasively impact any code outside of them when reintegrated into POP.

Finally, CGPOP includes *representative* implementation details. For the POP application, the most important implementation details are the discretization of the Earth's surface including the exclusion of land, the domain decomposition for parallelization, the interaction between communication and computation, and the metadata generated to represent the communication schedule.

The discretization and the domain decomposition appear throughout the POP application and as such these important implementation details are part of the input for the CGPOP miniapp. Code within CGPOP uses the same domain decomposition as in the full POP application.

The interaction between communication and computation and the communication meta-data are representative implementation details in POP, which CGPOP encapsulates. Thus CGPOP isolates programmability impacts for implementation details. This makes it easier to reintegrate such changes back into the full application while making CGPOP a more representative programmability proxy.

2.4 The Shallow Water Model code

The Global Cloud Resolution Model (GCRM) [5] is an application developed by David Randall's group at Colorado State University's Atmospheric Science Department. It is part of the Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program. GCRM helps scientists study how particles form and influence clouds. GCRM models the Earth with over one-hundred million hexagonal cells containing data about wind, temperature, and humidity. SWM is a shallow-water model simulation proxy application extracted from GCRM. We rewrite SWM's horizontal advection operation in our evaluation of GridWeaver in Chapter 8. We conduct this case study because SWM includes a non-compact stencil seen in a real-world application and because SWM applies its stencil on a grid with a complex connectivity pattern (an icosahedral grid).

SWM's horizontal advection operation applies two stencils per time step. One of these stencils is compact (references values only from neighboring cells) and one is non-compact (accesses values within some bounded distance). In the compact stencil SWM accesses values directly in an array, while in its non-compact stencil SWM accesses data through a graph-based representation of the grid.

In each timestep SWM uses 1,891 lines of communication code to update halos in an icosahedral grid and copy values between array based and graph-based data structures. With GridWeaver we are able to avoid having to write any communication code and are able to perform the non-compact stencil without converting values into a graph based representation (see Chapter 6 and the SWM case study in Section 8.2).

In both its graph-based and array-based representations, SWM has each cell store values representing the mass of particles in a given area, a wind normal for three of its edges, and a flux value for three of its edges. To store values for edges and corners, SWM has each cells assume ownership of three corners and three edges.

Each time SWM's horizontal advection operation is called, it performs a non-compact stencil to update flux values on cell edges then performs a compact stencil to update the value within a cell. SWM's non-compact stencil is bounded to access values that are no further than three cells away. In the rest of this section we discuss we discuss SWM's non-compact stencil. We list pseudocode for the operation in Figure 2.7. Lines 13 through 29 perform the update of flux values while lines 31 through 39 use values from neighboring cells and the flux values on edges to update cell values.

To calculate flux, SWM uses interpolated values computed in lines 20 through 23. We illustrate an example interpolated value in Figure 2.9. SWM computes the location of each point p_{up} as a three-dimensional coordinate on the unit sphere. This computation calculates a location projected on the unit sphere from a tangent line running from the center of cell p_0 through the midpoint of a specified edge.

To interpolate the value for p_{up} SWM pre-computes weight values reflecting the distance of an arch between p_{up} and the midpoints of cells containing p_a, p_b, and p_c. In Figure 2.7 these values are stored in the `weight` array, and the indices of cells p_a, p_b, and p_c are stored in the `indx` array.

In computing `indx` SWM makes use of function that can determine the cell that owns a location on the unit-sphere, and a function `indx`, which takes two adjacent cells X and Y and a twist amount n and returns the n-th cell visited when turning clockwise n times from Y around X. In Figure 2.8 we illustrate the function `twist` for hexagonal and pentagonal cells. We implement this `twist` function in GridWeaver.

```
routine horizontal_advection:
  INPUT:
    x(i,j), x_1d(idx)  ! Mass for a cell (array- and graph- representations)
    d_edge(e,i,j)      ! Weight associated with each edge
    wnd(e,i,j),        ! Wind pressure on each edge
    weight(x,p,e,i,j), indx(x,p,e,i,j)  ! precomputed weight and index values
                                        ! for interpolation points

  INPUT/OUTPUT:
    flx(e,i,j),  ! Flux value for each edge of each cell
    x_f(i,j),    ! Resulting mass for a cell (array representation)

!-------------------------------------------------
! Update flux (a non-compact stencil)
!-------------------------------------------------
for each point <i,j> in all subdomains:
  ! Calculate interpolation values
  for edge e from 1 to 3:
    for point p from 1 to 4:
      intrp(p, e) =
        weight(1, p, e, i, j) * x_1D(indx(1, p, e, i, j)) +
        weight(2, p, e, i, j) * x_1D(indx(2, p, e, i, j)) +
        weight(3, p, e, i, j) * x_1D(indx(3, p, e, i, j)

  ! Integrate (this portion is compact)
  for edge e from 1 to 3:
      ! ... code that uses intrp to calculate a value tmp ...

   flx(e, i, j) = wnd(e, i, j) * tmp

!-------------------------------------------------
! Update cell value (a compact stencil)
!-------------------------------------------------
for each point i,j in all subdomains:
  x_f(i,j) = area_inv(i,j) *
    ((flx(1, i,   j)   * d_edge(1, i, j)  + flx(2, i,   j)   * d_edge(2, i, j)  +
      flx(3, i,   j)   * d_edge(3, i, j)) -
     (flx(1, i-1, j)   * d_edge(4, i, j)  + flx(2, i-1, j-1) * d_edge(5, i, j)  +
      flx(3, i  , j-1) * d_edge(6, i, j)))
```

Figure 2.7: Psuedocode of SWM's horizontal advection operation.

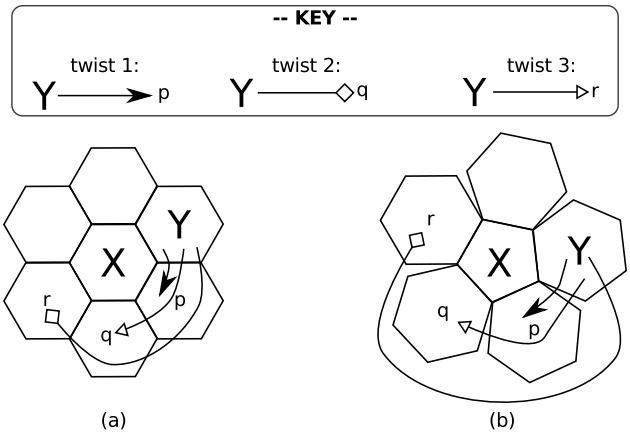


Figure 2.8: The twist function takes two adjacent cells X and Y and a twist amount n. (a) illustrates twists of 1, 2, and 3 for a hexagonal cell, (b) illustrates twists of 1, 2, and 3 for a pentagonal cell.

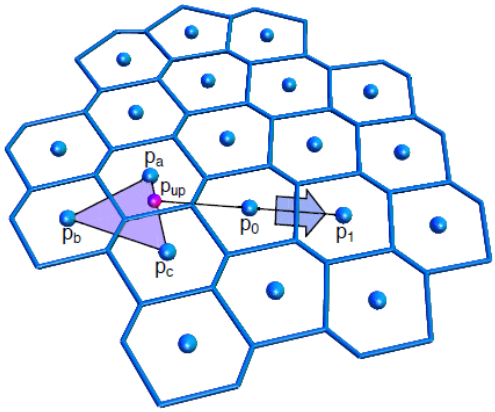


Figure 2.9: Points used to update the flux along a given edge. SWM calculates the value for p_{up} by interpolation using the values of three surrounding cells p_a, p_b, and p_c. Picture from [85].

2.5 Summary

In Chapter 8 specifically, we evaluate whether the GridWeaver abstractions are expressive enough to specify computations in the CGPOP and SWM proxy applications and whether using GridWeaver improves programmability while maintaining performance. As we describe in Section 2.2 proxy applications are smaller programs that model the behavior and implementation details of larger applications. We argue that researchers developing programming

models (such as GridWeaver) can use proxy applications to evaluate the performance and programmability impacts their programming models would have on real-world codes without having to reimplement real-world applications in their entirety.

We choose to evaluate with both CGPOP and SWM because they exhibit behavior seen in real-world Earth-science simulation codes and because they differ in terms of the grids they use, the stencils they use, and the data decompositions they use. CGPOP performs a compact stencil on a Dipole grid; and SWM performs compact and non-compact stencils on an icosahedral grid.

Chapter 3

Background on programming models

The approach we present in this dissertation to separate implementation concerns is to develop a new programming model for semiregular-grid stencil-computations. In this chapter we give background material about programming models by: (1) broadly discussing what programming models are (Section 3.1), (2) listing mechanisms used to realize programming models (Section 3.2), and (3) introducing terminology to describe and compare features in programming models that separate implementation concerns from algorithms (Section 3.3).

3.1 Programming models

A *programming model* is a conceptual framework that enables programmers to describe and reason about computation [109]. The term *programming model* is more abstract than *programming language* because a language implies a particular syntax. All programming languages have an associated programming model, and programming models can be implemented using mechanisms other than new languages. We describe mechanisms to implement programming models in Section 3.2.

Most commonly-used programming models assume a serial, Von-Neumann-style architecture [38, 151]. Von-Neumann architectures perform computation in a sequentially-executing manner where operations imperatively read-from or write-to a randomly-accessible store of memory, perform an arithmetic operation, or branch control flow. Programming languages that assume a Von-Neumann model include standard C and Fortran.

Most developers are skilled in serial, Von-Neumann-style programming models; however, these models no longer reflect current trends in computer architecture. Commercial and research interests continue to motivate a demand for increasingly powerful computers [30, 35]; however, the historically-taken approach of improving performance by increasing processor

clock speed has hit physical barriers such as the power wall [144, 30]. One way of increasing performance without hitting the power wall is to concurrently utilize multiple processing units. Given this fact, it seems likely that parallelism will play an increasingly central role in future architectures.

To address the changing hardware landscape, industry and academia are developing several new parallel programming models. Examples of general-purpose parallel programming models include — the DARPA HPCS languages [110]: X10 [74], Fortress [31], and Chapel [50]; and the Partitioned Global Address Space (PGAS) languages: Titanium [155], Unified Parallel C (UPC) [59], and Coarray Fortran [118]. Other models have been build around specific types of architectures, for example: CUDA [117], OpenCL [22], and OpenACC [21] for GPUs; OpenMP [62], Cilk [42], and Threading Building Blocks (TBB) [120] for multicore systems; and MPI [75] for distributed-memory clusters. Some models have been developed around particular types of data structures such as STAPL [124] for the C++ Standard Template Library (STL) containers and Hierarchically Tiled Arrays (HTA) [41] for arrays. Additionally a number of models exist for specific domains such as machine learning [143], image processing [53], network simulation [139, 138], and stencil computations [96, 149, 55, 112, 108, 69, 4, 39, 80]. In this dissertation we introduce a new programming model for stencil computations called GridWeaver; we discuss other programming models for stencil computations in Chapter 4.

3.2 Implementation mechanisms

In Section 3.1 we describe a *programming model* as a conceptual framework to specify and reason about computation. In this section we list *mechanisms* that provide ways for developers to execute a modeled program. In Chapter 4 we use the terminology listed here to describe and compare tools for stencil computations. Mechanisms for realizing programming models include:

- **Compiled or interpreted programming languages** [107] — since parallel programming differs from sequential programming a compiler or interpreter for a new language may help users to "think parallel" from the start instead of first writing serial

code then modifying it to execute on parallel hardware. Also, since a new language restricts what users can express, a compiler's analysis and transformation passes may be able to make assumptions that they could not otherwise make. For example, if the new language does not include pointers, data-flow analyzers can assume that aliasing will not arise due to pointer dereferencing. On the other hand, since new languages use new syntax porting existing applications will require a complete rewrite.

- **Libraries** [147] — since new languages require users to learn new syntax and rewrite code when porting existing applications, some programming models have taken a library-based approach. In this approach users model a computation with classes, structures, and functions defined in a software library. Since libraries can link to programs written in an existing language, software developed in the same language as the library can easily integrate it. One potential downside of a library is that it may introduce overhead.

- **Active libraries** [61] — active libraries address library overhead by either using existing language mechanisms (such as macros or templates) to statically remove overhead, or by using source-to-source transformation tools to replace library calls with more efficient code. GridWeaver takes an active library approach with a source-to-source translator that inlines calls to the library's subroutines and functions.

- **Languages preprocessors** [83] — a preprocessor based approach uses directives to annotate existing code, and uses a preprocessing tool to compile code with the annotations. OpenMP is an example programming model that takes this approach [62]. Some models that take a that take a language-preprocessing approach (such as OpenMP) enable a program to execute using both the original semantics (that is the program will correctly compile and run if directives are ignored) or the new semantics.

- **Languages extensions** [106] — language extensions avoid library overhead by introducing new syntax into an existing programming language and providing a compiler that is cognizant of the new syntax. Over time successful language extensions are

often integrated into the language they originally extended; for example, coarrays from Coarray Fortran (CAF) [118] are now part of the Fortran 2008 standard [119].

In GridWeaver we take an *active library* approach. A strict library approach adds overhead (we discuss this overhead in greater detail in Section 7.1), but we would like to be able to integrate GridWeaver into existing Fortran codes so we have chosen not to develop a new language or compiler. We evaluate using GridWeaver with existing code bases in Chapter 8.

3.3 Tangling and approaches to untangling

In this section we discuss different approaches programming models take to reduce tangling of implementation details with algorithm code. Reducing tangling of details leads to more readable and maintainable code. In Chapter 4 we discuss which of these approaches GridWeaver and existing work take.

Mechanisms that realize parallel programming models must address, or include features to address data and computation decomposition, and communication. We refer to concerns that relate to addressing data and computation decomposition as *implementation details* for parallelism. Implementation details for parallelism include: tile size, data distribution, synchronization mechanisms, and others. Parallelization and program optimization details complicate programming, and code that addresses them often obfuscates and tangles algorithm code. To be precise, the term *obfuscation* indicates that the code is difficult to read and maintain.

The term *tangling* indicates when implementation details are not cleanly separated from algorithm code [60]. *Tangling* implementation details with algorithm code often leads to obfuscation. Programming models research has produced programming languages and libraries with features that prevent tangling by separating implementation details, to various degrees, from algorithm code. Examples include parallel programming languages such as Chapel [48, 50], which provide abstractions for programmer-defined computation and data decompositions; OpenMP [62], which provides pragmas for labeling a loop/algorithm as having `forall` parallelism and indicating implementation preferences; and object-oriented

parallel libraries such as the Standard Template Adaptive Parallel Library (STAPL) [124, 32], which provides orthogonal scheduling of data structure iterators. There are also more restrictive programming models such as the polyhedral programming model [82, 40] that enable orthogonal specification of scheduling and storage mapping within the compiler, and MapReduce [67], which enables implementation details, such as the runtime environment, to evolve during program execution.

In general, it is important to minimize tangling while still providing the programmer control over how implementation details are addressed for performance tuning purposes. There is a tradeoff between minimal tangling and programmer control. Different features in programming models provide different points in this tradeoff space. In Figure 3.1a we illustrate a tradeoff space for programmer-control and tangling for six types of programming-model features.

Within this illustration we identify a goal point of minimal tangling with maximal programmer-control. Generally, less tangling is desirable since it alleviates code obfuscation. However, we argue that some minimal amount of tangling is advantageous since it clarifies where implementation details are relevant. Regardless of the impact a feature has on tangling, maximal programmer-control is desirable since it enables programmers to tune for performance if necessary.

Another impact programming model features can have, is to what degree they alleviate *programmer responsibility*. In order for a program to be on parallel hardware, implementation details such as computation decomposition must be addressed either by the user (through a feature in a programming model) or automatically by software or hardware. We refer to what degree a programmer must manually address an implementation detail as *programmer responsibility*. In Figure 3.1b we illustrate the relative responsibility features leave to programmers for six types of programming-model features.

We refer to the six types of features illustrated in Figure 3.1 as *feature classes*. Each feature class describes how a feature exposes itself in algorithm code. We arrange the classes within the figure to illustrate the relative impacts they have on tangling, programmer control,

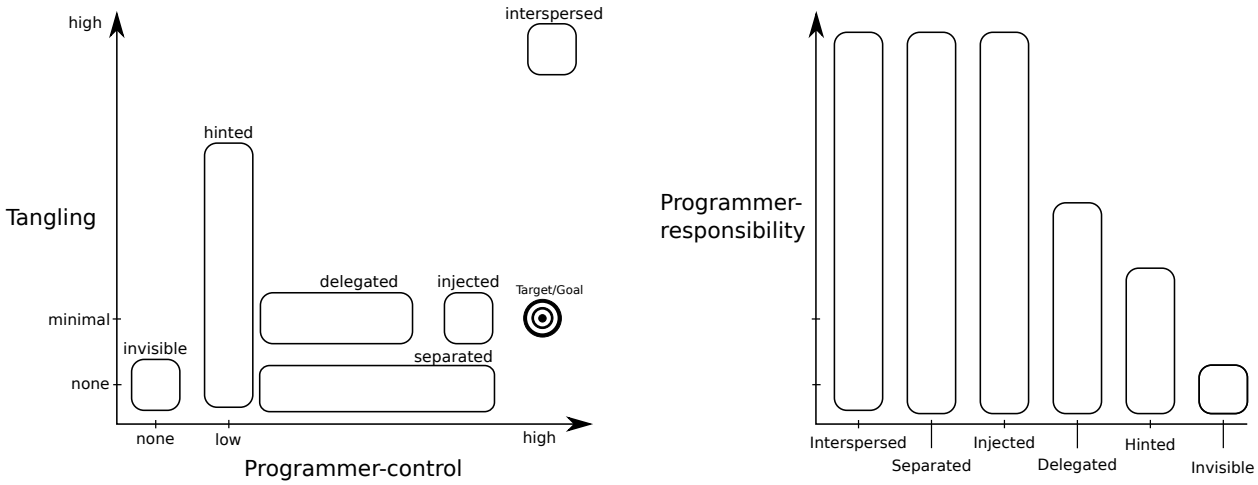


Figure 3.1: Features and impacts on tangling, programmer-control, and programmer-responsibility. (a) Relative classification of features and their programmer-control and tangling impacts. (b) Relative classification of features and their programmer-responsibility impacts.

and programmer responsibility. Thus, by identifying into what class a given feature falls into, one can gain insight into what impacts that feature has on tangling and programmer-control. We also illustrate several points in the figure that correspond to sample programming language features. In this section we define each of these six classes, namely: interspersed, invisible, hinted, separated, injected, and delegated.

- **Interspersed** — code addressing an implementation detail appears in algorithm code. Features in this class have the drawback of a high degree of tangling, but have the benefit of a high degree of programmer-control. For example the manual update for pole values in Figure 1.3 falls into this class. Interspersed features often do little to alleviate programmer responsibility.

- **Invisible** — some overarching system opaquely handles the implementation detail. An example would be a parallel library that determines how many threads to instantiate, but does not provide any mechanism for programmers to override or influence this decision. Another example would be automatic parallelization via compiler. Features

in this class are not exposed in algorithm code, and in fact are not exposed to the programmer at all. Thus there is no tangling, but at the cost of no programmer-control. Invisible features completely alleviate programmer responsibility.

- **Hinted** — users supply hints that guide how some overarching system should address the implementation detail. The overarching system is not required to follow these hints. One example would be specifying a processor affinity hint for a task that the runtime library may choose to ignore. Hinted features can have varying degrees of tangling depending on where they appear in a piece of code. There is no tangling if the hints appear outside of the algorithm code, but there is some tangling if this is not the case. Typically, features of this class have a low degree of tangling, but likewise provide a low level of programmer-control. Hinted features often alleviate programmers of responsibility.

- **Separated** — some external specification defines how to modify algorithm code. The programmer has access to this modification specification; however, nothing in the algorithm code indicates that modification occurs. The degree of programmer-control a separated feature has is dependent on how expressible these modifications can be. For example, aspects in Aspect Oriented Programming (AOP) languages are a separating feature [99]. However, AOP languages limit adding functionality to expressible join points. External configuration files are also a form of a separated feature, but limits programmer-control to what is expressible within such files. Features that fall in this class do not tangle algorithm code, but have a variable amount of programmer-control (always more control than invisible features but not necessarily as much as interspersed ones). Separated features do not alleviate programmers of much responsibility.

- **Injected** — programmers address implementation details via code exposed in some orthogonal data or algorithmic structure. Where to add injected details is explicitly stated in algorithm code with the injected factored out somewhere else. Iterators and normal function calls would fall into this category. Injected features have low tangling,

but limit programmer-control by the mechanism used to call the injected code. Injected features do not alleviate programmers of much responsibility.

- **Delegated** — programmers address implementation details behind some non-modifiable structure. As in the injected class, delegated features explicitly specify in the algorithm code the location at which to put details. However, unlike injected features, the implementation of delegated features is not specified in code available to the programmer. Delegated features often expose parameters to programmers. The more parameters a delegated feature exposes, the higher degree of programmer control it provides. Delegated features imply a similar level of tangling as injected features, but typically have less programmer-control. OpenMP's pragmas are examples of delegated features, as are BLAS function calls. Delegated features alleviate programmers of some responsibility.

The GridWeaver programming model includes abstractions to specify stencil algorithms in an injected manner, specify decompositions and grid-connectivity in a delegated manner, and address communication in an invisible manner.

Chapter 4

Programming models for stencil computations

Several tools, libraries, languages, and programming models exist to aid in the implementation of stencil computations [4, 55, 56, 69, 80, 81, 96, 108, 112, 150]. These tools take various approaches to represent grid connectivity and address implementation concerns such as data distribution and stencil-computation iteration order. These various approaches make different tradeoffs in terms of the qualitative tangling, programmer control, and programmer-responsibility metrics we discuss in Section 3.3. Ideally a tool should exhibit minimal tangling and leave the programmer with minimal amount responsibility while still giving the programmer a high degree of control. In this chapter we discuss what approaches different stencil computations tools take to addressing implementation concerns.

We begin this chapter, in Section 4.1, by broadly discussing and comparing the motivations behind the development of stencil computation tools. In Section 4.2 we discuss a tradeoff that exists in terms of grid connectivity and efficiency of data access. GridWeaver strikes a balance between these two concerns by targeting the semiregular class of grids. Finally, in Section 4.3 we discuss how GridWeaver and other stencil computation tools address various implementation concerns.

4.1 Motivation for stencil computation tools

The tools we discuss in this chapter involve the creation of new programming languages and models for stencil computations. The development of new models and languages is typically motivated by the desire to improve programmer productivity through hiding or separating implementation concerns. For example, industry and academia developed early languages and compilers [89] so that programmers could avoid considering low level implementation concerns

apparent in assembly code such as the movement of data to CPU registers. Later programming languages further raised the level of abstraction; especially the domain specific languages (DSLs), which are languages tailored to a specific application domain [114]. Example DSLs include: Make [91] for build systems, HTML [152] for web page design, YACC [93] for parsers, and SQL [34] for database queries.

Although there may seem to be an intractable number of domains for DSLs to describe, at-least in the domain of scientific computing, there are certain characteristic algorithms that commonly emerge. Phillip Colella [57] described seven such algorithms, which he termed the seven "dwarfs." Later, The Berkeley View [35] extended this set into thirteen "motifs," which are patterns emergent in scientific and parallel computing.

Tools such as Mint [150], Patus [55], Physis [112], Pochoir [108], and Chombo [56] help programmers implement computations described by the Berkley View's [35] *structured grids* motif; whereas tools such as OP2 [80] and Liszt [69] help programmers implement computations for *unstructured-grids* motif. GridWeaver is a DSL that targets a motif not explicitly defined by the Berkley View [35] that lies somewhere between the existing *structured grids* and *unstructured-grids* motifs. Emerging trends in high performance computing architecture motivate the need to improve programmer productivity for the structure grid, unstructured grid, and semiregular-grid motifs.

The Exascale Working Group [3, 72], a committee of researchers and scientific computing experts funded by the Department of Energy and National Science Foundation, predicts that Exascale Computing systems (systems have a peek compute bandwidth greater than 10^{18} flops) will have a power budget of at-most 20 Megawatts [2].

Due to the demand for greater performance and the constraints placed on power, many researchers are predicting that the Exascale systems will contain significantly more parallelism within a node rather than across nodes [79, 131]. Unfortunately, existing scientific applications

[2]To put this into perspective: in 2011 the energy demand of the average United States residential utility customer was 1.29 kilowatts [26].

based on a distributed-memory style of parallelism will not automatically benefit from these new architectures and thus in order to fully utilize emerging hardware, in the absence of an automated solution or programming model that can abstract away the hardware details, developers of structured grid, unstructured grid, and semiregular grid applications will have to manually rewrite code. This is what has motivated the development of domain specific programming models for grid applications.

4.2 Grid connectivity and data structure tradeoffs

Stencil computations typically store data in either arrays or graphs. Arrays work well for regular grids and graphs are necessary for irregular grids because a tradeoff between flexibility and efficiency of data access exists between these two datastructures, and in this section we discuss this tradeoff and the decisions various stencil computations tools have made regarding it.

We illustrate an example regular grid in Figure 4.1c and an example irregular grid in Figure 4.1a. Formally, a regular grid is a mesh that consist of a series of cells that tessellate an n-dimensional rectalinear space in a regular pattern An irregular grid is a mesh that discretizes the surface of some underlying geometry with a series of simple polytopes such as triangles or tetrahedra. We refer to each of the discretizing shapes (rectangles, triangles, tetrahedra, etc.) as cells. In Figures 4.1a and 4.1c we present cell based representations of grids and in Figures 4.1c and 4.1d we present corresponding node-and-edge based representations. In the node-and-edge based representation each node corresponds to a cell from the corresponding subfigure and each edge represents an adjacency relationship between two cells.

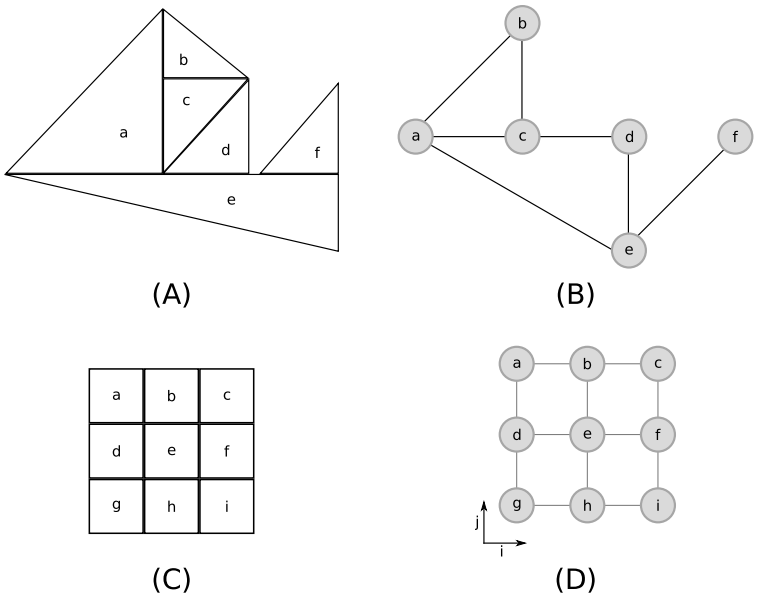


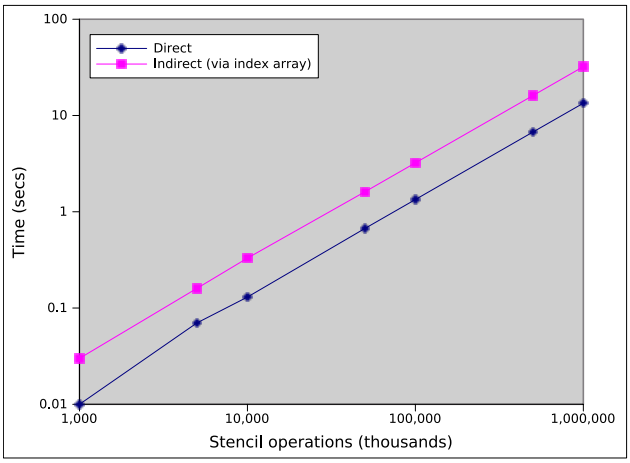
Figure 4.1: Regular vs irregular grids. In subfigure (a) we present a cell based representation of an irregular grid and in (b) we present a corresponding graph based representation. In (c) we present a cell based representation of a regular grid and in (d) we illustrate an array that could contain it. Notice that the connectivity pattern in (d) is implicit.

```
1  for i = 1 to n
2    for j = 1 to m
3      A(i,j) = A(i-1, j) + A(i+1, j) +
4               A(i, j-1) + A(i, j+1)
```

Figure 4.2: Pseudocode of stencil using direct data access. This type of code is common for computations on regular grids.

```
1  for x = 1 to n do:
2    for neigh = 1 to numNeighs(x):
3      A(x) += A(neighbor(x, neigh))
```

Figure 4.3: Pseudocode of stencil using indirect data access (via an adjacency list). This type of code is common for computations on irregular grids.



1000s of calls	direct (s)	indirect (s)	ratio
1,000	0.01	0.03	3.00
5,000	0.07	0.16	2.29
10,000	0.13	0.33	2.54
50,000	0.67	1.61	2.40
100,000	1.34	3.22	2.40
500,000	6.72	16.08	2.39
1,000,000	13.42	32.16	2.40

Figure 4.4: Log-log plot of cost of accessing data in three point stencil via index array. We list the code used for this experiment in Figure C.4 of Appendix C.

Note that there is a constrained and predictably regular pattern to the adjacency relationships in the example of the regular grid (Figure 4.1d). As such these adjacency relationships do not need to be explicitly stored, and thus arrays work to store data for regular grids. On the other hand, for irregular grids, connectivity patterns can be more nuanced, but (Figure 4.1b) adjacency relationships must be explicitly stored. Thus for irregular grids, graph datastructures are commonly used.

The overhead incurred from using a graph based versus array based datastructure is not only the extra storage required. Additionally, stencil computations that operate on graphs require the use of indirect array access. We show pseudocode for an example stencil computation in Figure 4.3.

Note that in this example to access neighboring value for some node `x` through an array of indices `neighbor`. Compare this to the pseudocode we list in Figure 4.2, which is an example of a stencil computation for a regular grid.

In Figure 4.4 we plot the execution time for a sample three-point stencil using either accesses data access or indirect access through an index array. In the indirect version of the benchmark the index array is a random permutation of all indices into the data array; we do this to model the worst case scenario for a graph based data structure. Note the consistent overhead incurred by the indirect access: on average, the indirect version of the benchmark performs 2.4 times slower than the direct version. Thus there is not only a storage advantage to using arrays but also a performance advantage.

The indirection cost shown in Figure 4.4 occurs not only due to the fact that the indirect version of the code performs twice as many array accesses, but whereas the direct access version streams through memory, the indirect access version accesses memory in a more haphazard manner. This haphazard access of memory causes more cache misses to occur.

Thus a tradeoff exists between flexibility of grid connectivity versus data-access. In Section 2.1 we discuss an additional class of grids, which we call semiregular. Semiregular grids consist of a finite number of regular sections that may connect to one another in an irregular fashion. We illustrate semiregular grids in Figures 2.1 and 2.2, and we designed the GridWeaver tool to focus on this type of grid.

Other stencil computation tools have focused on other types of grids. Tools for regular grids include Mint [150], Physis [112], Pochoir [108], and Kamil et al's. autotuning framework [96]. Tools for irregular grids include OP2 [80] and Liszt [69].

The Chombo [56] framework focuses on mapped multiblock grids [9]. Like GridWeaver's semiregular grids, mapped multiblock grids consist of a set of subgrids. In GridWeaver, border mappings describe connectivity between subgrids; we discuss border mappings in Section 5.1.1. Contrarily, mapped multiblock grids specify connectivity with an equation that maps between a two-dimensional grid domain and a three-dimensional range. Points in the three-dimensional range correspond to the underlying geometry discretized by the grid.

Additionally, Chombo also focuses on adaptively refined grids. In these grids, a portion of a regular subgrid may be adaptively refined to form an embedded, finer grained, subgrid. In such a case Chombo will have the coarser, enclosing, grid will copy data from the neighbor points in the finer, enclosed, grid; an averaging operator translates between the finer and coarser data [39].

The Overture system also has the ability to work on grids composed of multiple subgrids. In Overture [46] users specify a geometry; Overture then discretizes that geometry with overlapping regular grids.

The Model for Prediction Across Scales (MPAS) [4] includes routines to apply stencils on a specific type of irregular grid called Voronoi meshes [73]. To generate Voronoi meshes programs take a set of seed points than generating cells such that each cell corresponds to a region where all containing points are closer to a given seed than any other. The generated Veronoi cells forms a dual graph as compared to Deluanay triangulation [141, 36].

4.3 Comparing existing stencil tools

In this section we compare existing stencil tools to GridWeaver and each other. Specifically, we examine the autotuning framework by Kamil et al. [96], the Cactus framework [81], Chombo [56], Liszt [69], Mint [150], MPAS [4], OP2 [80], Overture [46], Patus [55], Physis [112], and Pochoir [108]. For each tool we identify the mechanism used to realize the programming model (Section 4.3.1), the types of grids and stencils that are expressible (Sections 4.3.2 and 4.3.3), the means of specifying stencils (Section 4.3.4) the means of specifying or addressing grid connectivity (Section 4.3.5) , the means of specifying or addressing iteration order for stencil (Section 4.3.6), and the means of specifying or addressing decomposition of data for parallelism (Section 4.3.7).

4.3.1 Mechanism

Mechanisms are ways of realizing programming models. Possible mechanisms include: new languages and compilers, libraries, active libraries, languages preprocessors, and languages

extensions. Different mechanisms make different tradeoffs regarding interoperability with existing code, analyzability, and performance. For example, it may be easier to analyze code written in a new language but a new language may no interoperate with existing code. We broadly discuss and compare different mechanisms in Section 3.2.

In GridWeaver, we take the approach of developing an active Fortran library. By takes this approach because it enables us to interoperate with existing Fortran codes and debugging tools. By including a source-to-source translation tool with GridWeaver, we are able to eliminate library overhead.

OP2 [80] also takes an active library approach, but operates with C++ code and does not target semiregular grids. MPAS [4] takes a Fortran library approach and Overture [46] takes a C++ library based approach, but these tools do not include a source-to-source translation tool to eliminate library overhead.

The autotuning framework by Kamil et al. [96] and Mint [150] take preprocessor based approaches. In these tools users annotate loop nests with pragmas to indicate where to perform a parallel stencil computation. A preprocessor based approach maintains serial semantics so that programs can be executed in a serial manner without passing them through the preprocessing tool. Like our active library based approach this provides the user with a way to debug their original code.

Patus [55] introduces a new DSL for stencil computations, and Physis [112], Pochoir [108], and Liszt [69] extend a base language with new, domain specific, syntax for stencil computations. Physis [112] extends C, Pochoir [108] extends C++, and Liszt [69] extends Scala. DSLs based on language extensions have both the advantage of introducing new syntax and the advantage of being able to interoperate with existing code.

4.3.2 Grid type and data structure

This dissertation focuses on semiregular grids; that is grids that consist of a finite number of regular subgrids. Programs can store each of the regular portions of a semiregular grid in an array. Semiregular grids reach a compromise between the flexibility of irregular grids,

which are typically stored in graphs, and regular grids, which are arrays with internally regular connectivity. Additionally, many tools allow for regular grids with *periodic boundaries*; that is: a boundary that wraps around two opposing edges of a subgrid. In a two-dimensional grid periodic boundaries allow for ring or torus topologies. Grids may also be *Adaptively refined*; that is: portions of the grid may be dynamically re-discretized into a finer resolution. Adaptively refined computations use extrapolation and interpolation to convert values between the coarser and finer regions of the grid.

A tradeoff exists between flexibility of grid connectivity and the data structure used to store grid data. Graph based data structures offer the most flexibility in terms of possible connectivity patterns but data is often accessed through arrays. Arrays offer direct access but limit possible connectivity patterns. In Section 4.2 we discuss this tradeoff in greater depth.

The autotuning framework by Kamil et al. [96], Cactus [81], Mint [150], Overture [46], Patus [55], Physis [112], and Pochoir [108] operate on regular grids and store data in arrays. Liszt [69] and OP2 [80] operate on irregular grids and store data in graphs. Chombo [56] and Cactus [81] are able to operate on adaptively refined grids and store data in arrays.

4.3.3 Stencil type

Simulation applications often solve partial differential equations by discretizing data and performing stencil computations. Programs use different stencil operations to solve different types of equations. *Compact stencils* update each cell in a grid using the values at that cell and the immediately surrounding neighbors; likewise, *non-compact stencils* update each cell but may use values from neighbors further out. Stencils also have *dimensionality*; that is: whether the stencil operates in a one-dimensional, two-dimensional, three-dimensional, or n-dimensional space.

GridWeaver is able to apply both compact and non-compact stencils. Physis [112], Pochoir [108], OP2 [80], Mint [150], and Liszt [69] are able to apply non-compact stencils. Overture applies compact stencils but users specify computations in terms of partial differential equations to solve rather than write code for the stencil directly.

4.3.4 Stencil specification

Some tools have users specify stencil computations in C or Fortran functions, others include a domain-specific syntax. In GridWeaver we take an approach of having stencils specified in an injected manner: users write a stencil function that is separated from other implementation concerns such as grid connectivity and iteration specification. Patus [55], Physis [112], and Pochoir [108] take a similar, injection based, approach.

The Autotuning Framework by Kamil et al. [96] take an interspersed approach by having users explicitly write stencil functions in loop nests. Since this is the approach taken when writing C+MPI or Fortan+MPI, the autotuning framework by Kamil et al. will work with previously written code.

In the Overture system [46] stencils are specified in a hinted manner. Rather than specify the stencil users specify a partial differential equation (PDE) and the overture system automatically determines what stencil to apply in order to solve the PDE.

4.3.5 Connectivity specification

In GridWeaver users specify grid connectivity in a delegated manner through its border map abstraction. By taking a delegated approach we are to separate grid connectivity from stencil specification, but do not leave the user with the responsibility of explicitly determining how to reindex values when they fall off of a subgrid. Similarly, in Liszt [69] users specify connectivity through a series of node and edge abstractions.

In Pochoir [108] users can express connectivity in an injected manner by writing a boundary map function that is called whenever a stencil accesses an index off the subgrid. This approach enables users a great deal of flexibility (programmer control) over connectivity and enable Pochoir to express a broader class of grids than GridWeaver. For example: in Pochoir, grid connectivity could change between timesteps. However, this approach does require users to explicitly write the re-indexing logic, which depending on the grid, could be complicated.

In Overture [46] grid connectivity is supplied in a hinted manner: users describe some underlying geometry that is discretized and the Overture system automatically infers the connectivity from this specification. In this approach users are not responsible for explicitly considering connectivity, but have little control in terms of what decisions the Overture system makes.

Some systems leave the user to address grid connectivity in an interspersed manner, by explicitly addressing it in loop boundaries and stencil access functions. This is the case for the autotuning framework by Kamil et al. [96] and Mint [150].

4.3.6 Iteration specification

Stencils work by iterating over all elements in a grid and applying a stencil operation. In GridWeaver this iteration process is invisible to the user. This approach frees the user from all responsibility of addressing iteration specification, but leaves the user no control to modify it should they dislike the approach GridWeaver takes. Cactus [81], Chombo [56], OP2 [80], Overture [46], Physis [112], and Pochoir [108] also address iteration order in an invisible manner.

At the other end, the autotuning framework by Kamil et al. [96] and Mint [150] users explicitly write loop nests for stencils around the specification for a stencil operation. Since users can modify these loop nests it gives them a good deal of control over iteration order, but at the expense of tangling code.

Patus [55] takes another approach by having users write iteration order in a separated manner inside a strategy file. This approach leaves users with as much responsibility as an interspersed approach, but does not have the disadvantage of tangling code.

4.3.7 Decomposition specification

Most parallel stencil computations follow an owner-computes style of data-parallelism. That is, they partition and distribute data to processors, and a given processor is responsible for updating the data it owns. In GridWeaver decomposition is specified in a delegated manner.

We restrict subgrids to being partitioned into a finite number of equally-sized blocks but users can explicitly assign the ownership of blocks to processors. Several tools address decomposition in an invisible manner: they automatically decompose data and computation. This is the case for the autotuning framework by Kamil et al. [96], OP2 [80], Overture [46], Physis [112], and Pochoir [108]. Patus [55] allows users to address decomposition a separated manner through its use of strategy files. Mint [150] handles decomposition in a delegated fashion by having users specify decomposition using parameters within a preprocessor directive.

4.3.8 Targets

In order for a program to compute anything, some underlying architecture must execute it. Architectures that stencil tools target include *distributed memory clusters*, *shared memory multicore* systems, *GPUs*, and *accelerators* (vector processing units). In addition to hardware target there is the software target: what mechanism realizes the post-processed or compiled code for the hardware architecture. Examples software targets include: pthreads [1], MPI [75], OpenMP [62], OpenACC [21], CUDA [117],and OpenCL [22].

GridWeaver targets distributed memory clusters with Fortran+MPI [75]. We target clusters programmed with MPI, since Earth science simulation codes are commonly run on this type of machine. Cactus [81], Chombo [56], Liszt [69], MPAS [4], OP2 [80], and Physis also target distributed memory clusters with MPI. Cactus and OP2 also target multicore systems with OpenMP. Although MPI code can also run on multicore systems cores still communicate with each other through message passing rather than directly accessing their shared memory so communication on a multicore when using MPI may not be done as efficiently as possible. Pochoir [108], Patus [55], and the autotuning framework by Kamil et al. are also able to target multicore systems, but they are not able to target distributed memory clusters. Cactus [81], Mint, OP2, Patus, and Physis are also able to target GPUs programmed with CUDA.

4.4 Summary

Commercial and research interests motivate the need for increasingly powerful computers, but because processor clock speeds have hit physical barriers such as the power wall [144, 30] future architectures will likely utilize greater degrees of parallelism. Due to the difficulty in writing and porting parallel codes, a number of domain-specific programming models have emerged for common types of algorithms seen in high-performance codes. The Berkeley View [35] lists thirteen motifs for such algorithms; among them are the structured-grid and unstructured-grid motifs. Thus a number of tools have been developed for structured, and unstructured, grid computations [4, 55, 56, 69, 80, 81, 96, 108, 112, 150]. We discuss the motivation behind these tools in greater detail in Section 4.1.

Earth-science simulation programs conduct stencil computations on semiregular grids, which fall somewhere in-between the structured and unstructured grid motifs. Although semiregular grids can be modeled as unstructured grids, doing so would require grid data to be stored in a graph-based datastructure, which would incur a performance penalty (see Section 4.2). In GridWeaver we enable the specification of semiregular grids and store grid data in arrays.

In Section 4.3 we compare GridWeaver against several existing tools for specifying and conducting stencil computations. Specifically we compare and examine the mechanism used to realize the programming model (Section 4.3.1), the type of grids and stencils that are expressible (Sections 4.3.2 and 4.3.3), the means of specifying stencils (Section 4.3.4) the means of specifying or addressing grid connectivity (Section 4.3.5), the means of specifying or addressing iteration order for stencil (Section 4.3.6), and the means of specifying or addressing decomposition of data for parallelism (Section 4.3.7).

To enable GridWeaver to work with existing code, it uses an active library based approach, which is also seen with the OP2 tool. Both GridWeaver and Chombo are able to express semiregular grid, but they do so with different abstractions (GridWeaver uses border maps, Chombo uses equations to relate a mapped space to a real space). GridWeaver, Physis [112],

Pochoir [108], OP2 [80], Mint [150], and Liszt [69] are all able to apply both compact and non-compact stencils. GridWeaver, along with Patus [55], Physis [112], and Pochoir [108], enable users to specify stencil computations in an injected manner.

Chapter 5

Semiregular-grid abstractions

GridWeaver is an active library and programming model for specifying stencil computations on semiregular grids. We start this chapter by introducing the abstractions that make the GridWeaver programming model (Section 5.1) and then we discuss how these abstractions are formally defined and implemented (Section 5.2).

5.1 User view of abstractions

The GridWeaver programming model includes abstractions for describing grid topologies (Section 5.1.1), decomposing grid data (Section 5.1.2), and performing operations on grid data (Section 5.1.3). In this section we discuss these abstractions and describe how users construct them with the GridWeaver library; in appendix B we provide a full reference guide to the library's abstractions and routines. In this section we also give an example implementation of a five-point averaging stencil on the tripole grid (Figure 5.4).

5.1.1 Grid connectivity

There are three main types of abstractions we use to specify connectivity of semiregular 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 5.1 we present the interface for defining connectivity in GridWeaver. Although GridWeaver is a Fortran 90 library we use an object-oriented approach in its design. The functions `subgrid_new`, `grid_new`, and `grid_addBorder` to construct new subgrids, grids, and border maps.

Construct a subgrid of size n by m:

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

Construct a new grid:

```
grid_new()
```

Add subgrid sg to the a grid g:

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

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):

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

Figure 5.1: Interface for specifying semiregular grid connectivity in GridWeaver library

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 1.2 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 we could generalize our approach to handle greater dimensionality. Note that this limitation does not prevent stencils from operating on higher dimensional data: for example, each point of a grid could contain an array of values rather than a scalar. This limitation does prevent a grid from having differing connectivity along the higher dimensions; however, for the CGPOP 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 GridWeaver, subgrids connect to one another via border mappings. Border mappings assume that a layer of nodes, called a halo, surround subgrids. Border mappings indicate connectivity with two rectangles: a target rectangle corresponding to a region of nodes within a subgrid's halo, and a source rectangle that lies along the border of another (or the

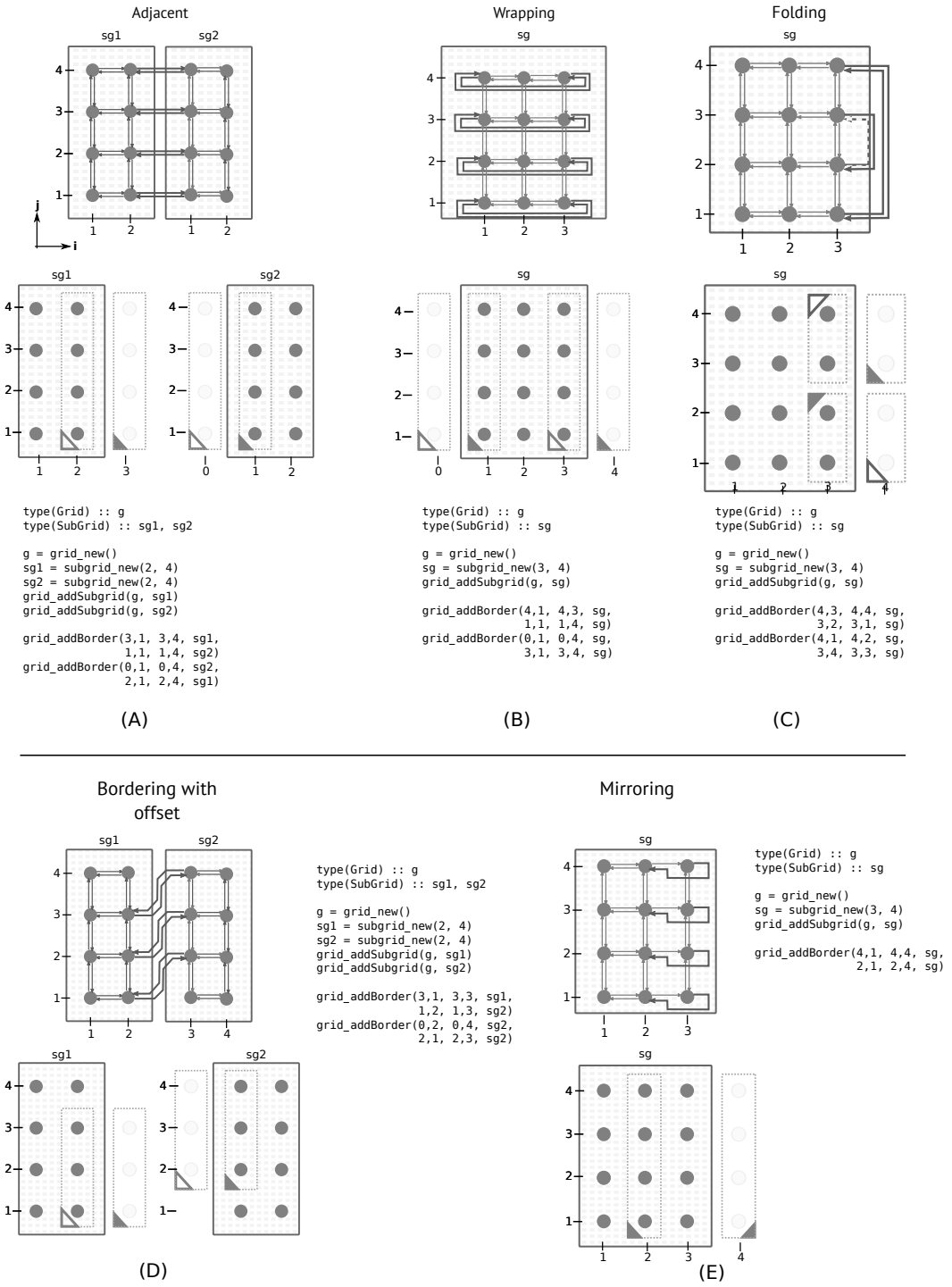


Figure 5.2: Examples of border connectivity patterns. In each subfigure we include an illustration of the connectivity pattern between nodes in the subgrid. Below each connectivity pattern illustration we show how border mappings represent the connectivity (the dashed rectangles). Below (in a,b, and c) or next (in d and e) to that we show code using the GridWeaver interface from Figure 5.1 that specifies the connectivity.

same) subgrid. The mapping signifies that when conducting communication GridWeaver should copy values from the source region into the target. For example, in Figure 5.2a the border mapping `grid_addBorder(3,1, 3,4, sg1, 1,1, 1,4, sg2)` indicates that the target rectangle in the halo of the right hand should copy its values to fill the rectangle on the left hand subgrid. With border mappings we are able to model the complex connectivity patterns seen along array borders in semiregular grids (see Figures 5.2b and 5.2c for more examples).

Typically the halo stores copies of data from neighboring subgrids. After populating a halo GridWeaver can perform a stencil computation 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 is achievable; however, for specification purposes the user only need think about a halo of nodes existing around the subgrid. In this dissertation and in our current implementation of GridWeaver we limit halos to be of single element depth. An implication of this is that stencil operations that work on the grid must be compact (only accessing nearest neighbor values).

Figure 5.2 illustrates different patterns of mapping that occur in Earth grids. In this figure we illustrate how to specify each pattern with border mappings and provide code that uses the interface in Figure 5.1 to specify the mapping. One thing to note is that the rectangular regions specified by the mappings have an orientation. In Figure 5.2 we illustrate orientation using small triangles in one of the four 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 `grid_addBorder` function in Figure 5.1, 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 5.2 exist in actual grids. Cubed-sphere [128] and icosahedral grids [86, 148] use the adjacency pattern in Figure 5.2a. Toroidal and dipole grids [134] use the wrapping pattern in 5.2b. The tripole grid uses the folding

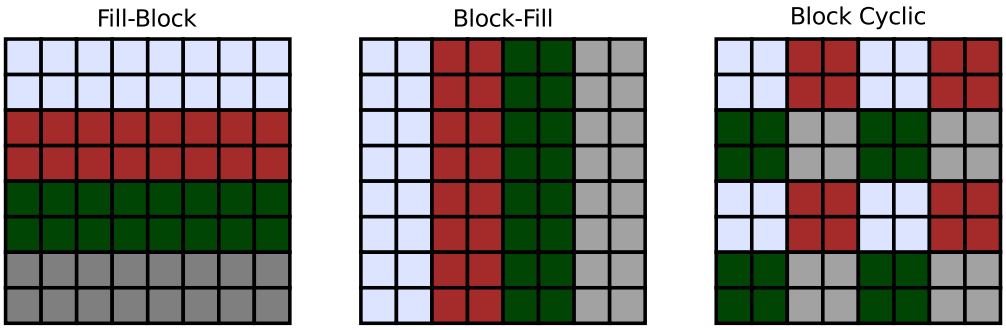


Figure 5.3: GridWeaver includes functions for decomposing an array into blocks in the typical block and cyclic combinations. Each color in this figure represents a different process.

pattern in 5.2c, and the icosahedral grid [116] uses the offset pattern illustrated in 5.2d. We include the mirroring pattern in 5.2e since array-based languages such as ZPL [51] use it.

5.1.2 Grid decomposition

GridWeaver allows users to assign values to nodes after they have defined a grid's connectivity. If a stencil computation is to run in serial then a simple strategy of storing each subgrid in a separate array works. On the other hand, if the stencil is to run in parallel then it is necessary to decompose it into chunks and to distribute chunks to 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 and 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 GridWeaver 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 5.3 we illustrate three different

decomposition patterns; we base these patterns on the array decomposition features of High Performance Fortran [101]. Users may also manually assign or modify the decomposition.

In line 45 of Figure 5.4 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 5.1.1 to define the connectivity of the tripole grid.

5.1.3 Stencil algorithms

After users have specified a decomposition, GridWeaver allows them to assign values to grid nodes and perform stencil operations can update these values. Figure 5.4 shows a complete example program that uses GridWeaver to perform a stencil operation on the tripole grid [116].

Before applying stencil operations, GridWeaver requires that users instantiate data objects. Data objects map values to grid nodes. GridWeaver includes functions for reading and writing data objects to and from files. The functions that construct data objects take 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 takes a data object and 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 5.4.

In GridWeaver, users specify the stencil operations by writing them in separate function. Stencil functions take the coordinates of a given point (`i`, `j`) and a function `A` that retrieves 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 1.4.

```
module Stencils; contains
    real function fivePtAvgStencil(A, i, j)
        fivePtAvgStencil = 0.2 * &
            (A(i, j) + A(i-1, j) + &
            (i+1, j) + A(i, j-1) + A(i, j+1))
    end function
end module

program StencilOnTripole
    type(SubGrid) :: sg
    type(Grid) :: g
    type(Decomposition) :: dcmp
    type(Data) :: data_in, data_out
    integer :: N, M

    ! Create subgrid
    N = 2048; M = 2048
    call subgrid_new(sg, N, M)

    ! Create a tripole grid
    call grid_new(g)
    call grid_addSubgrid(sg)

    ! Wrap left and right borders
    call grid_addBorder(  0, 1,    0, M,  sg,
                        N+1, 1,  N+1, M,  sg)
    call grid_addBorder(N+1, 1,  N+1, M,  sg,
                          1, 1,    1, M,  sg);

    ! Fold top border
    call grid_addBorder(1, M+1,    N/2, M+1, sg
                        N, M,    N/2+1, M,   sg)
    call grid_addBorder(N/2+1, M+1,  N, M+1, sg,
                          N/2, M,    1, M,   sg)

    ! Specify a decomposition and input data
    dcmp = decomposition_new_block_fill(g, 64, 64)
    data_in = data_input("input.dat", dcmp)
    data_out = data_new(dcmp)

    ! Perform stencil operation
    call data_apply(data_out, data_in, fivePtAvgStencil)
end program StencilOnTripole
```

Figure 5.4: GridWeaver code for stencil on tripole grid. Fortran source using GridWeaver for a five point stencil applied to a tripole grid. The code uses the connectivity functions from Figure 5.1. The `decomposition_new_block_fill` function applies a block-fill decomposition (illustrated in Figure 5.3) to grid g where each block is 64 by 64 nodes. The call to `data_input` 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.

5.2 Formalizing and implementing abstractions

In the previous section we introduced abstractions for specifying the connectivity of semiregular grids. These abstractions, which we have implemented in the GridWeaver library, enable a separate specification of grid connectivity 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.

5.2.1 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 1.2 have five neighbors while most other nodes in the grid have six.

When a stencil computation attempts to access a non-existent neighbor GridWeaver 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 linear combinations where a coefficient multiplies the value of each neighboring node, 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.

5.2.2 Communication planning for halos

We have found one of the most complicated portions of implementing semiregular 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 [135] (a 3,214 line miniapp of POP) communication takes 1,147 lines of code.

Communication in these applications and in GridWeaver is cleanly separated by making use of halo regions around blocks of data. However, in SWM and CGPOP this code is specific to the grid used. In GridWeaver we have implemented a generic communication algorithm that will populate block halos for any semiregular 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 GridWeaver, that will properly populate halo values for a semiregular grid. To do this we begin by formalizing the semiregular grid abstractions introduced in Section 5.1. These formalisms describe how GridWeaver stores and constrains its abstractions. Also, we write the communication algorithm (shown in Algorithm 5.6) in terms of these formalisms.

5.2.3 Formalisms for semiregular grid abstractions

We define a subgrid σ as a tuple:

$$\sigma = (s, n, m)\,, \tag{5.1}$$

where s is a value (integer) used to uniquely identify the subgrid, n is the width of the subgrid, and m is the height of the subgrid. We define the index space of a subgrid $I(\sigma)$ to be:

$$I(\sigma) = \{(s, \langle i, j\rangle) \mid s, i, j \in \mathbb{Z} \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m\}. \tag{5.2}$$

All subgrids have an associated halo $H(\sigma)$ and set of borderpoints $B(\sigma)$. The halo $H(\sigma)$ of a subgrid σ is:

$$H(\sigma) = \{s, \langle i, j\rangle \mid \quad s, i, j \in \mathbb{Z} \wedge (i = 0 \vee i = n + 1 \vee j = 0 \vee j = m + 1)\,, \tag{5.3}$$

and the set of border points $B(\sigma)$ of a subgrid σ is:

$$B(\sigma) = \{s, \langle i, j\rangle \mid \quad s, i, j \in \mathbb{Z} \wedge (i = 0 \vee i = n \vee j = 0 \vee j = m)\,. \tag{5.4}$$

Border maps specify how nodes in a subgrid σ_1's halo $H(\sigma_1)$ map to nodes within subgrid σ_2's border points $B(\sigma_2)$. Note that border maps can apply to the same subgrid (that is σ_1 may equal σ_2). We define a border map β (equation 5.12) with two rectangular regions ρ_{tgt} and ρ_{src} (equations 5.7 and 5.8) that respectively lie in $H(\sigma_1)$ and $B(\sigma_2)$ (equations 5.5 and 5.6) where:

$$(s_1, \langle p_i, p_j \rangle), \;\; (s_1, \langle q_i, q_j \rangle) \;\; \in \;\; H(\sigma_1) \tag{5.5}$$

$$(s_2, \langle v_i, v_j \rangle), \;\; (s_2, \langle w_i, w_j \rangle) \;\; \in \;\; B(\sigma_2) \tag{5.6}$$

$$\rho_{\text{tgt}} = (s_1, \; \langle p_i, p_j \rangle \, , \; \langle q_i, q_j \rangle) \tag{5.7}$$

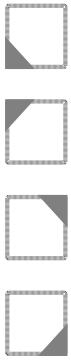
$$\rho_{\text{src}} = (s_2, \; \langle v_i, v_j \rangle \, , \; \langle w_i, w_j \rangle) \tag{5.8}$$

$$\beta = (\rho_{\text{tgt}}, \rho_{\text{src}}) \, . \tag{5.9}$$

It is 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 have different orientations. 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:

$$o(\rho) = \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } q_i \geq p_i \; \wedge \; q_j \geq p_j \\ \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} & \text{if } q_i \leq p_i \; \wedge \; q_j \geq p_j \\ \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} & \text{if } q_i \leq p_i \; \wedge \; q_j \leq p_j \\ \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } q_i \geq p_i \; \wedge \; q_j \leq p_j \end{cases} \tag{5.10}$$



$$f(\rho_{\text{tgt}}, \rho_{\text{src}}) \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if} |p_i - p_j| = |v_i - v_j| \wedge |q_i - q_j| = |w_i - w_j| \\ \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & \text{if} |p_i - p_j| = |w_i - w_j| \wedge |q_i - q_j| = |v_i - v_j| \end{cases} \tag{5.11}$$

The border mapping function $m : \beta \times \mathbb{Z}^3 \rightarrow \mathbb{Z}^3$ uses border maps to translate an index in the rectangle ρ_{tgt} to an index in the rectangle ρ_{src}. We define m as:

$$m(\beta, s_1, \langle i, j \rangle) = (s_2, \langle i', j' \rangle) \tag{5.12}$$

such that

$$\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} v_i \\ v_j \end{pmatrix} + o(\rho_h)o(\rho_b)f(\rho_h, \rho_b) \left(\begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} p_i \\ p_j \end{pmatrix} \right). \tag{5.13}$$

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 is copied into the halo point.

We define a grid as a tuple:

$$\gamma = (S, B) \tag{5.14}$$

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

$$g(\gamma) = \bigcup_{\sigma \in S} \sigma. \tag{5.15}$$

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:

$$\theta = (g, l, p, s, \langle u_i, u_j \rangle, \langle v_i, v_j \rangle), \tag{5.16}$$

where $\langle u_i, u_j \rangle$ to $\langle v_i, v_j \rangle$ is a rectangular region in subgrid s, g is a unique global identifier for the block, l is a local identifier for the block, and p is the process assigned to the block.

We define a decomposition D 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 D 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.

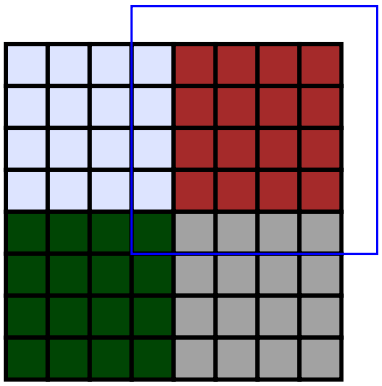


Figure 5.5: Halo of a block in a subgrid. When populating the red block's halo some values come from neighboring blocks in the same subgrid, other come from outside the subgrid.

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

$$\delta(x) : \mathbb{Z}^3 \rightarrow \mathbb{R}. \tag{5.17}$$

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.

5.2.4 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 GridWeaver 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 halo depths is a more difficult and the algorithms presented in this dissertation do not easily scale to multi-depth halos. In Figure 5.5 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 a (n+1)-point stencil operation is applied to a point with less than n neighbors it will

```
routine generate_communication_plan:
  INPUT:
    γ  ! A semiregular grid
    D  ! Data decomposition

  OUTPUT:
    (μ_r, μ_s) ! A communication plan

CODE:

! Iterate through every block in the decomposition
for each θ ∈ D
  let (g, l, p, s, u_i, u_j, v_i, v_j) = θ

  ! Iterate through each block in s that intersects with θ's halo
  let ρ_halo = (s, ⟨u_i − 1, u_j − 1⟩, ⟨v_i + 1, v_j + 1⟩)
  let S = set of blocks (except θ) in subgrid s that intersect with ρ_halo
  for each θ_n ∈ S
    ! Find intersecting region and register a communication transfer
    let (g', l', p', s', u'_i, u'_j, v'_n, v'_j) = θ_n
    let ρ_o = intersecting region of block θ_n and ρ_halo
    μ_r(p) ∪= (l, tr(ρ_o, g), p')
    μ_s(p') ∪= (l', tr(ρ_o, g'), p)

  ! Find regions of border maps intersecting with ρhalo
  let (S, B) = γ
  for each (ρ_tgt, ρ_src) ∈ B
    let ρ_m = intersecting region of ρ_tgt and ρ_halo
    let ρ'_m = analogous region in ρ_src of ρ_m

    ! Find blocks that inersect with the portion of the source side of
    ! the border map intersecting with the halo for θ
    let S = set of blocks that intersect with ρ'_m
    for each θ_n ∈ S
      let (g', l', p', s', u'_i, u'_j, v'_n, v'_j) = θ
      let ρ_o = intersecting region of ρ_m and θ_n
      let ρ'_o = analogously cut region in ρ'_m

      μ_r(p) ∪= (l, tr(ρ_o, g), p')
      μ_s(p') ∪= (l', tr(ρ'_o, g), p)
```

Figure 5.6: Algorithm to generate a communication plan. The function tr(ρ, g) returns a region that translates the global indices used in ρ into the local indices used in block g.

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 GridWeaver uses an abstraction internal to itself called a communication plan. This object contains metadata describing how to conduct 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

$$\pi = (\mu_r, \mu_s), \tag{5.18}$$

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

$$\mu = (b, r, \rho), \tag{5.19}$$

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 5.6 we present a method for generating a communication plan object.

In our GridWeaver implementation we then populate a halo by 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 $\mu_s(r)$. We copy the values from the received calls into the specified region of the specified block's halo.

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.

Chapter 6

Addressing non-compact stencils

In Chapter 5 we introduce abstractions to specify semiregular-grid connectivity, data decomposition, and compact stencil algorithms. Compact stencil algorithms, such as the `fivePtAvgStencil` function in Figure 5.4, are passed a 2D location $\langle i, j \rangle$ and a function that can access values that are immediate neighbors of $\langle i, j \rangle$. Non-compact stencils are those that access values that are two or more elements away from $\langle i, j \rangle$. In this chapter we discuss how non-compact stencil algorithms are modeled and expressed in GridWeaver, and how GridWeaver automates communication for non-compact stencils.

Specifically, in Section 6.1 we introduce notation for describing non-compact halos. We then use this notation, in subsequent sections, to describe different communication algorithms. The three algorithms we introduce are: (1) a naive algorithm, which is sufficient for blocks nested inside a subgrid (Section 6.2); (2) a more sophisticated algorithm that considers orientation changes when moving between subgrids (Section 6.3); and (3) an algorithm that stores block neighboring values in a one-dimensional data structure of ghost cells (Section 6.4). In GridWeaver, we use a mixture of these algorithms: we use the naive algorithm for interior blocks and the third algorithm for border blocks. We use the second algorithm to determine what global index corresponds to a two-dimensional index offset from a given point $\langle i, j \rangle$ when conducting a stencil operation. Throughout this Chapter we illustrate the algorithms using the example of the small icosahedral grid in Figure 6.1. In Section 6.5 we describe how to specify non-compact stencils in GridWeaver, and in Section 6.6 we conclude this chapter by discussing the performance impact of conducting communication for non-compact stencils.

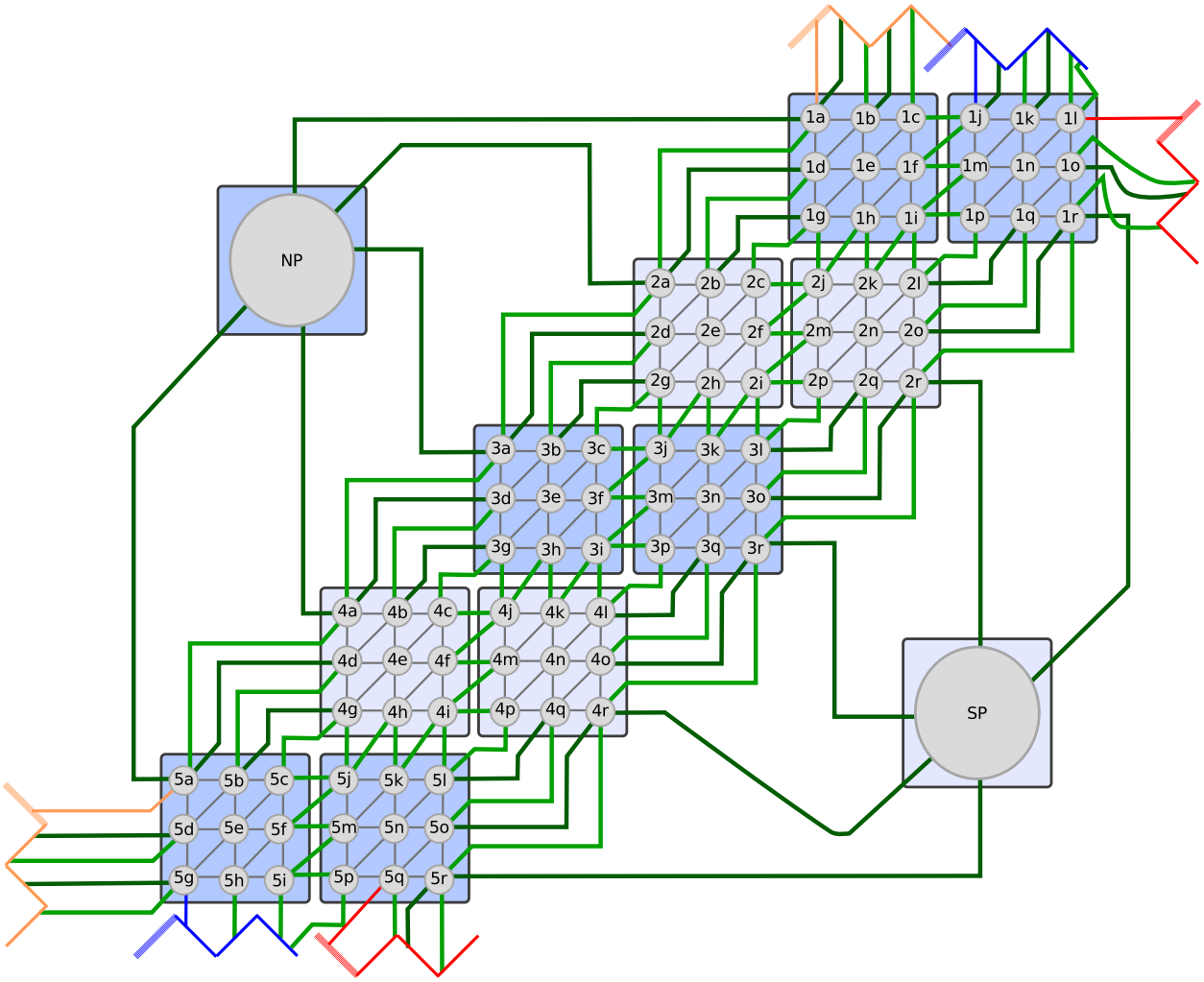


Figure 6.1: A small icosahedral grid. Assume that we distribute data on this grid so that all subgrids consist of single block and each of twelve separate MPI ranks has unique ownership over one of these blocks. To help with referencing, we give each node in the grid a label (NP = North pole, SP = South pole).

6.1 Halos and index spaces

Stencil operations update a value associated with grid node by performing a computation dependent on the node's current values and the values of its neighbors. In the GridWeaver model, we refer to nodes with two different types of indices: local indices $\langle i, j \rangle$, which specify a unique index to a node within a given block, and global indices $(s, \langle i, j \rangle)$, which specify a unique index for a node within an entire grid.

Users execute stencils in GridWeaver by calling the `data_apply` routine. This routine is passed input and output data objects and a user-defined function that specifies the stencil operation. The `data_apply` routine iterates through each grid node and calls the user-specified stencil function. When calling the stencil function GridWeaver passes it a local index for the node currently being iterated over. The user-defined stencil function has access to all neighboring values that fall within the operator's *stencil's domain*. Specifically, for a stencil operation that updates a node $\langle i, j \rangle$ we define its stencil domain to include the set of local indices $I(\langle i, j \rangle, d)$, where:

$$I(\langle i, j, \rangle, d) = \{(x, y) \mid i - d \leq x \leq i + d \ \wedge \ j - d \leq y \leq j + d\}. \tag{6.1}$$

We consider *compact stencils* to be those where d is 0 or 1, and *non-compact stencils* to be those where $d > 1$. In Chapter 8 we describe our case studies with CGPOP, which includes compact stencils, and SWM, which includes both compact and non-compact stencils. Each index in a stencil domain will either either refer to a node within a block or to a node within a block's halo. GridWeaver's communication algorithms populate a block's halo with values, and in this section we introduce a function m that GridWeaver's communication algorithm uses to determine how to populate halos. However, before we introduce m we will give some background material on block's and halos.

Recall that in Equation 5.16 we defined a block θ as:

$$\theta = (g, l, p, s, \langle u_i, u_j \rangle, \langle v_i, v_j \rangle), \tag{6.2}$$

where g is the block's global ID, l is the block's local ID, p is processor ID, and s is a subgrid ID. The points $\langle u_i, u_j \rangle$, $\langle v_i, v_j \rangle$ define the region of subgrid s that is owned by the block. We refer to the set of global indices a block owns as $I(\theta)$, where:

$$I(\theta) = \{(s, \langle x, y \rangle) \mid u_i \leq x \leq v_i, \;\; u_j \leq y \leq v_j\}. \tag{6.3}$$

The set $I_{lcl}(\theta)$ is the set of local indices for a block θ, we define $I_{lcl}(\theta)$ as:

$$I_{lcl}(\theta) = \{\langle x, y \rangle \mid 1 \leq x \leq (v_i - u_i) \;\wedge\; 1 \leq y \leq (v_j - u_j)\}. \tag{6.4}$$

We can translate between local indices and global indices using the translation function t. Given some block θ we define t as:

$$t(\theta, \langle i, j \rangle) = (s, \langle i + u_i - 1, j + v_i - 1 \rangle). \tag{6.5}$$

We define a halo of depth d for a block θ to include the local indices:

$$H_{lcl}(\theta, d) = \{\langle x, y \rangle \mid (1-d) \leq x \leq (v_i - u_i + d) \;\wedge\; (1-d) \leq y \leq (v_j - u_j + d)\} - I_{lcl}(\theta). \tag{6.6}$$

The mapping function m takes a grid γ, a block θ, an index $\langle i, j \rangle \in I_{lcl}(\theta)$ and an index $\langle i', j' \rangle \in H_{lcl}(\theta)$ and returns the global index of the node that should fit in the halo location $\langle i', j' \rangle$ when conducting the stencil for the block location $\langle i, j \rangle$.

In Figure 6.2a we illustrate an interior block of a subgrid and three halos of varying depth surrounding it. We label a block as *interior* when all blocks that surround it lie in the same subgrid. If we assume that a stencil computation is applied on an interior block, and the stencil's depth does not exceed the width or height of the surrounding block, then the indices for the stencil's halo do not have to be resolved through any border maps, and thus in Figure 6.2a we can define m so that the index $\langle i, j \rangle$ in the block's halo maps to the index translated into the global domain by adding the global coordinate of the block's lower-left corner ($\langle u_i, v_i \rangle$). That is:

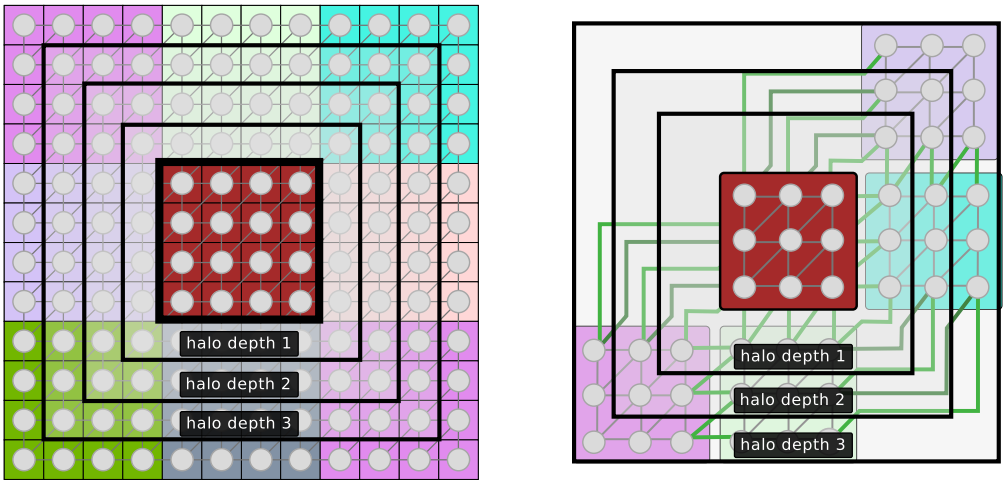


Figure 6.2: Halos of depth 1 through 3 for (a) an interior block of a subgrid and (b) a block in the mini-icosahedral grid (see Figure 6.1).

$$m(\gamma, \theta, \langle i, j \rangle, \langle i', j' \rangle) = (s, \langle i' + u_i - 1, j' + v_i + 1 \rangle). \tag{6.7}$$

However, this mapping function will not work to resolve the halos in Figure 6.2b since it has neighbors that are off of the subgrid. In the following sections we discuss how to resolve this issue.

6.2 Naive halo expansion

In this section we describe an algorithm that calculates values for the mapping function m by performing a synchronized breadth-first search algorithm in both a block's local domain and across a grid's global domain. As we iterate through each corresponding pairs of points, we assign values to m. We give pseudo-code for this algorithm in Figure 6.5, present an example of its execution across the local domain in Figure 6.4, and illustrate how the search is conducted in the global domain in Figure 6.3. In our example we perform the breadth-first search starting from element $3e$ from Figure 6.1. Note that this algorithm will map multiple points to the same halo location, and thus this algorithm will not work in the general case.

In step 1 of Figure 6.4, we illustrate that the breadth-first identifies the neighbors for node 3e as the set $\{3b, 3c, 3d, 3f, 3g, 3h\}$ and places. Since none of these values fall in the block's halo ($H_{lcl}(\theta)$), we do not assign any values to m.

In step 2 several nodes to fall into the halo, when expanding the search frontier to include the nodes 2d, 2g, 4b, 4a, and 4c we not only fall into the halo but also into the subgrid's border $B(\sigma)$ (see equation5.4). To map the local indices of a node falling outside of a subgrid into a global index we can use the border-map mapping function m from Equation 5.12.

In step 3 we expand the search frontier to include neighbors from node 2d notice that the neighbor 2e is one element to the right of 2d as illustrated in Figure 6.1 and so we place it one element to the right in the halo; however, by doing this we are over-writing the value for node 2g that we visited in step 2. Similar overlaps occur in other places in step 3 and subsequent steps. Thus this approach will not generally work to populate a halo of elements around a block of depth greater than 1.

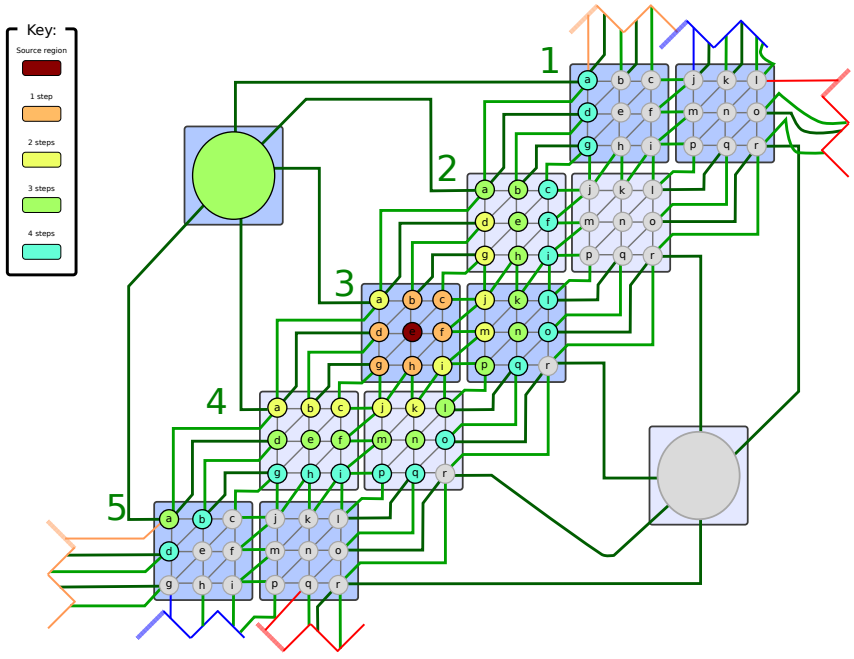


Figure 6.3: Breadth-first search expanding from point 3e in a small icosahedral grid.

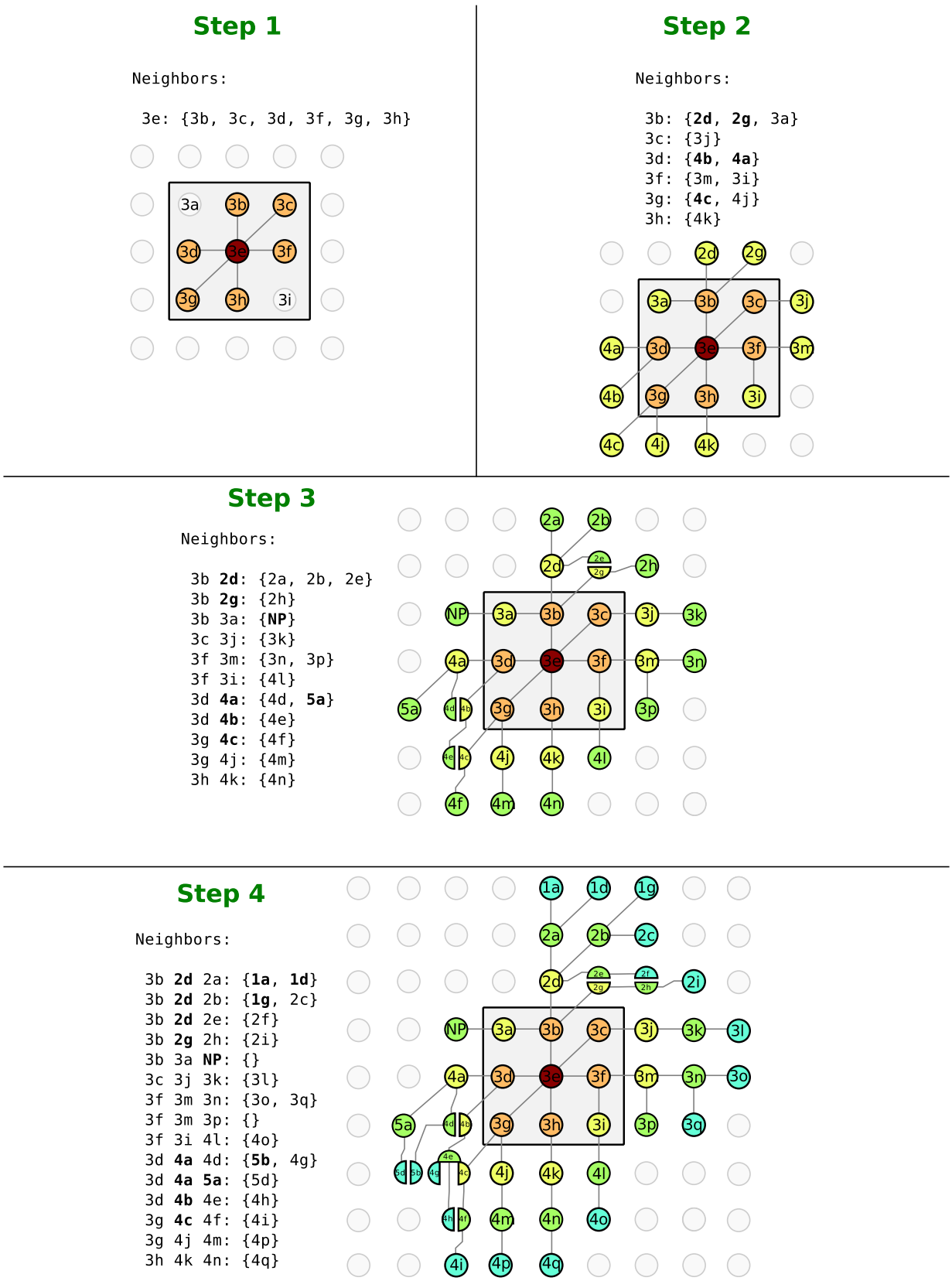


Figure 6.4: Expanded halo generation for point 3e using naive algorithm.

```
routine naive_halo_solver:
  INPUT: γ, θ, d, ⟨i,j⟩ ! Graph, block, halo depth, and initial point
  OUTPUT: m    ! A function (mapping) between indices in i(θ) and i(γ)

let B be the border maps for γ

! We store frontier nodes for the current and next breadth first search
! iterations in the following initialized lists. The _blkIdx lists store
! indices from i(θ) the _glbIdx lists store corresponding indices from i(γ)
let frontier_blkIdx, frontier_glblIdx be empty lists
let nextFrontier_blkIdx, nextFrontier_glblIdx be empty lists

! We store indices of visited nodes from i(γ) in this list:
let visited be an empty list

! Set the initial frontier
frontier_blkIdx.add(⟨i,j⟩); frontier_glbIdx.add((θ_s, ⟨i + θ_{u_i} − 1, j + θ_{u_j} − 1⟩))

! Repeatedly do BFS until no nodes are left on the frontier
while(frontier_blkIdx is not empty):
  set nextFrontier_blkIdx and nextFrontier_glblIdx to be empty

  ! Perform BFS algorithm:
  for each node (n_blk, n_gbl) in lists (frontier_lcl, frontier_gbl):
      let ⟨n_blkIdx_i, n_blkIdx_j⟩ = n_blk
      let (n_s, ⟨n_glblIdx_i, n_glblIdx_j⟩) = n_gbl

      for each neighbor neigh of n_gbl not in visited:
        visited.add(neigh)

        ! Determine where to place the neighbor locally
        let (neigh_s, ⟨neigh_i_glb, neigh_j_glb⟩) = neigh
        let neigh_blkIdx_i = n_blkIdx_i + (neigh_glblIdx_i − n_glblIdx_i)
        let neigh_blkIdx_j = n_blkIdx_j + (neigh_glblIdx_j − n_glblIdx_j)

        ! If the element is within the halo, map the neighbor and add
        ! it to the frontier
        if(⟨neigh_blkIdx_i, neigh_blkIdx_j⟩ ∈ (i(θ) ∪ h(θ,d))):
          let m(γ,θ,⟨neigh_blkIdx_i, neigh_blkIdx_j⟩) =
            (neigh_s, ⟨neigh_glblIdx_i, neigh_glblIdx_j⟩)
          nextFrontier_blkIdx.add(⟨neigh_blkIdx_i, neigh_blkIdx_j⟩)
          nextFrontier_glblIdx.add(neigh)

  ! Set frontier for next iteration
  frontier_blkIdx = nextFrontier_blkIdx
  frontier_glbIdx = nextFrontier_glbIdx
```

Figure 6.5: Psuedocode for naive halo-generation algorithm.

6.3 Expansion with rotation

The example in Figure 6.4 illustrates a result where a single halo location maps to multiple grid locations. This occurs because as we perform the breadth-first search on the grid and as we move across subgrids edges rotate but we do not perform the corresponding rotation in the halo. Consequently, elements that form along a straight line topologically do not fall in a straight line within the halo; for example compare the elements in the path $[3e, 3b, 2d, 2b, 1g]$ in Figure 6.1, where it forms a straight path across different subgrids, and in 6.4 where the nodes do are not placed along a straight line.

To address this we introduce *neighborhoods* and *rotations*. Neighborhoods specify how nodes within a subgrid connect and rotations are used to determine where to place a node in a block's halo. Formally we define a neighborhood as a list of neighbors and a neighbor ν as a tuple:

$$\nu = \langle i, j \rangle \tag{6.8}$$

where $-1 \leq i, j \leq 1 \vee i, j \in \mathbb{Z}$, and we define a neighborhood N to be a list of neighbors ordered in a clockwise fashion. That is N satisfies the equation:

$$N \subseteq [\langle 0, 1 \rangle, \langle 1, 1 \rangle, \langle 1, 0 \rangle, \langle 1, -1 \rangle, \langle 0, -1 \rangle, \langle -1, -1 \rangle, \langle -1, 0 \rangle, \langle -1, 1 \rangle]. \tag{6.9}$$

We also require that for any neighbor $\nu \in N$ that its opposite also be in the list; that is:

$$\langle i, j \rangle \in N \Rightarrow \langle -i, -j \rangle \in N \tag{6.10}$$

We define a rotation function r

$$r(N, n, \langle i, j \rangle) = N[mod(x + 1, |N|)] \mid \exists x : N[x] = \langle i, j \rangle. \tag{6.11}$$

We give the function r for the mini-icosahedral grid in Figure 6.6.

We also update our abstraction for border mappings to be passed in a rotation amount. We define this extended definition of a border map to be:

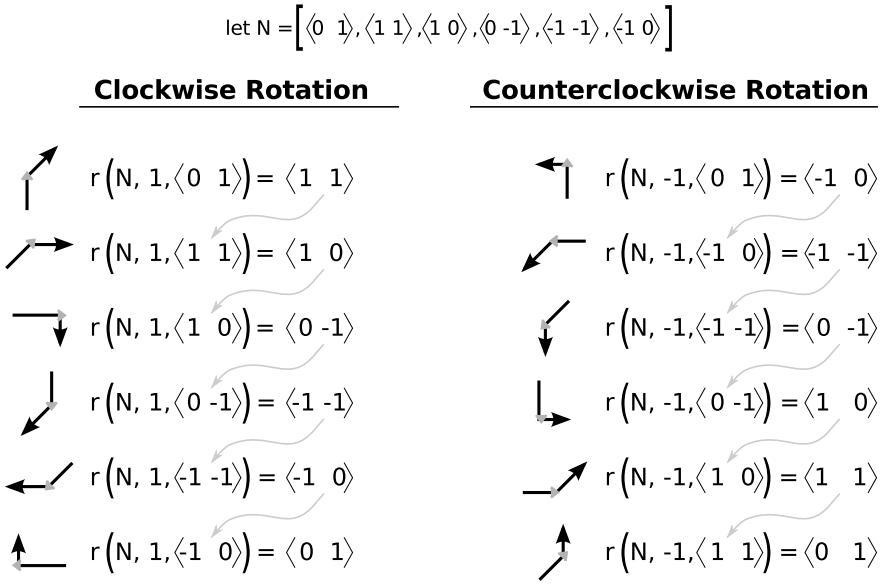


Figure 6.6: Rotation function r for neighborhood for six-point mini-icosahedral grid.

$$\beta = (\rho_{tgt}, \rho_{src}, n)\,. \tag{6.12}$$

Using the rotation function we can update the halo algorithm to track rotation. Specifically, we introduce an array `frontier_rotation` that keeps track of rotation for nodes on the frontier. We change lines 38 - 41 from Figure 6.5 to the following:

```
! Determine where to place the neighbor locally
 let (neigh_s,⟨neigh_i_glb,neigh_j_glb⟩) = neigh
 let (neigh_blkIdx_i, neigh_blkIdx_j) =
   r(N, n,
     (n_blkIdx_i + (neigh_gblIdx_i - n_gblIdx_i))
     (n_blkIdx_j + (neigh_gblIdx_j - n_gblIdx_j))
```

We illustrate an example of this algorithm in Figure 6.7. Step 1 occurs the same for the algorithm with or without rotation (as illustrated in Figure 6.4); however, in step 2 when adding nodes 2d, 2g 4a, 4b, and 4c we recall that a rotation occurred in the global domain. We mark this in Figure 6.7 by placing an arrow next to the node. When expanding out from these nodes in step 3 we must apply the rotation function r. For example, when expanding from node 2d to node 2b we calculate $r(N, -1, \langle 1, 1\rangle)$, which results in $\langle 0, 1\rangle$, so we place node 2d above node 2b in the block's halo. Similarly, we apply the rotation function to other nodes in step 3 and subsequent steps.

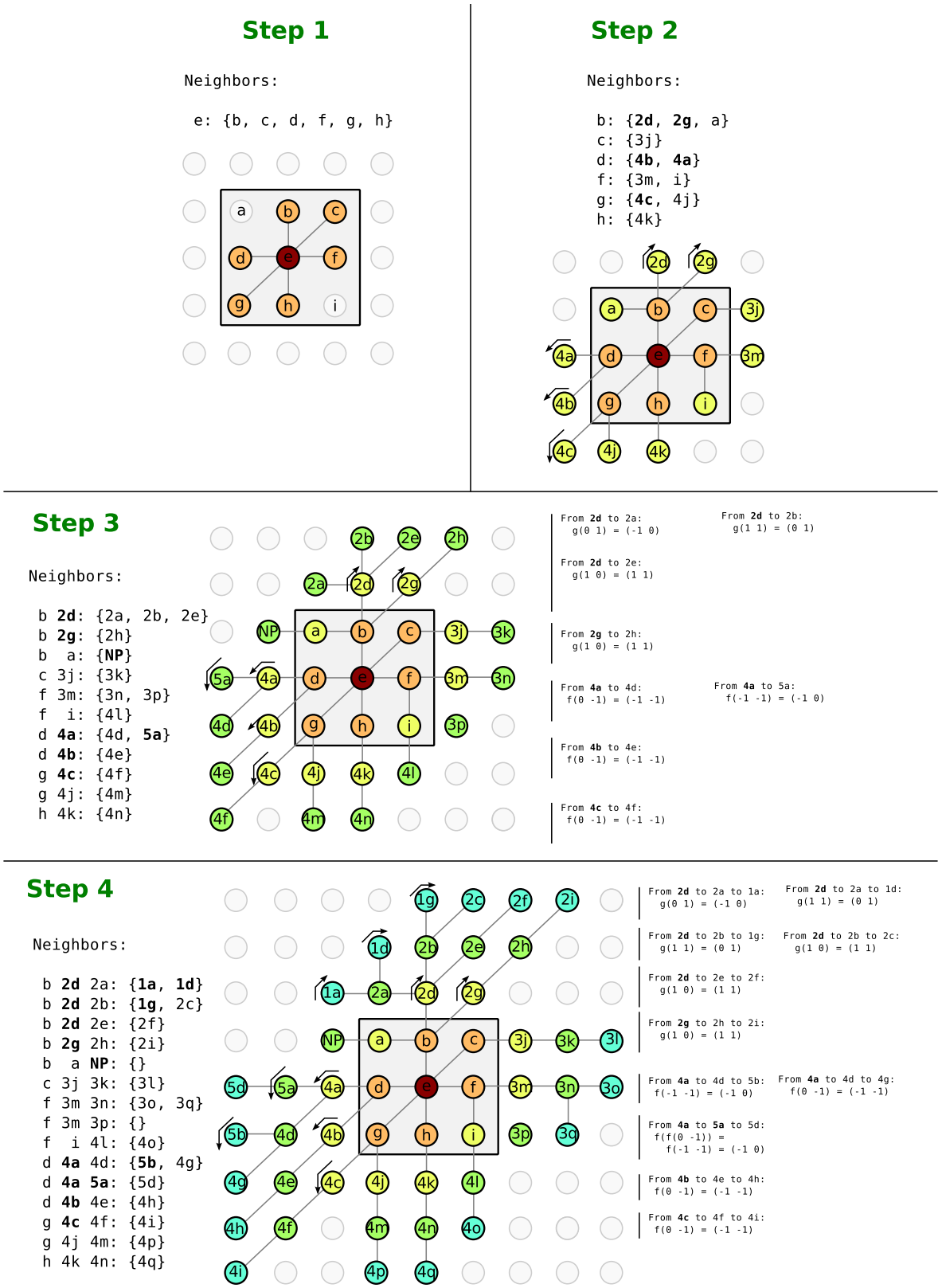


Figure 6.7: Expanded halo generation for point 3e using algorithm with rotation.

6.4 Ghost cells

The halo-generation algorithm from Section 6.3 produces a correct halo for a block when applying a stencil for a given point $\langle i, j \rangle$, however the result depends on the values of i and j. This indicates that it is not possible for a stencil computation to update all nodes in a block with a single halo.

To address this we take an alternative method of storing remote-data based on a data structure we call a ghost-cell list. Ghost cell lists are one dimensional arrays that list global indices (that is elements in $I(\gamma)$) that may be referenced when updating a block.

We associate a ghost-cell list G with each block; we will update our definition of a block θ from Equation 5.16 to a new definition θ' where:

$$\theta' = (b, p, s, \langle u_i, u_j \rangle, \langle v_i, v_j \rangle, G). \tag{6.13}$$

Prior to performing any stencil computation it is necessary to calculate a communication plan that is cognizant of ghost-cells. We expand our definition of message so that it includes information necessary to transfer ghost-cells. Specifically, we define a message for ghost cells mu' to be:

$$\mu' = (p, G). \tag{6.14}$$

If mu' is a receiving message, then p is the processor that the message is being received from; if mu' is a sending message, then p is the processor that the message is sent to. In either case G is the list global indices to transfer.

We define a communication plan for ghost cells $\pi'(p)$ to be

$$\pi'(p) = (\mu'_r, \mu'_s), \tag{6.15}$$

where mu'_r is a set of receiving messages for processor p and mu'_s is a set of sending messages for processor p.

```
for each block θ′ in D that is not an interior block:
  let p be the processor that owns θ′
  G = find_nodes_to_ghost(θ′, d)
  π′ = π′ ∪ generate_ghostcell_communication_plan(G, p)
```

Figure 6.8: Code to generate communication plan for ghost cells.

To do this we execute the code in Figure 6.8 where the `find_nodes_to_ghost` function returns a list of ghost nodes accessible from block θ' within d steps; it will perform breadth first search of the graph starting with a frontier that is all the border nodes in block θ'. The `find_nodes_to_ghost` routine will produce the set of send and receive messages necessary for process p to receive the ghost nodes in G.

6.5 Data access in non-compact stencils

GridWeaver includes two ways of accessing ghost-cell values: relative coordinates and absolute coordinates. Using relative coordinates, nodes are referenced using two-dimensional (x, y) coordinates within the current subgrid. In both compact and non-compact stencils this is the default access method. For a stencil of depth d updating a node $(\langle i, j \rangle, \text{sg})$, users may access any value in the domain $\{(x, y), midi - d \leq x \leq i + d \ \wedge \ j - d \leq y \leq j + d\}$. If the point (x,y) falls off the subgrid than GridWeaver uses the search-with-rotation algorithm described in Section 6.3.

Using absolute coordinates, nodes are referenced using three-dimensional $(\langle i, j \rangle, \text{sg})$ coordinates. If the point (x, y) falls outside of the block than GridWeaver searches through ghost-cells to find the one corresponding to the absolute coordinate. To use absolute coordinates, users pass the data object through the `indexabs` function.

6.6 Performance impacts

The cost of conducting communication for a stencil of depth $n + 1$ will always be the same, if not more costly, than for a stencil of depth n. Depending on the connectivity of the grid, the additional number of communicated values will, at worst, grow geometrically.

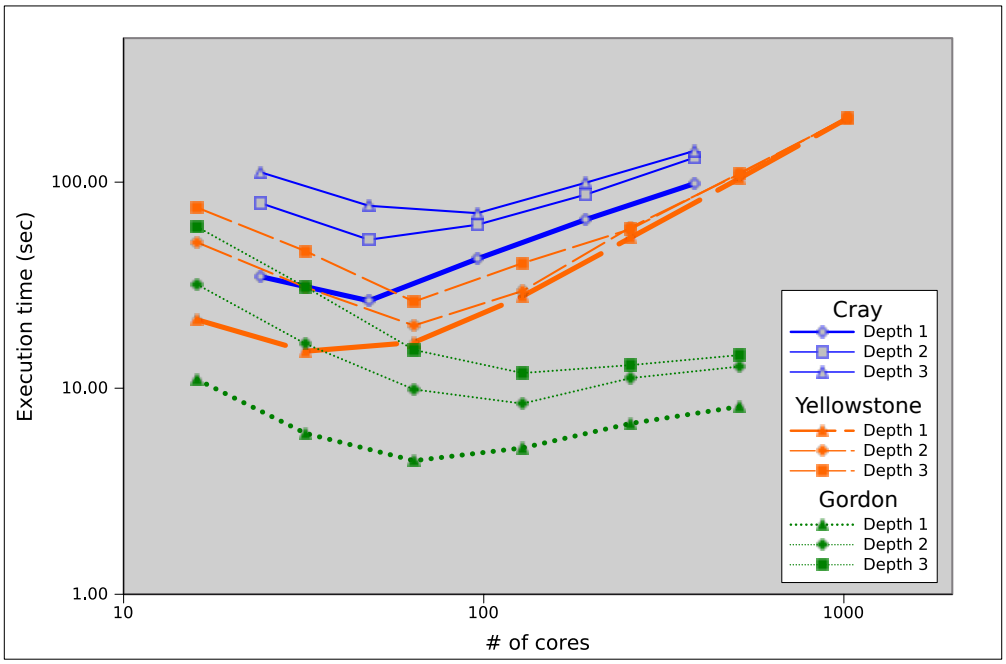
```
! Call to stencil function:
call data_apply_noncompact(x, x, indexabs(x))

! Stencil function:
stencil(test_stencil, x, x_rel, x_abs, i, j, sg)
  test_stencil = X(i, j+1) * abs(i, j+1, sg)
end function
```

Figure 6.9: Code for a simple, non-compact, stencil.

In Figure 6.10 we plot the scalability of conducting communication calls to populate halos and ghost cells for stencils of depth 1, 2, and 3. We present results on three machines: (1) the ISTeC Cray [16], a 1,248 core Cray XT6m located at Colorado State University (CSU); (2) Yellowstone [27], a 72,288 core IBM iDataPlex machine located at the National Center for Atmospheric Research (NCAR); and (3) Gordon [12], a 16,384 core Appro GreenBlade machine located at the San Diego Supercomputer Center (SDSC). We present technical information about these three machines in Table 8.1. We run experiments for our case studies in Chapter 8 on these same machines.

Each of the experiments in Figure 6.10 conduct 10,000 communication calls on an icosahedral grid. We set the size of the ten non-polar subgrids contain 1000x1000 nodes and be partitioned into blocks of 50x50. We partitioned data in a block-cyclic manner.

At worst communication for a depth 2 stencil, when compared to a depth 1 stencil is 2.89 times slower and at best is 1.01 times slower. At worst communication for a depth 3 stencil, when compared to a depth 1 stencil, is 5.51 times slower and at best is 1.01 times slower. The relative cost between depth 1 and larger-depth stencils typically lowers when more cores are used. All experiments running on 128 or more cores ran at worst 2.31 times slower and at best 1.01 times slower.



Cray

	depth (s)			ratio	
cores	1	2	3	2:1	3:1
24	34.82	79.13	111.74	2.27	3.21
48	26.56	52.57	76.80	1.98	2.89
92	42.45	62.15	70.71	1.46	1.67
184	65.84	87.03	99.37	1.32	1.51
368	98.31	131.40	141.75	1.34	1.44

	Yellowstone					Gordon				
	depth (s)			ratio		depth (s)			ratio	
cores	1	2	3	2:1	3:1	1	2	3	2:1	3:1
16	21.5	51.01	75.28	2.37	3.50	11.02	31.86	60.68	2.89	5.51
32	15.12	30.91	46.27	2.04	3.06	6.04	16.42	31.01	2.72	5.13
64	16.68	20.15	26.32	1.21	1.58	4.45	9.87	15.35	2.22	3.45
128	27.92	29.63	40.43	1.06	1.45	5.13	8.42	11.85	1.64	2.31
256	53.82	60.4	59.22	1.12	1.10	6.75	11.2	12.95	1.66	1.92
512	104.1	109.33	110.36	1.05	1.06	8.14	12.74	14.48	1.57	1.78
1024	204.2	206.45	205.73	1.01	1.01					

Figure 6.10: Costs for populating halos of various depths.

Chapter 7

Performance optimizations

We have developed a library called GridWeaver, which implements the abstractions introduced in Chapters 5 and 6. The GridWeaver library enables a separate specification of semiregular-grid connectivity 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 it introduces library overhead. In Section 7.1 we discuss this overhead and describe how the GridWeaver code rewriting tool inlines code to remove library overhead. In Section 7.2 we discuss a communication avoiding optimization.

7.1 Inlining stencil code

A disadvantage of a library-based approach is that libraries necessarily introduce overhead. In GridWeaver this overhead occurs because it calls a stencil function for every point in the grid individually. Furthermore, the stencil function calls another function whenever accessing the point's value or the value of any of its neighbors.

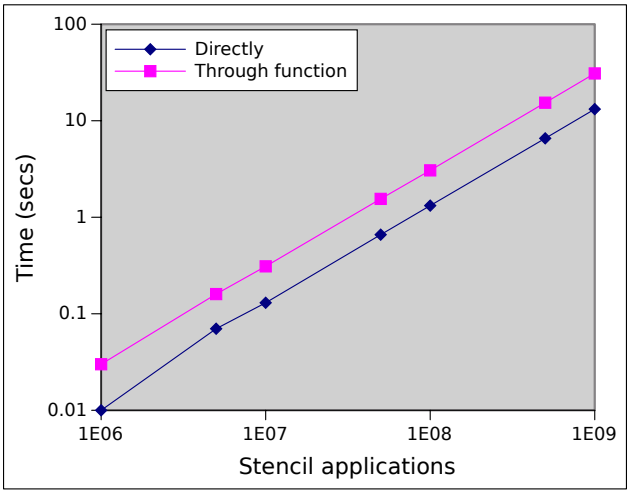
For example, GridWeaver calls the `fivePtAvgStencil` function once for every point in the grid, and, although it syntactically appears like an array, the symbol `A` is actually a

```
1  ! Apply stencil function
2  call data_apply(data_out, data_in, fivePt)
3
4  ! ...
5  contains
6    stencil(fivePt, in, i, j)
7        fivePt = 0.2 * (in(i, j)   + in(i-1, j)  + in(i + 1, j) +
8                        in(i, j-1) + in(i,   j+1))
9    end function
```

Figure 7.1: Code for a five-point stencil function in GridWeaver.

```
1  ! Apply stencil function
2  do gw____lbid=lbound(data_out%vals,3), ubound(data_out%vals,3)
3    ! Iterate over all points <blkI, blkJ> within the block
4    do gw____blkJ=1,gw____blkH
5      do gw____blkI=1,gw____blkW
6        ! Perform the stencil operation
7        data_out%vals(gw____blkI, gw____blkJ, gw____lbid) =
8          0.2 * (data_in%vals(gw___blkI, gw___blkJ, gw____lbid) +
9                 data_in%vals(gw___blkI - 1, gw___blkJ, gw____lbid) +
10                data_in%vals(gw___blkI + 1, gw___blkJ, gw____lbid) +
11                data_in%vals(gw___blkI, gw___blkJ - 1, gw____lbid) +
12                data_in%vals(gw___blkI, gw___blkJ + 1, gw____lbid))
13     end do
14   end do
15 end do
```

Figure 7.2: Inlined code for a five-point stencil produced by GridWeaver's source-to-source translation tool.



calls (1000s)	direct (s)	indirect (s)	(times slower)
1000	0.01	0.03	3.00
5000	0.07	0.16	2.29
10000	0.13	0.31	2.38
50000	0.66	1.55	2.35
100000	1.32	3.06	2.32
500000	6.57	15.38	2.34
1000000	13.19	30.9	2.34

Figure 7.3: Log-log plot of time to execute a three point stencil on one-dimensional data using direct access versus access through a function. We list the code used for this experiment in Figure C.5 of Appendix C.

function that returns the value associated with some grid point from data structures internal to GridWeaver.

To address this overhead we have developed a source-to-source code translation tool that preprocesses programs written with GridWeaver and inlines GridWeaver calls. To parse and translate Fortran programs we use the Rose source-to-source translation framework [122]. The name GridWeaver comes from both the notion of tying several subgrids together to form a single grid and the idea of weaving the GridWeaver library code into a user's program. This weaving done by GridWeaver is analogous to the weaving process in Aspect Oriented Programming [99] 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 GridWeaver 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 GridWeaver by its user but is invoked internally. In Figure 7.1 we list code for a stencil function for a five-point operation; in Figure 7.2 we present what a call to this function looks like after it has been inlined.

7.2 Communication avoiding

As we discuss in Section 8.1.1 GridWeaver performs computation in an owner-computes fashion. Because a stencil operation requires values from neighboring nodes, we conduct communication to populate values in halos (as discussed in Section 5.2.2) and ghost lists (as discussed in Section 6.4). As shown in Figure 7.4, if a stencil computation both reads from and writes to the same data object then the call to update halos for the data object must be made every iteration. Depending on a machine's network, and how data is distributed, this communication may be costly.

```
call schedule_calculate(sched, dist, 1)
x = data_new(sched)

do n=1,nTimesteps
  call data_forceUpdate(x)
  call data_apply(x, x, stencilFunc)
end do
```

Figure 7.4: GridWeaver code that applies a Jacobi-style stencil for a set number of iterations.

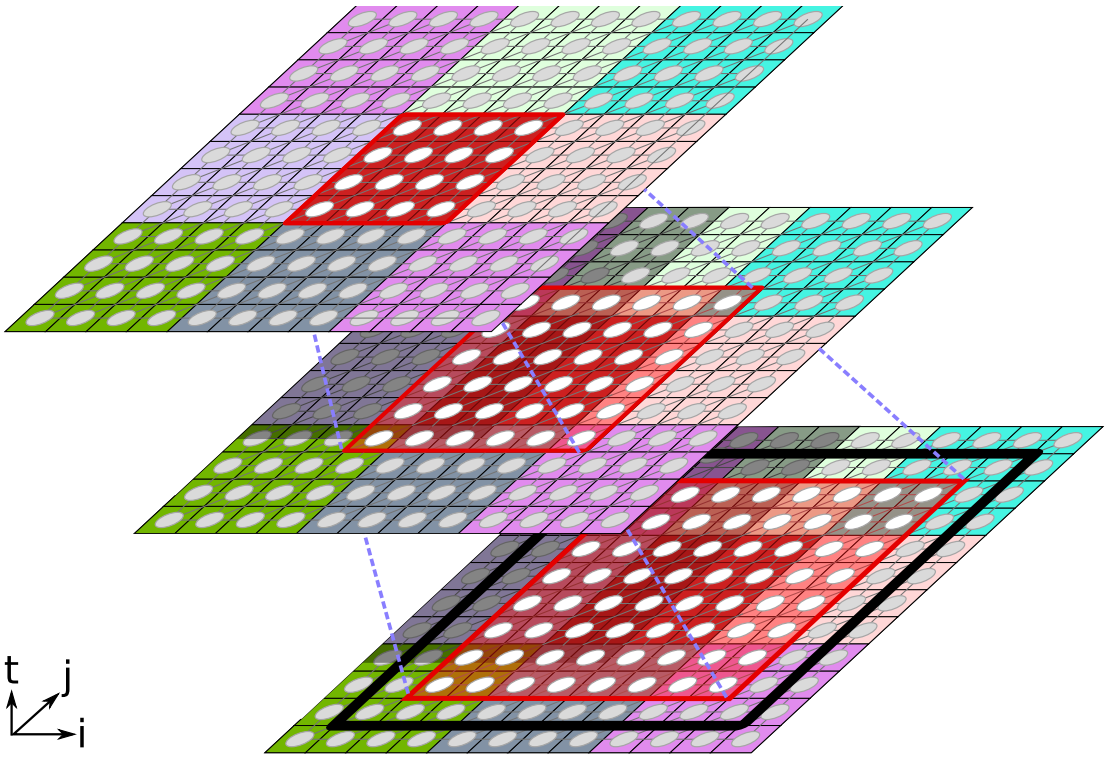


Figure 7.5: Overlapped tiles for center (red) block for three time iterations. In each timestep the processor that owns the red block computes the red-shaded region. On the first timestep a single communication call populates the halo (represented using the rectangle drawn with thick, black, strokes) on the first timestep.

```
call schedule_calculate(sched, dist, 1+nOverlaps)
x = data_new(sched)

do n=1,nTimesteps
  do i=nOverlaps,1,-1
      call data_forceUpdate(x)
      call data_apply(x, x, stencilFunc, i)
    end do
end do
```

Figure 7.6: Applying stencil computation with overlapping tiles.

One way to avoid communication is for a block is to perform extra computation in place of conducting communication. For example, in a compact stencil, nodes along a block's border will access values from nodes in the block's halo. Between time steps, values for a block's halo nodes must be updated. We can avoid communication by performing extra computation that updates these values rather than by passing messages to retrieve the updated values from another processor.

To update the values that surround a block's local data we must expand the halo an additional layer out. With this strategy we can avoid conducting communication for a single time step. By expanding halos out further we can avoid communication for additional time steps. We illustrate this concept in Figure 7.5. In this figure the processor that owns the red block will compute the red-shaded region for each of the three illustrated time steps. On the first time step a single communication call populates the halo around the red block; between time steps two and three no communication is necessary.

We can modify the `data_apply` routine so that it updates all values within its n-depth halo. By updating n-layers of extra values we can avoid conducting communication for n time steps. In Figure 7.6 we list example code.

Chapter 8

CGPOP and SWM case studies

In this chapter we present case studies that use GridWeaver to implement CGPOP's compact stencil operation (Section 8.1), change from using dipole to tripole grid connectivity (Section 8.1.3), and implement the compact and non-compact stencils used in SWM's horizontal advection operation (Section 8.2). These case studies demonstrate that GridWeaver is able to express computations seen in real-world codes without incurring a large performance penalty.

In our case studies we perform experiments on three distributed-memory clusters: (1) the ISTeC Cray [16], a 1,248 core, 52 node, Cray XT6m located at Colorado State University (CSU); (2) Yellowstone [27], a 72,288 core, 4518 node, IBM iDataPlex machine located at the National Center for Atmospheric Research (NCAR); and (3) Gordon [12], a 16,384 core, 1024 node , Appro GreenBlade machine located at the San Diego Supercomputer Center (SDSC). We list technical information about these platforms in Table 8.1. Our experiments on the ISTeC Cray and Yellowstone were compiled with version 4.6.3 of the GNU compiler (GCC), and on Gordon were compiled with version 12.1.0 of the Intel Compiler (ICC). We compiled all codes with default compiler optimizations enabled.

8.1 Using GridWeaver in CGPOP

The CGPOP miniapp [135] is a proxy application of version 1.4.3 of the Parallel Ocean Program (POP) [94]. In [137], we modified the original Fortran implementation of CGPOP to use coarrays in order to examine the performance and programmability impacts of using the Coarray Fortran programming model. In this section we take the approach used in [137] to evaluate the GridWeaver programming model. Specifically, we measure the impact GridWeaver has in terms of performance, how it affects code size, and provide code listings

Table 8.1: Description of compute platforms used for GridWeaver case studies.

System			
Name	ISTeC Cray [16]	Yellowstone [27]	Gordon [12]
Company	Cray	IBM	Appro
System Type	XT6m	iDataPlex	GreenBlade
# of cores	1,248	72,288	16,384
Processor			
CPU	Opteron	Sandy Bridge EP	Xeon E5
Mhz	1900	2600	2600
Peak Gflops/core	9.6	20.8	20.8
cores/node	12	16	16
Memory Hierarchy			
L1 data-cache	128 KB	32 KB	32 KB
L2 cache	512 KB	256 KB	256 KB
L3 cache	12 MB	20 MB	15 MB
Network			
topology	2D torus	Full fat tree	3D torus
# of Links/per node	4	varies	6
Bandwidth/link	6.6 GB/s	13.6 GB/s	8 GB/s

that demonstrate how GridWeaver enables a separate specification of grid-connectivity, data distribution, and stencil computation.

Our implementation of CGPOP in GridWeaver contains 11 lines of code to model grid-connectivity, 236 lines of code to conduct CGPOP's Conjugate Gradient (CG) algorithm, and 41 lines of code to move data stored in CGPOP's original data structures into GridWeaver's data structures. GridWeaver eliminates the need for CGPOP's 1226 lines of communication code and 180 lines of existing stencil code. We net a reduction of 979 lines of code. As a metric SLOCcount can be problematic: not all lines of code reflect an equal amount of work. More important is the fact that our approach enables a separation of concerns and removes the need to explicitly write communication code.

In Section 8.1.1 we detail how to transfer data from CGPOP's original data structures into GridWeaver. We also discuss how to ensure that data in GridWeaver is distributed in the same manner as it is in CGPOP. In Section 8.1.2 we discuss the code we added to conduct CGPOP's Conjugate Gradient algorithm, and in Section 8.1.3 we discuss how to modify our GridWeaver based implementation of CGPOP to operate on a tripole grid. In Section 8.1.4 we summarize the performance and programmability impact of using GridWeaver for both

dipole and tripole implementations of CGPOP. We conduct our experiments on the platforms listed in Table 8.1.

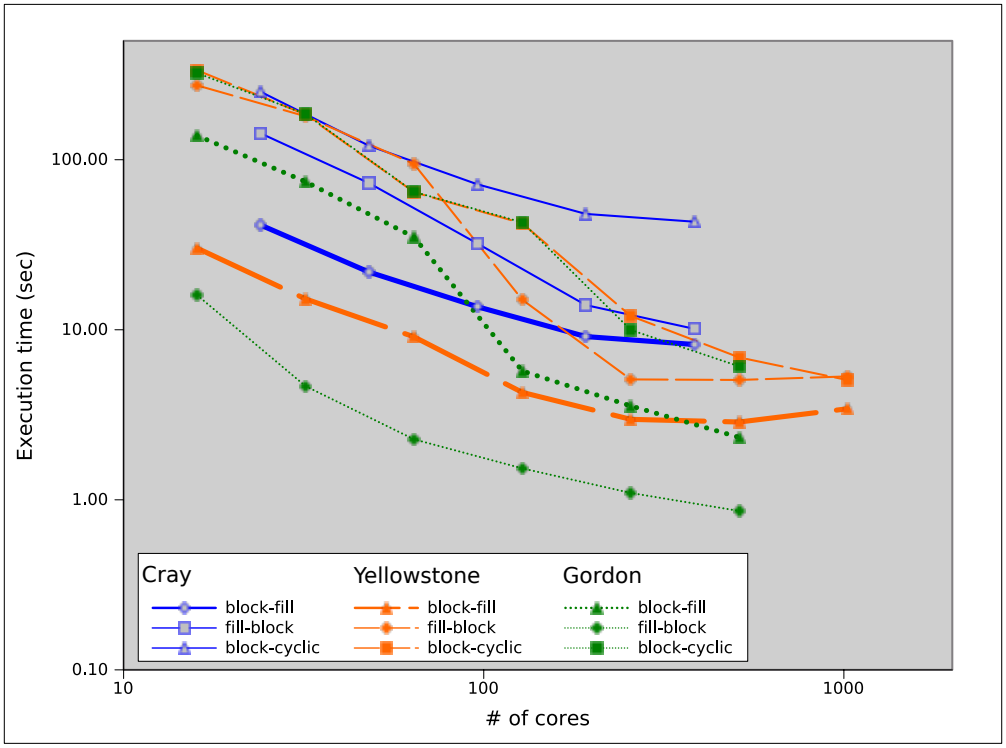
8.1.1 Decomposition and data

Since we want to evaluate GridWeaver by comparing the performance of a GridWeaver based implementation of CGPOP against the performance of a MPI+Fortran implementation, it is important that we control for factors that influence performance. How data and computation are distributed is one such factor.

CGPOP and GridWeaver partition grid data into rectangular regions called blocks. In a distributed-memory message-passing model, each processing element has direct access to a locally-accessible pool of memory. Both CGPOP and GridWeaver assign ownership of blocks to processors, who then store their block data in their respective memory pools.

Both CGPOP and GridWeaver use an *owner computes* [44] style of computation distribution. In an owner-computes style of computation distribution a processing element is responsible for updating the same blocks that it stores. In GridWeaver, both computation and data distribution is specified via a decomposition abstraction, which we describe in greater detail in Section 5.1.2.

How an application distributes its data and computation can significantly impact its performance. For example, when updating halo values it may be necessary for a processing element that owns one block to communicate with a processing element that owns an adjacently located block. If two adjacent blocks are stored on the same processing element, then communication between the blocks will occur by copying data directly from one block into the other block's halo; however, if two adjacent blocks are stored on different processing elements, then communication occurs over a network. Thus by minimizing the number of adjacent blocks that are assigned on different processing elements, we can minimize network communication. Furthermore, the latency cost of a message between two processing elements can vary depending on the relative location of the two elements. For example, two processing



Cray

cores	fill-blk (s)	blk-fill (s)	blkcyc (s)
24	41.32	143.00	251.00
48	21.85	72.88	120.90
92	13.66	32.17	71.70
184	9.12	13.97	48.00
368	8.19	10.13	43.20

	Yellowstone			Gordon		
cores	fill-blk (s)	blk-fill (s)	blkcyc (s)	fill-blk (s)	blk-fill (s)	blkcyc (s)
16	30.19	273.65	335.96	15.99	138.57	323.16
32	15.19	180.05	184.28	4.66	74.31	185.72
64	9.09	94.43	64.47	2.27	35.08	64.77
128	4.28	15.01	42.21	1.53	5.72	42.79
256	2.97	5.10	12.05	1.10	3.57	9.97
512	2.87	5.06	6.88	0.86	2.33	6.09
1024	3.43	5.31	5.06			

Figure 8.1: Execution time for 10,000 communication calls using three different distributions (filled rows with blocked columns, blocked rows with filled columns, and block cyclic).

elements located on the same cluster node can send and receive messages quicker than two processing elements located on different nodes.

We demonstrate this point in Figure 8.1, where we plot the execution time to conduct 10,000 calls to populate single-element depth halos for a 5000x5000 node, dipole grid. We assign grid nodes random, floating point values and decompose data into 25 by 25 element blocks. We implement the computation in GridWeaver. Note that when data is distributed in a block-fill manner total execution time is between 15.95 and 1.24 times slower than when distributed in a fill-block fashion, and when data is distributed in a block-cyclic manner total execution time is between 20.21 and 1.47 times slower than when distributed in a fill-block fashion.

Furthermore, if the number of blocks assigned to any processing element exceeds the amount assigned to any other, or if the amount of computation required to update one block exceeds the amount of another, then the workload between processors will vary. When the workload betweens processors varies significantly, then an application is said to exhibit poor *load balance*.

In order to control for the effects that communication and load balance have on performance, we have our GridWeaver implementation of CGPOP use the same distribution as the original Fortran+MPI code. By doing so, we also demonstrate that GridWeaver is flexible enough to enable distributions seen in real-world applications.

To minimize communication and perfect load balance, POP and CGPOP use a space-filling curve distribution. Since CGPOP is an ocean simulator, only values corresponding to regions of ocean need to be computed on and stored. To enable this behavior in GridWeaver, we allow users to specify that a block is distributed to a null processor (represented as MPI rank `-1`).

In Figure 8.2 we present code that imports CGPOP's data distribution into GridWeaver. This code starts by instantiating a blank distribution object with the `distribution_applyBlankDist` function (line 1), iterating through CGPOP's blocks (line 4), and passing

```
1  call distribution_applyBlankDist(dist, g, blkW, blkH)
2
3  ! Iterate through blocks
4  do i=1,nblocks_tot
5    ! Get block metadata from CGPOP
6    call get_block_parameter(i,
7      cgpop_lbid, ib, ie, jb, je, iblock, jblock, npoints)
8
9    ! If the block contains any ocean points then tell GridWeaver
10   ! GridWeaver to use the same processor for the block as CGPOP
11   if(npoints > 0) then
12     gw_gbid = distribution_gbidAt(dist, sg,
13       (iblock-1) * blkW + 1, (jblock-1) * blkH + 1)
14     gw_proc = distrb_tropic%proc(all_blocks(i)%block_id) - 1
15     call distribution_setProcForBlock(dist, gw_gbid, gw_proc)
16   end if
17 end do
```

Figure 8.2: Code to import CGPOP's space-filling curve distribution into GridWeaver. The `distribution_gbidAt` function and `distribution_setProcForBlock` routine are part of the GridWeaver library.

metadata about ocean-containing blocks into GridWeaver (lines 5 through 15). Line 11 determines if a given block in CGPOP contains ocean points.

More specifically, in lines 6 and 7 of Figure 8.2 we call the CGPOP function `get_block_parameter`, which returns metadata about a block. The values `iblock` and `jblock` represent a block's Cartesian block coordinates. In line 13 we multiply these values to get the position of the first node in the block, and in line lines 12 through 13 we pass this position to GridWeaver's `distribution_gbidAt` function to return the index GridWeaver uses to refer to the block.

After we define how to distribute data, we can assign CGPOP's input values into GridWeaver data objects (we describe CGPOP's input files in Section 2.3.2). There are two main approaches we can take to do this: one is to modify CGPOP's existing input routines to place data directly into GridWeaver data objects, and the second is to maintain the existing input routines and include code that transfers data from the original datastructures into GridWeaver. Since we maintain the distribution used in CGPOP, GridWeaver's data objects and CGPOP's original arrays are aligned, and we can transfer data from one to the other by simplify iterating over the datastructures lock-step and copying values.

```
1  ! Iterate through CGPOP blocks; find the associated GW block; copy data
2  do cgpop_gbid=1,nblocks_tot
3    call get_block_parameter(cgpop_gbid,
4      cgpop_lbid, ib, ie, jb, je, iblock, jblock, npoints)
5    gw_proc = distrb_tropic%proc(all_blocks(cgpop_gbid)%block_id) - 1
6
7    if(gw_proc == my_task) then
8      cgpop_lbid = distrb_tropic%local_block(cgpop_gbid)
9      gw_gbid = distribution_gbidAt(dist, sg,
10                (iblock-1) * blkW + 1, (jblock-1) * blkH + 1)
11     gw_lbid = distribution_gbid2lbid(dist, gw_gbid)
12
13     ib = all_blocks(cgpop_gbid)%ib-1; jb = all_blocks(cgpop_gbid)%jb-1
14     do j=1,blkH; do i=1,blkW
15       data_RHS%vals(i,j,gw_lbid) =  RHS(i+ib,j+jb,cgpop_lbid)
16       ! ... copy additional arrays ...
17     end do; end do
18   end if
19 end do
```

Figure 8.3: Code to translate data in CGPOP's into a GridWeaver data object. Specifically, we translate the array RHS. In RHS, CGPOP stores the values for b from the linear equation being solved (Ax=b.).

Since one of our goals is to minimize the amount of change existing programs require to utilize GridWeaver, and since CGPOP already reads in data and stores it into subdomains, we take the second approach of maintaining CGPOP's existing input and initialization routines and transfer data into GridWeaver. Although this second approach may lead to a larger total lines-of-code count, it does require modifying existing lines of code and has the advantage of keeping CGPOP's existing datastructures intact. Keeping these datastructures intact may be useful if there are parts of the application, which we do not wish to modify, that leverage the existing datastructures.

In Figure 8.3 we list code that transfers data from CGPOP's original grid data structures into GridWeaver's data objects. Line 2 iterates through each block of data in CGPOP's original data structures, and lines 3 through 17 copy the block's data into CGPOP when applicable. Specifically, line 7 determines whether the current processing element should copy the block's data data; lines 8-11 determine what block in GridWeaver corresponds to the block in CGPOP; lines 13-14 iterate through the blocks values; and line 15 copies from the CGPOP block into the GridWeaver block. The code in Figure 8.3 could be modified to

```
1  call get_block_parameter(gid,ib=ib,ie=ie,jb=jb,je=je)
2  do j=jb,je
3  do i=ib,ie
4    AX(i,j,bid) = A0 (i  ,j  ,bid)*X(i  ,j  ,bid) +
5      AN (i  ,j,bid) * X(i  ,j+1,bid) + AN (i  ,j-1,bid) * X(i  ,j-1,bid) +
6      AE (i  ,j,bid) * X(i+1,j  ,bid) + AE (i-1,j  ,bid) * X(i-1,j  ,bid) +
7      ANE(i  ,j,bid) * X(i+1,j+1,bid) + ANE(i  ,j-1,bid) * X(i+1,j-1,bid) +
8      ANE(i-1,j,bid) * X(i-1,j+1,bid) + ANE(i-1,j-1,bid) * X(i-1,j-1,bid)
9  end do
10 end do
```

Figure 8.4: Original baratropic operator from CGPOP; bid is a block ID.

transfer data from GridWeaver's data objects back into CGPOP's original data structures by flipping the left- and right-hand sides of the assignment operation in line 15.

8.1.2 Implementing CGPOP operations

Implementing CGPOP in GridWeaver requires rewriting its conjugate gradient operation (specified in Figure 2.4) so that it operates on data objects. This operation consists of a series of reduction and simple arithmetic computations, and a kernel computation: a nine-point barotropic stencil. In GridWeaver we are able to reimplement this stencil without requiring a large code rewrite.

In Figure 8.4 we present CGPOP's original stencil code and in Figure 8.5 we present the GridWeaver-based implementation. Note the GridWeaver code is simply a copy-and-paste of the CGPOP code where we remove the dimension corresponding to blocks from CGPOP's arrays. In both figures, the arrays `AN`, `AE`, and `ANE` respectively store the weights of north/south, east/west, and northeast/southwest edges. In GridWeaver, we use CGPOP's existing input routines to read in these weights and then include code to transfer values into GridWeaver (discussed in Section 8.1.1.

8.1.3 Transition from dipole to tripole grid

The original Fortran+MPI implementation of CGPOP models POP 1.4.3, which uses a dipole grid. In version 2.0, POP's developers modified POP to use a dipole grid; in our case study we implement both dipole and tripole versions of CGPOP. We list code to construct a

```
stencil(btrop_operator, X, A0, AN, AE, ANE, i, j)
  btrop_operator =
    A0 (i,   j  ) * X(i,   j  )  +
    AN (i,   j  ) * X(i,   j+1)  +  AN (i,   j-1) * X(i,   j-1)  +
    AE (i,   j  ) * X(i+1, j  )  +  AE (i-1, j  ) * X(i-1, j  )  +
    ANE(i,   j  ) * X(i+1, j+1)  +  ANE(i,   j-1) * X(i+1, j-1)  +
    ANE(i-1, j  ) * X(i-1, j+1)  +  ANE(i-1, j-1) * X(i-1, j-1)
end function
```

Figure 8.5: Gridweaver code for CGPOP's baratropic operator

```
type(Subgrid)      :: sg
type(Grid)         :: g
integer            :: width, height

! get grid width and height from CGPOP
width  = nx_global
height = ny_global

! Construct grid
sg = subgrid_new("sg", width, height)
g  = grid_new("g")
call grid_addSubgrid(g, sg)

! Wrap left and right borders
call grid_addBorder(g,       0, 1,         0, height,  sg,
                         width, 1,     width, height,  sg, 0)
call grid_addBorder(g, width+1, 1,   width+1, height,  sg,
                             1, 1,         1, height,  sg, 0)

! Fold top border
call grid_addBorder(g,     1, height+1,     width/2, height+1, sg,
                       width, height+1,   width/2+1, height+1, sg, 3)
call grid_addBorder(g, width/2+1, height+1,   width, height+1, sg,
                         width/2, height,         1, height+1, sg, 3)
```

Figure 8.6: GridWeaver code for a single subgrid's connectivity in an icosahedral grid

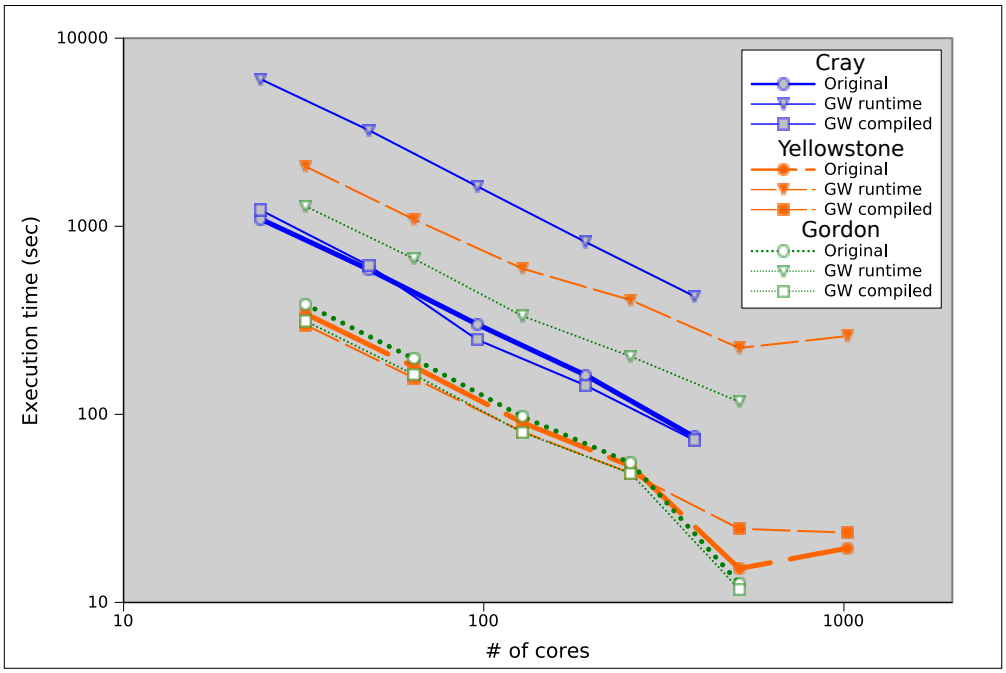
tripole in Figure 8.6. Implementing a tripole grid with GridWeaver is simply a matter of adding code to specify the folded north border. We present this code in lines 20 through 26 of Figure 8.6.

In practice, transitioning from a dipole to tripole grid would also involve updating edge weight values (stored in data objects `AN`, `AE`, and `ANE` in Figure 8.5). Since the focus of this dissertation is to separate the specification of grid connectivity and automate communication, rather than provide a complete push-button system to translate from using one type of grid to another, we did not re-calculate edge weights values from our dipole implementation to our tripole implementation of CGPOP. Existing tools such as the Earth System Modeling Framework [2] are able to solve this re-gridding problem. Since our experiments iterate for a fixed number of time steps, the amount of computation performed is not dependent on edge values, and thus edge weight values do not affect the performance measured in our experiments.

8.1.4 Impact on performance and programmability

CGPOP contains 3,214 lines of code, of which 1226 are devoted to conducting communication to update halo values. By using GridWeaver we able to remove these 1226 lines of code. We spend 11 lines of code modeling the dipole grid, and can model the tripole grid with an additional 2 lines of code. We spend 236 lines of code implementing CGPOP's Conjugate Gradient algorithm.

In Figure 8.7 we present experimental results that compare the performance of the original CGPOP against GridWeaver on the machines listed in Table 8.1. When using the GridWeaver runtime library without passing code through its source-to-source translation tool the GridWeaver implementation performs up to 14.88 times slower than the original implementation (5.54 times slower as a median average). However, when code is passed through the translation tool the GridWeaver version executes within 10% of the original computation time.



Cray

	version (s)			ratio to orig	
cores	orig	lib	comp	lib	comp
24	1093.16	6058.38	1220.41	5.54	1.11
48	588.73	3238.58	617.88	5.50	1.04
92	300.80	1629.23	249.27	5.41	0.82
184	161.38	823.77	142.38	5.10	0.88
368	161.38	823.77	142.38	5.59	0.96

	Yellowstone					Gordon				
	version (s)			ratio to orig		version (s)			ratio to orig	
cores	orig	lib	comp	lib	comp	orig	lib	comp	lib	comp
32	342.84	2079.20	298.99	6.06	0.87	384.42	1277.23	313.46	3.32	0.81
64	179.21	1087.06	155.94	6.06	0.87	197.29	672.35	162.72	3.40	0.82
128	90.40	595.96	80.90	6.59	0.89	97.18	333.57	80.23	3.43	0.82
256	53.11	404.76	48.59	7.62	0.91	55.14	203.24	48.59	3.68	0.88
512	15.14	225.32	24.63	14.88	1.62	12.65	116.61	11.75	9.21	0.92
1024	19.43	260.80	23.50	13.41	1.20					

Figure 8.7: Performance of CGPOP with GridWeaver.

8.2 Using GridWeaver in SWM's horizontal-advection operation

The Shallow Water Model (SWM) code is a proxy application for the Global Cloud Resolution Model (GCRM) [5]. SWM is written in Fortran+MPI and in this section we discuss how to model SWM's horizontal advection operation in GridWeaver. Our GridWeaver based implementation contains 145 lines of code that model SWM's grid-connectivity, 54 lines of code that import data into GridWeaver, and 607 lines of code to conduct SWM's he horizontal advection stencils. GridWeaver eliminates the need for SWM's 1010 lines of communication and 561 of lines of existing stencil computation code, for a net reduction of 910 lines of code.

Like our case study of CGPOP described in Section 8.1, reimplementing SWM in GridWeaver requires modeling the grid SWM uses, as well as its decomposition. We describe how to model SWM's icosahedral grid in Section 8.2.1, and discuss how to match SWM's decomposition and import its data into GridWeaver in Section 8.2.2.

In Section 8.2.3 we describe how to model SWM's horizontal advection operations in GridWeaver, and in Section 8.2.4 we discuss the performance impact of doing so.

8.2.1 Modeling the icosahedral grid

In Figure 8.8 we illustrate the connectivity of an icosahedral grid. The grid consists of twelve subgrids: ten representing the twenty triangular faces of an icosahedron, and two representing the Earth's north and south poles. In Figure 8.9 we list code that models the connectivity for two subgrids (specifically subgrids `3L` and `3R`). The code to connect to the other eight, face related, subgrids is similar.

Lines 6 through 8 of Figure 8.9 define a six-point neighborhood. It is necessary to define this neighborhood to conduct communication for non-compact stencils, which we discuss in Chapter 6. The six point neighborhood implies that each node inside a subgrid represents a hexagonal cell. Nodes along the borders of a subgrid can contain fewer sides, depending on what border maps are defined for the subgrid. In the icosahedral grid there are twelve

pentagonal points, which we define as such by simply not specifying a border map for one of their neighbors.

As we discuss in Section 5.2.1, when updating pentagonal cells, the stencil accesses a non-existent, sixth, neighbor that always has a value of zero. This behavior correctly implements the stencils in SWM.

We take a total of 145 lines of code to model SWM's icosahedral grid in GridWeaver. In lines 25 through 47 of Figure 8.9 we include the lines that connect subgrids `3L` and `3R`. The other 98 lines of code used to model the icosahedral grid are similar to what we list in Figure 8.9 but handle other subgrids.

Specifically, in Figure 8.9, lines 26 and 27 connect the top border of subgrid `sg3L` to the left border of subgrid `sg4L`; lines 30 and 31 connect the right border of subgrid `sg3R` to the bottom border of subgrid `sg4R`; lines 34 and 35 connect the top border of subgrid `sg3R` to the bottom border of subgrid `sg4L`; lines 38 and 39 connect the right border of subgrid `sg3L` to the left border of subgrid `sg2L`; lines 42 and 43 connect subgrid `sg3L` to the subgrid for the north pole, and lines 46 and 47 connect subgrid `sg3R` to the south pole.

8.2.2 Importing decomposition and data

As we discuss in Section 8.1.1, data distribution and computation distribution can have a large impact on performance; thus to control for this impact in our experimental results, we programmed our GridWeaver based implementation of SWM to import the decomposition used in the original Fortran+MPI implementation.

Like CGPOP, SWM decomposes data into blocks (termed sub-domains in SWM), distributes these blocks to processors, and computes on blocks in an owner-computes fashion. We import SWM's distribution into GridWeaver by iterating over each block in SWM, determining the processor that SWM assigns to it, and informing GridWeaver of this assignment. We differ from SWM's distribution slightly in that we store north and south poles in separate subgrids that we distribute to processor 0; SWM stores north and south poles with redundant copies in each subgrid. We list code to conduct this import operation in Figure 8.10.

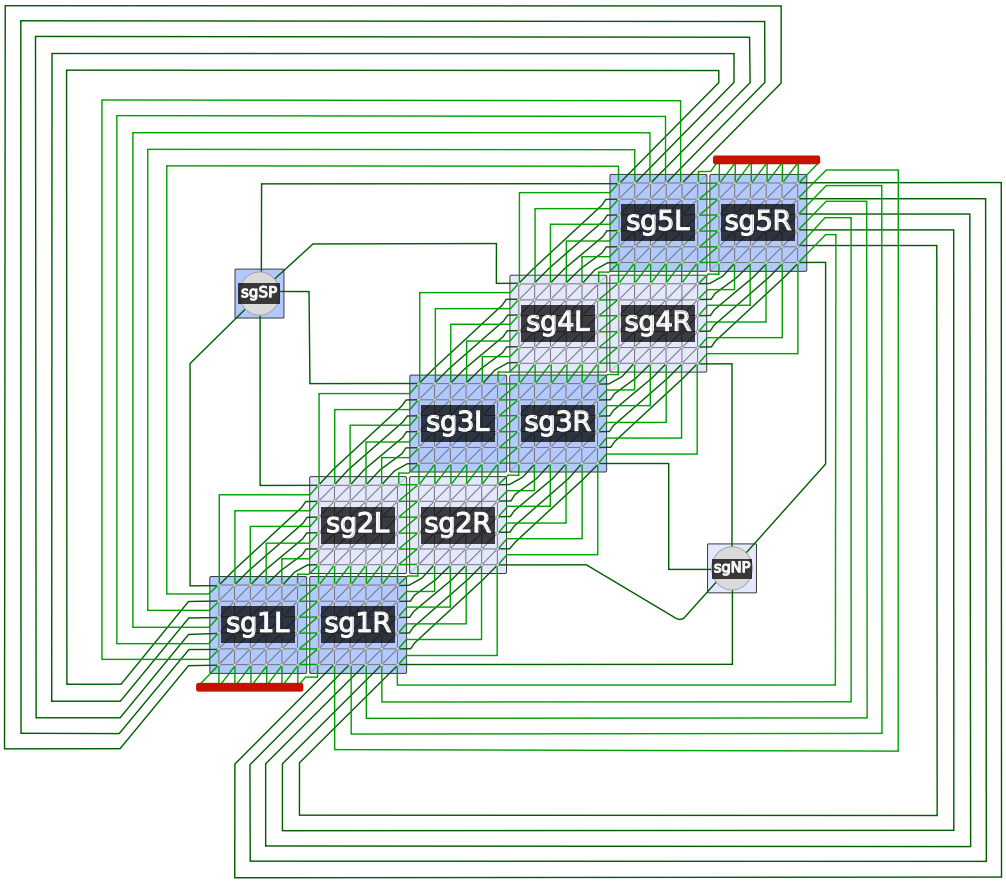


Figure 8.8: Connectivity of the icosahedral grid.

8.2.3 Modeling horizontal advection operation

We describe SWM's horizontal advection operation in Section 2.4 and SWM's horizontal advection operation consists of a non-compact stencil to compute flux values on edges and a compact stencil to apply flux values to cells; see Section 2.4 and Figure 2.7 for more details about this algorithm.

In our GridWeaver-based implementation we split the code into two stencil functions: `updateFlux`, which computes the non-compact stencil listed in lines 13-34 of Figure 2.7, and `applyFlux`, which applies the non-compact stencil in lines 46 through 54 of Figure 2.7. In each stencil we replace the arrays used in Figure 2.7 with GridWeaver data objects. In Figure 8.11 we list code that computes the horizontal advection operation 1,000 times by iteratively applying these stencils.

```
type(Neighbor) :: n1, n2, n3, n4, n5, n6
type(Subgrid)  :: sg1L, sg2L, sg3L, sg4L, sg5L, sg1R, sg2R, sg3R, sg4R,
                  sg5R, sgNP, sgSP
type(Grid)     :: g

n1 = neighbor_new("neigh1",  0,  1);  n2 = neighbor_new("neigh2", 1,  1)
n3 = neighbor_new("neigh3",  1,  0);  n4 = neighbor_new("neigh4", 0,  -1)
n5 = neighbor_new("neigh5", -1, -1);  n6 = neighbor_new("neigh6", -1, 0)

sg1L = subgrid_new("sg1L", N, N);  sg1R = subgrid_new("sg1R", N, N)
sg2L = subgrid_new("sg2L", N, N);  sg2R = subgrid_new("sg2R", N, N)
sg3L = subgrid_new("sg3L", N, N);  sg3R = subgrid_new("sg3R", N, N)
sg4L = subgrid_new("sg4L", N, N);  sg4R = subgrid_new("sg4R", N, N)
sg5L = subgrid_new("sg5L", N, N);  sg5R = subgrid_new("sg5R", N, N)
sgNP = subgrid_new("sgNP", 1, 1);  sgSP = subgrid_new("sgSP", 1, 1)

g = grid_new("g")
call grid_addSubgrid(g, sg1L);  call grid_addSubgrid(g, sg1R);
call grid_addSubgrid(g, sg2L);  call grid_addSubgrid(g, sg2R);
call grid_addSubgrid(g, sg3L);  call grid_addSubgrid(g, sg3R);
call grid_addSubgrid(g, sg4L);  call grid_addSubgrid(g, sg4R);
call grid_addSubgrid(g, sg5L);  call grid_addSubgrid(g, sg5R);
call grid_addSubgrid(g, sgNP);  call grid_addSubgrid(g, sgSP)

! Top and left borders (for subgrid 3)
call grid_addBorder(g,  1, N+1,  N,N+1,  sg3L,   1, N,  1, 1,  sg4L, 0)
call grid_addBorder(g,  0, 0,    0,N-1,  sg4L,   N, N,  1, N,  sg3L, -1)

! Right and bottom borders (for subgrid 3)
call grid_addBorder(g,  N+1,2,  N+1,N+1,  sg3R,   N,1,  1,1,  sg4R, -1)
call grid_addBorder(g,    1,0,    N,0,    sg4R,   N,N,  N,1,  sg3R,  1)

! Connect top and bottom borders
call grid_addBorder(g,   1,N+1,  N,N+1,  sg3R,   1,1,  N,1,  sg4L, 0)
call grid_addBorder(g,   1,0,    N,0,    sg4L,   1,N,  N,N,  sg3R, 0)

! Connect left and right borders
call grid_addBorder(g,  N+1,1,  N+1,N,  sg3L,   1,1,  1,N,  sg3R, 0)
call grid_addBorder(g,    0,1,    0,N,  sg3R,   N,1,  N,N,  sg3L, 0)

! Connect to north pole
call grid_addBorder(g,  0,N,  0,N,  sg3L,   1,1,  1,1,  sgNP, 1)
call grid_addBorder(g,  2,1,  2,1,  sgNP,   1,N,  1,N,  sg3L, -1)

! Connect to south pole
call grid_addBorder(g,  N+1,1,  N+1,1,  sg3R,   1,1,  1,1, sgSP, -1)
call grid_addBorder(g,    0,1,    0,1,  sgSP,   N,1,  N,1, sg3R,  1)
```

Figure 8.9: Code to model connectivity of an Icosahedral grid for subgrids 3L and 3R.

```
! Set subdomains in the ten panels
do i = 1, nsdm_glbl
    swm_tag = (i - 1) * (blkW*blkH) + 3
    gw_gbid = i
    gw_proc = get_proc(level_max, swm_tag)

    call distribution_setProcForBlock(dist, gw_gbid, gw_proc)
end do

! Set north pole to rank 0
gw_gbid = distribution_gbidAt(dist, sgNP, 1, 1)
call distribution_setProcForBlock(dist, gw_gbid, 0)

! Set north pole to rank 0
gw_gbid = distribution_gbidAt(dist, sgSP, 1, 1)
call distribution_setProcForBlock(dist, gw_gbid, 0)
```

Figure 8.10: Code to import data distribution from original SWM implementation into GridWeaver.

```
do i=1,1000
    call data_forceUpdate(data_mss,data_flx_1,data_flx_2,data_flx_3)

    call data_apply_noncompact(data_mss,
        tag(data_mss),
        rel(data_wght_1), rel(data_wght_2), rel(data_wght_3),
        rel(data_indx_1), rel(data_indx_2), rel(data_indx_3),
        rel(data_wnd_1),  rel(data_wnd_2),  rel(data_wnd_3),
        updateFlux)

    call data_apply(
        data_mss,
        data_area_inv,
        data_flx_1,    data_flx_2,    data_flx_3,
        data_d_edge_1, data_d_edge_2, data_d_edge_3,
        data_d_edge_4, data_d_edge_5, data_d_edge_6, applyFlux)
end do
```

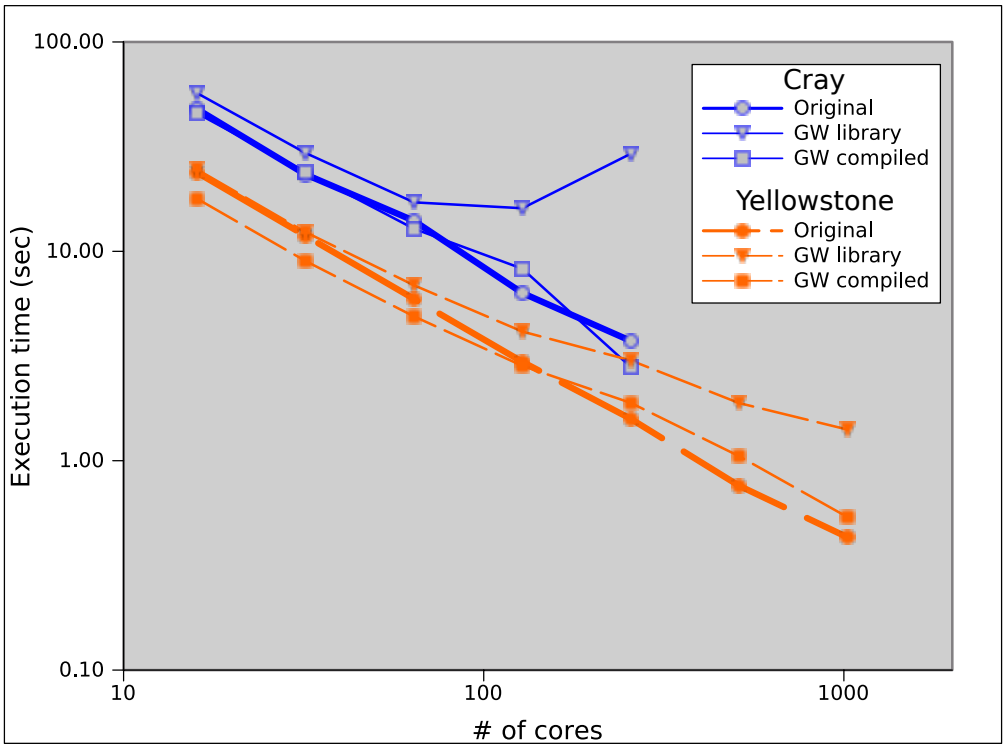
Figure 8.11: Code to apply SWM's horizontal advection operation 1,000 times in GridWeaver.

8.2.4 Impact on performance and programmability

SWM horizontal advection operation contains 14,822 lines of code, 1010 that are devoted to conducting communication to update halo values. By using GridWeaver we able to remove these 1010 lines of code. We spend 145 lines of code modeling the icosahedral grid and spend 607 lines of code implementing CGPOP's horizontal advection algorithm. We are able to net a loss of 910 lines of code.

In Figure 8.12 we present experimental results that compare the performance of the original SWM against our GridWeaver based implementation on the machines listed in Table 8.1.

When using the GridWeaver runtime library without passing code through its source-to-source translation tool the GridWeaver implementation performs at worst 7.84 times slower than the original implementation (1.27 times slower as a median average) However, when code is passed through the translation tool the GridWeaver version executes at worst 1.38 slower and at best 1.33 times faster than the original computation time.



Cray

	version (s)			ratio	
cores	orig	lib	comp	lib:orig	comp:orig
16	47.80	56.95	45.63	1.19	0.95
32	23.30	29.50	23.94	1.27	1.03
64	14.00	17.14	12.82	1.22	0.92
128	6.32	16.06	8.26	2.54	1.31
256	3.73	29.26	2.80	7.84	0.75

Yellowstone

	version (s)			ratio	
cores	orig	lib	comp	lib:orig	comp:orig
16	23.78	24.62	17.84	1.04	0.75
32	11.86	12.36	8.99	1.04	0.76
64	5.93	6.89	4.89	1.16	0.82
128	2.97	4.14	2.84	1.40	0.96
256	1.58	3.01	1.89	1.90	1.19
512	0.76	1.88	1.05	2.48	1.38
1024	0.43	1.41	0.54	3.27	1.25

Figure 8.12: Performance of GridWeaver in SWM.

Chapter 9

The GridWeaver active library

This Chapter serves as a user guide for the GridWeaver active library. In Section 9.1 we present the GridWeaver package's organization of a runtime library and code generation tool. Section 9.2 includes user instructions for unpacking and compiling the GridWeaver library and source-to-source compiler. In Section 9.3 we discuss how to link to GridWeaver and compile using GridWeaver's code generation tool; we also give an example of how to write a batch script to deploy applications written using GridWeaver on large clusters. In Section 9.4 we guide the user through a complete example of writing a program to compute the Conway's Game Of Life on the mini-icosahedral grid (illustrated in Figure 6.1).

9.1 Software architecture and package organization

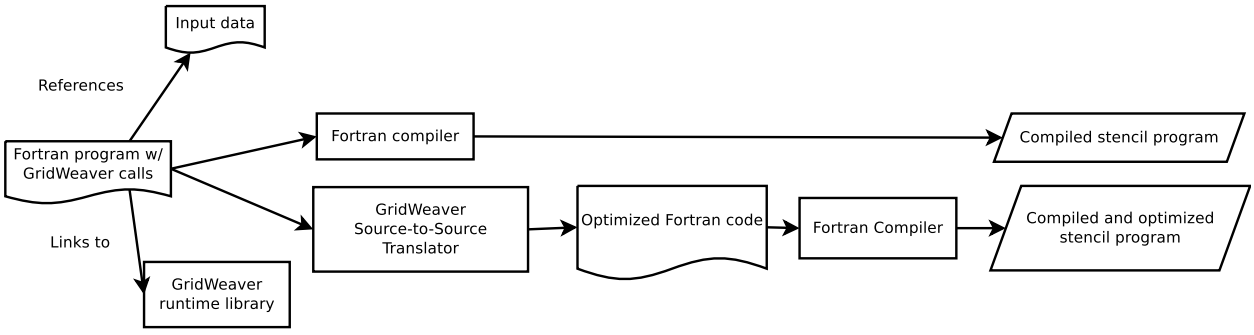


Figure 9.1: System diagram of the GridWeaver package.

In Figure 9.1 we illustrate two use cases of the GridWeaver tools. The GridWeaver library includes abstractions for specifying grids and functions for applying stencil computations, reductions, and halo updates. The GridWeaver source-to-source translation tool applies program optimizations to improve performance.

Although programs that use the GridWeaver library will produce correct results, use of the library results in suboptimal performance. Library approaches introduce overhead from calls to library functions and are unable to take advantage of certain compile-time optimizations; for example, fusing loops from multiple stencil computations together. However, despite the performance limitations of a library approach it does offer some advantages. Code linked to GridWeaver will work with most debugging tools such as GDB and library calls can be integrated into existing projects.

To address the performance limitations of the GridWeaver library, the GridWeaver project also includes a source-to-source code translation tool. The translation tool takes as input a Fortran program that uses the GridWeaver library. The translation tool uses the Rose framework to parse the Fortran code and detect GridWeaver library calls. GridWeaver then replaces the library calls with optimized code. The code produced by GridWeaver can then be passed to a Fortran compiler to produce an executable.

9.2 Installation

To build GridWeaver run the makefile for your machine; we include makefiles for the ISTeC Cray, Yellowstone, and Gordon (the three machines we use in our case studies), which are listed in Table 8.1. The makefile for the ISTeC Cray, Yellowstone, and Gordon will compile and build a C++ library, a Fortran runtime library, and a sample testing program. The Makefile for the ISTeC cray assumes that system has modules for GCC 4.6.3 xt-mpt 5.3.5 , and NetCDF 4.1.2 loaded; the makefile for Yellowstone assumes the system has modules for GCC 4.7.2, and NetCDF 4.2 loaded; and the makefile for Gordon assumes the system has modules for ICC 12.1.0, mvapich2_ib 1.8a1p1, and NetCDF 4.0.1 loaded. We also include a Makefile for a generic linux machines that has as access to GCC 4.7.2, OpenMPI 1.5.4, NetCDF 4.1.2, and Rose 0.9.5a. This generic makefile will build the libraries and sample program like the other makefiles, but will also produce GridWeaver's source-to-source translation tool.

machine	makefile
ISTeC Cray	`Makefile.cray2`
Yellowstone	`makefile.yellowstone`
Gordon	`makefile.gordon`
Generic linux	`makefile`

These Makefiles expect modules to be loaded for GCC (on the ISTeC Cray and YellowStone), Rose, and NetCDF. To install GridWeaver for one of these platforms run the appropriate Makefile in the root GridWeaver directory. To build on a generic Linux machine simply run "`make`"; to run on a specific machine run the appropriate makefile, for example with Gordon, execute:

```
make -f Makefile.gordon
```

9.3 Invocation and Use

After compiling, users may run a batch job of GridWeaver's sample stencil program by changing directories into `runtimelib/` and running one of the following commands on the appropriate platform:

machine	batch file
ISTeC Cray	`qsub cray2_24.pbs`
Yellowstone	`bsub < yellowstone_16.bsub`
Gordon	`qsub gordon_16.pbs`

To use GridWeaver in one of your programs include the GridWeaver header file and call the routines detailed in Appendix B. To link the GridWeaver library to a program either link to the static version by adding `libgridweaver.a` to your compilation command, or add the dynamic version by adding `-lgridweaver`.

Since the runtime library has components in both Fortran and C++ it may also be necessary to link to the standard C++ library by adding `-lstdc++` to your compile command. On some systems we also found it necessary to link to a C++ version of the MPI library by adding `-lmpi_cxx`.

To use the GridWeaver source-to-source translation tool simply invoke the `gridweaver` tool on the desired source file. For example:

```
./gridweaver input.F90
```

9.4 Conway's Game of Life on an icosahedral grid

In this section we provide a tutorial GridWeaver by describing how to implement Conway's Game of Life [78] on an icosahedral grid [123, 24, 86]. Conway's game of life iteratively conducts a stencil and is a well known toy problem for cellular automata. Specifically we describe the rules for the game of life (Section 9.4.1), how to model and test a model of the icosahedral grid (Section 9.4.2), how to encapsulate the rules for Life in a stencil function (Section 9.4.3), and finally how to build and execute from the command line (Section 9.4.4). We provide full GridWeaver code for Life in Figure C.7.

9.4.1 Rules for the Game of Life

Conway's game works by iteratively computing a cellular-automata. The automata simulates complex growth and decay patterns that emerge from a small set of simple rules. Conway's game is a popular "toy problem" in a diverse range of fields including artificial intelligence [45], biology [49], biochemistry [90], computation theory [125], ecology [63], economics [100], mathematics [43], philosophy [77, 68], and physics [130]. The Game of Life is commonly used to illustrate concepts such as computation [125], self replication [54], and emergent behavior [115].

Typically, and as described by Conway himself [78], the Game of Life is played on a regular, infinite, two-dimensional grid of square cells. On each turn, each cell is computed to either be set to a dead state or a live state, based on the current state of the cell and the states of its eight surrounding neighbors. Specifically, the rules are as follows:

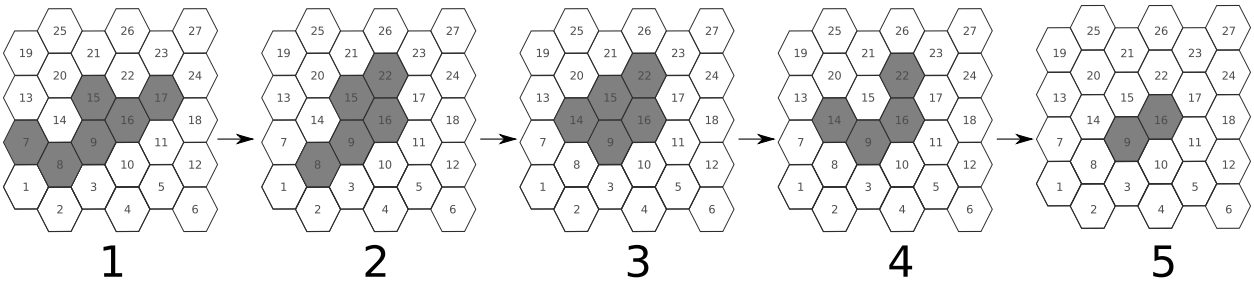


Figure 9.2: Five iterations of Conway's game of life applied on a hexagonal subgrid. Live cells are shaded with gray; dead cells are white. We label each cell with a unique integer so that we can refer to specific cells in the text.

1. A live cell with one or zero live neighbors dies (*underpopulation*).

2. A live cell with two or three live neighbors lives on (*stable population*).

3. A live cell with more than three neighbors dies (*overpopulation*).

4. A dead cell with exactly three neighbors becomes alive (*reproduction*).

For our GridWeaver based implementation we modify Conway's game to operate on a semiregular icosahedral grid. Within each subgrid are a series of hexagonal cells. We illustrate five iterations of Life in Figure 9.2. In iteration 2 of the game, cell 7 and cell 17 die because they each have only one neighbor (underpopulation rule), cell 22 becomes alive because it has exactly three live neighbors (reproduction rule), and the remaining live cells stay alive because they have either two or three live neighbors (stable population rule). In iteration 3, cell 8 dies from underpopulation, and cell 14 becomes alive because of the reproduction rule. In iteration 4, cell 15 dies because it has more than three neighbors (overpopulation rule). In iteration 5 cells 14 and 22 die from underpopulation.

9.4.2 Modeling an icosahedral grid

In Section 8.2.1 we describe how we modeled an icosahedral grid for our case study using the Shallow Water Model (SWM) code. When modeling complex grids users may want to verify the correctness of the grid connectivity. To do this we suggest constructing a data object with the `data_new` routine, populating initial values in the data object with

the data_initializeSeqVals routine, and outputting block and halo values with the data_printForProcs routine.

In Figure 9.3 we illustrate an icosahedral grid where each subgrid contains 3x3 nodes. The bottom of the figure is a listing of console output produced by the data_printForProcs when we initialize values using the data_initializeSeqVals, and distribute data so that blocks are sized to be 3x3 nodes.

The red values in Figure 9.3 correspond to locally owned nodes of each block, and the surrounding gray values correspond to halo values. By manually checking the halo values users can verify the correctness of their grid-connectivity specification. For instance notice that in the grid illustrated in Figure 9.3 the top row of values in subgrid sg1L connect to the left-most column of values in subgrid sg2L.

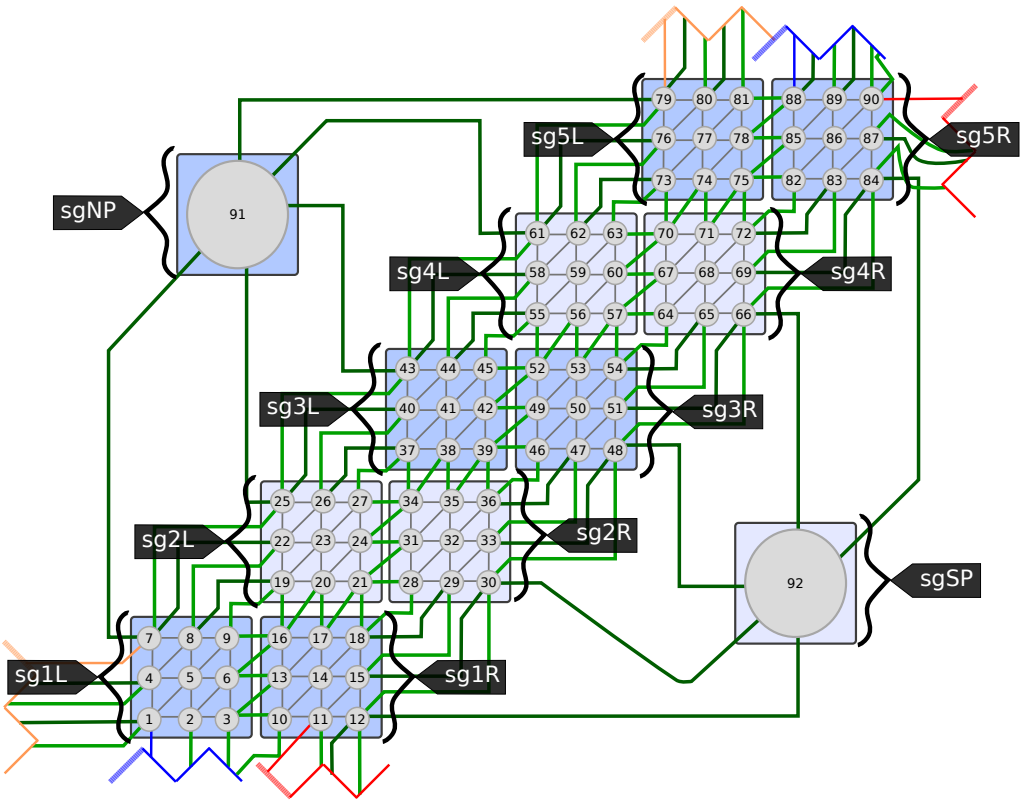
If the connectivity specification for this grid is correct, then we would expect that in the halo for the block covering sg1L would have the values [25, 22, 19] in the halo region above the values of its top row. This is indeed the case so we know the connectivity specification between sg1L and sg2L is correct.

9.4.3 Game of life stencil function

In Figure 9.4 we list code for a stencil function that evaluates these four rules for a cell (i, j) on the icosahedral grid. In Figure 9.5 we list code that uses this stencil function to apply the game of life for n iterations.

In Figure 9.5 the golStencil function is applied by passing it to the data_apply routine (line 4). The data_apply routine is passed two data objects, one that stores the cell values from the previous iteration (earth) and one that stores cell values for the current iteration (earthNext).

In each of these data objects a value of 1 signifies that a cell is alive and a value of 0 signifies that a cell is dead. To update halo values before applying the stencil we call the data_forceUpdate routine (line 3), and to swap we call the data_pingPong routine (line 5).




```
Legend: internal halo
Subgrid: sg1L
Block  1
   0.0   25.0   22.0   19.0    0.0
  91.0    7.0    8.0    9.0   16.0
  79.0    4.0    5.0    6.0   13.0
  80.0    1.0    2.0    3.0   10.0
  81.0   88.0   89.0   90.0    0.0
Subgrid: sg1R
Block  2
   0.0   19.0   20.0   21.0   28.0
   9.0   16.0   17.0   18.0   29.0
   6.0   13.0   14.0   15.0   30.0
   3.0   10.0   11.0   12.0   92.0
   0.0   90.0   87.0   84.0    0.0
Subgrid: sg2L
Block  3
   0.0   43.0   40.0   37.0    0.0
  91.0   25.0   26.0   27.0   34.0
   7.0   22.0   23.0   24.0   31.0
   8.0   19.0   20.0   21.0   28.0
   9.0   16.0   17.0   18.0    0.0
Subgrid: sg2R
Block  4
   0.0   37.0   38.0   39.0   46.0
  27.0   34.0   35.0   36.0   47.0
  24.0   31.0   32.0   33.0   48.0
  21.0   28.0   29.0   30.0   92.0
   0.0   18.0   15.0   12.0    0.0
Subgrid: sg3L
Block  5
   0.0   61.0   58.0   55.0    0.0
  91.0   43.0   44.0   45.0   52.0
  25.0   40.0   41.0   42.0   49.0
  26.0   37.0   38.0   39.0   46.0
  27.0   34.0   35.0   36.0    0.0
Subgrid: sg3R
Block  6
   0.0   55.0   56.0   57.0   64.0
  45.0   52.0   53.0   54.0   65.0
  42.0   49.0   50.0   51.0   66.0
  39.0   46.0   47.0   48.0   92.0
   0.0   36.0   33.0   30.0    0.0
Subgrid: sg4L
Block  7
   0.0   79.0   76.0   73.0    0.0
  91.0   61.0   62.0   63.0   70.0
  43.0   58.0   59.0   60.0   67.0
  44.0   55.0   56.0   57.0   64.0
  45.0   52.0   53.0   54.0    0.0
Subgrid: sg4R
Block  8
   0.0   73.0   74.0   75.0   82.0
  63.0   70.0   71.0   72.0   83.0
  60.0   67.0   68.0   69.0   84.0
  57.0   64.0   65.0   66.0   92.0
   0.0   54.0   51.0   48.0    0.0
Subgrid: sg5L
Block  9
   0.0    7.0    4.0    1.0    0.0
  91.0   79.0   80.0   81.0   88.0
  61.0   76.0   77.0   78.0   85.0
  62.0   73.0   74.0   75.0   82.0
  63.0   70.0   71.0   72.0    0.0
Subgrid: sg5R
Block 10
   0.0    1.0    2.0    3.0   10.0
  81.0   88.0   89.0   90.0   11.0
  78.0   85.0   86.0   87.0   12.0
  75.0   82.0   83.0   84.0   92.0
   0.0   72.0   69.0   66.0    0.0
Subgrid: sgNP
Block 11
   0.0   79.0   61.0
   0.0   91.0   43.0
   7.0   25.0    0.0
Subgrid: sgSP
Block 12
   0.0   66.0   84.0
  48.0   92.0    0.0
  30.0   12.0    0.0
```

Figure 9.3: Console output for initialized mini-icosahedral grid.

```
! Rules:
! (1) a live cell with one or zero live neighbors dies
! (2) a live cell with two or three live neighbors lives on
! (3) a live cell with more than three neighbors dies
! (4) a dead cell with exactly three neighbors becomes alive

stencil(golStencil, A, i, j)
  integer, intent(in) :: nNeighbors

  ! Count number of neighboring cells containing life
  nNeighbors = &
    A(i,j+1) + A(i+1,j+1) + A(i+1,j) + A(i,j-1) + A(i-1,j-1) + A(i-1,j)

  ! Update cell using the game-of-life rules:
  if(A(i, j) == 1) then       ! Check rules (1), (2), and (3)
    if(nNeighbors < 2);                        golStencil = 0; return
    if(nNeighbors == 2 .or. nNeighbors == 3);  golStencil = 1; return
    if(nNeighbors < 2);                        golStencil = 0; return
  else                        ! Check rule (4)
    if(nNeighbors == 3); golStencil = 1; return
  end if
  golStencil = 0
end function
```

Figure 9.4: Stencil computation for Conway's game of life applied on a hexagonal field.

```
! Run for n turns
do i=1,n
  call data_forceUpdate(earth)
  call data_apply(earth, earthNext, golStencil)
  call data_pingPong(earth, earthNext)
end do
```

Figure 9.5: Code to apply game of Life stencil.

```
1  FTN=mpif90
2  LDFLAGS=-rdynamic -lgridweaver -lstdc++ -lmpi_cxx
3
4  DRIVER=gol
5  OBJS=gol.o
6
7  $(DRIVER): $(OBJS)
8      $(FTN) -o $(DRIVER) $(OBJS) $(LDFLAGS)
9
10 %.o: %.F90
11     $(FTN) $(FTNFLAGS) -c $<
12
13 clean:
14     -rm -f $(DRIVER) $(OBJS)
```

Figure 9.6: Makefile for game of Life

9.4.4 Compiling and executing

In Section 9.3 we described how to compile and invoke programs that use GridWeaver. In Figure 9.6 we list a Makefile that will compile and build the game of life program with GridWeaver [3].

Users wishing to compile the game-of-life themselves may have to modify this Makefile depending on the machine they are on. For example the MPI+Fortran is `ftn` on the ISTeC Cray, `mpfort` on Yellowstone, and `mpif90` on Gordon.

Once built the program can be run like any other MPI program. For example with:

```
mpirun -np 4 ./gol
```

9.4.5 Summary

In this section we described how to use GridWeaver with an example implementation of Conway's game of Life. We describe Conway's game of life in Section 9.4.3; in Figure 9.4 we include code to conduct the game of Life's stencil operation, and in Figure 9.5 we show

[3] specifically for machines in the Linux Labs for the Department of Computer Science at Colorado State University as of July, 2013.

how to apply this stencil for a set number of time-steps. In section 9.4.2 we discuss how to model an icosahedral grid, but users could modify this code (without having to modify the Figures 9.4 and 9.5) so that they could apply the game of Life to a different grid. We describe how to compile a GridWeaver-based implementation of Game of Life in Section 9.4.4, and provide a complete code listing for Life in Figure C.7 of Appendix C.

Chapter 10

Current limitations and future work

The GridWeaver model is currently limited to applying stencil computations on statically defined semiregular grids. The GridWeaver active library is currently limited to realizing computation on shared memory and distributed memory systems. Certainly, we could improve GridWeaver by generalizing the its abstractions to work on additional types of grids, generalizing its implementation to work with different architectures, and adding additional optimizations to further improve performance.

10.1 Relaxing implementation constraints

In order to ease the processing of implementing our prototypes of GridWeaver we made certain assumptions. These assumptions do not but GridWeaver could be made a more mature tool by relaxing the following constraints:

- **Generic data types** — currently GridWeaver only allows users to store eight-bit floating-point values in data objects. We could relax this constraint by including different data objects for several commonly used types (such as integer, complex numbers, etc.). If we reimplemented GridWeaver in a language other than Fortran we could accomplish this by using language features for generic data types (such as templates in C++ or generics in Java). Researchers have proposed extending Fortran to include such features [58]. Until such extensions are added to the Fortran standard, generic data types could be implemented by using the C++ preprocessor.

- **Multi-dimensional grids** — GridWeaver currently operates on two-dimensional grids. However, many Earth science codes operates on three dimensional grids where the third dimension corresponds to elements of altitude.

- **Stencils with more input parameters** — GridWeaver is limited to applying stencils with ten or fewer input parameters.

10.2 Additional optimizations and target architectures

A number of optimizations are commonly applied to stencil computations. Some optimizations that we have not integrated into GridWeaver include:

- **Tiling** [153] — reorganize a loop-nest so that it iterates through blocks of data that fit into cache. **Multi-level tiling** embeds multiple levels of tiles inside a loop nest to address multiple levels of the memory hierarchy.

- **SIMDization** [105] — parallelizes stencil code through SIMD vectorization calls. These calls use special SIMD CPU registers to perform the same operation on multiple pieces of data simultaneously.

- **Option to Store Data as Array of Structs or Struct of Arrays** [70] — when a grid node has multiple values assigned to it, programs can either store these values in separate arrays (as a struct of arrays) or in a single array where each element is a structure holding these multiple properties (as an array of structs). The struct of array style exposes opportunities for vectorization while struct-of-array style may allow for better cache behavior depending on what values are accessed by a stencil computation.

- **Array padding** [102] — pad the ends arrays in order to ensure that their data is stored on separate cache-lines. This is done in order to avoid data cache thrashing (cache conflicts).

- **Software Controlled Prefetching** [29] — some architectures include explicit fetch instructions to control prefetching of data into cache. Prefetching avoids memory-fetch latency by moving data to cache prior to the CPU referencing it.

- **Loop unrolling** [71] — unroll a loop so that its body includes multiple iterations of the previous version of the loop. Loop unrolling optimizations aim to reduce loop overhead. Compilers are often able to perform this optimization automatically.

- **Autotuning** — autotuning approaches automatically and iteratively perform experiments and use statistical analysis and machine learning techniques to tune parameters such as block size to optimize for execution time, power, or another specified metrics.

We wrote GridWeaver to execute on large clusters that consists of several processors connected to one another via network. Each processor has access to local data and in order to access remote data communication must occur over a network. GridWeaver addresses communication and synchronization using the MPI library. However, modern supercomputers contain components based on different types of memory models. Future work in improving GridWeaver could focus on enabling the active library to optimally make use of such components. For example –

- **Multicore Systems** [108, 95, 113, 65, 64] — multicore systems consist of multiple cores that have access to a shared pool of memory. Cores can access this shared pool without having to explicitly invoke communication calls. Although in its current implementation GridWeaver will produce a correct result when executed on a shared memory system, it may be possible to improve its performance by avoiding MPI communication calls when accessing shared data. Modern supercomputing architectures often consist of several shared memory systems that connect to one another over a specialized network.

- **Vectorization** [105, 104] — vectorization is a type of parallelism where the same operation is simultaneously applied to multiple pieces of data. Intel's MMX, SSE, AVX and include vectorization instructions [15]. Optimizing compilers should be able to automatically apply vectorization in loops (such as the stencil loop) when applicable. However, we have not examined whether compilers actually do apply the optimal vectorization for GridWeaver. Future work could include examining this.

- **Accelerators** [84, 156, 47, 28] — it has become more common to use Graphical Processing Units (GPUs) to perform general computation such a stencil operations. GPUs are able to quickly apply some operation (called a kernel) to different pieces of data at the same time. Compared to CPUs GPUs have very high compute-bandwidth, but the performance gained by the GPUs parallelism must amortize the cost of sending data back and forth from the GPU. Recently, architectures such as Intel's Many Integrated Circuit (MIC) aim to integrate accelerators directly onto a system's chipset.

10.3 Identifying regularity in existing meshes

We designed GridWeaver with semiregular Earth grids in mind, but Earth science is not the only domain where meshes exhibit regularity. For example, semiregular meshes appear in the field of computer graphics and scientific visualization. InWood et al. [154] discuss a method for generating semiregular meshes from finite volumes. Their technique works by first generating a coarse triangular mesh and then refining it via recursively applying a quadrisection algorithm to a subset of the triangular elements. GridWeaver's semiregular

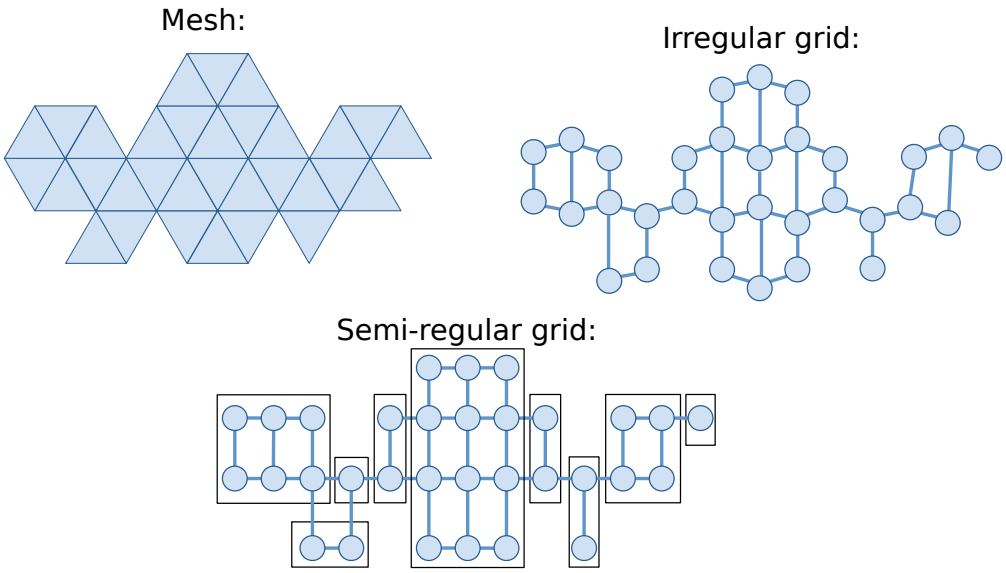


Figure 10.1: A triangular mesh with its representation as an irregular grid and semiregular grid.

grids differ from the structures discussed in [154] by the fact that GridWeaver stores regular portions of the mesh in a regular data structure (a 2D array).

Since all irregular grids are expressible as semiregular grids, by having each node contained in its own subgrid, it may be possible to automatically identify regularity in existing meshes and generate GridWeaver-based specifications of the grid. Since GridWeaver introduces overhead for each subgrid, a stencil that operates on a semiregular grid may perform better or may perform worse than when applied to an equivalent irregular grid.

10.4 Visualizing grid connectivity

In Section 9.4.2 we discuss how to hand verify GridWeaver-specified connectivity of an icosahedral grid. In this section we guide the user through populating the grid with unique values and using the to manually check halo values by examining output from the `data_printForProcs` function. This process is tedious and error prone. By creating a tool to visualize grid connectivity we could enable users to more quickly and accurately check connectivity. In Figure 10.2 we illustrate an example visualization for the cubed sphere grid.

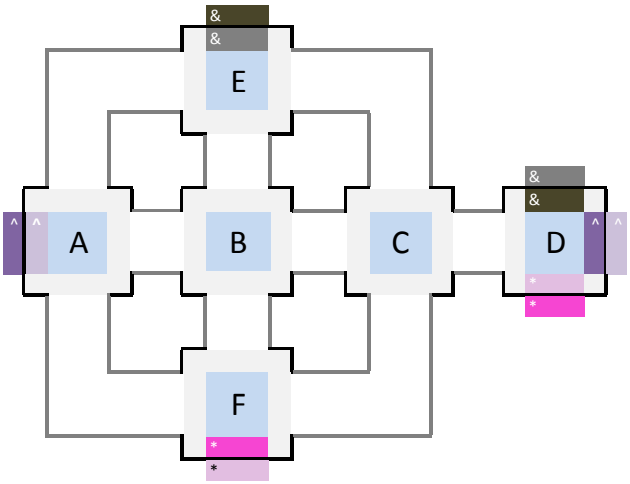


Figure 10.2: Example visualization of cubed-sphere topology.

Chapter 11

Conclusions

In this dissertation we focus on improving the programmability and maintainability of Earth simulation applications. Earth simulation applications are used by scientists to gain insight into how environmental systems function and to make predictions about these systems' future conditions. These applications simulate weather and climate by numerically solving systems of Partial Differential Equations (PDEs). In hand-written Earth simulation codes, such as POP [94] and GCRM [5], PDEs are solved with stencil computations. Stencil-computation code is often tangled with implementation details such as grid connectivity and data decomposition. We reduce tangling by introducing abstractions that enable an orthogonal specification of stencil computation, grid connectivity, and data decomposition. We implement these abstractions in the GridWeaver active library. Specifically, we use border maps to specify connectivity, a block-based decomposition abstraction to map data to processing elements, and enable users to define stencil operations as functions that access neighboring grid values via array (as is done by regular stencils). GridWeaver automates communication for the user.

In GridWeaver, users specify stencil operations as functions that are passed the location of a grid node to update. Stencil functions are also passed an access function, which enables users to access neighboring values with an array-access-style syntax. To enable stencils to access neighboring values owned by different processing elements, GridWeaver automates communication to populate locally accessible halos and ghost-cell lists. Specifically, GridWeaver determines what communication to perform, and stores communication metadata into a communication plan abstraction.

We compare GridWeaver to other stencil generation tools by introducing terminology to describe how features separate algorithm code from implementation concerns. We discuss

the tradeoffs these different approaches make in terms of how they tangle algorithm code, how much control they enable the programmer to have, and how much they leave as the responsibility of the programmer. In GridWeaver we enable specification of stencil computations in an injected manner and address decomposition and connectivity in a delegated fashion.

We conduct case studies of GridWeaver by rewriting portions of the computations in CGPOP and SWM. CGPOP is a proxy application for version 1.4.3 of the Parallel Ocean Program (POP) [94] and performs a compact stencil operation on a dipole grid. In version 2.0, POP's developers modified it to perform the stencil on a tripole Grid. We modify our GridWeaver-based implementation to use a Tripole grid by simply adding two additional lines of connectivity specification code. SWM is a proxy application of the shallow water model component of the Global Cloud Resolution Model (GCRM). We conduct a case study with SWM because it includes a non-compact stencil seen in a real-world application, and because it is applied to a grid with a complex connectivity pattern (an icosahedral grid).

Using the GridWeaver library we are able to achieve performance for CGPOP that is at worst 14.88 times slower than the original version and at best 3.32 times slower; for SWM we are able to achieve performance that is at worst 7.84 times slower and at best 1.04 times slower. To solve this issue we develop a source-to-source translation tool that inlines GridWeaver function calls. When we apply the source-to-source translator to CGPOP we are able to achieve performance that is at worst 1.62 times slower and at best 1.23 times faster. When we apply the source-to-source translator to SWM, we are able to achieve performance that is at worst 1.38 times slower and at best 1.33 times faster.

In both our CGPOP and SWM case studies GridWeaver eliminates the need for the programmer to explicitly write communication code. The original version of CGPOP included 1,226 lines of communication code, and the original version of SWM included 1,010 lines of communication code. Since there is now a separation of concerns not only can the communication plan be automatically derived, but also other program optimizations that cross cut these concerns become possible, for example communication avoiding.

We also propose directions for future work; such work could focus on overcoming limitations made in our initial implementation of GridWeaver. For example, we could relax our assumption that subgrids are two-dimensional and that grid data is stored as eight-bit floating point values. We could also improve GridWeaver by introducing adding additional performance optimizations, enabling it to target additional architectures, and by adding a grid visualization tool. Finally, we describe how semi-regular grids are used in applications outside of Earth science simulation codes (e.g. semi-regular meshes are used in computer graphics), and we propose identifying regularity and automatically generating semiregular specifications from existing irregular mesh specifications.

The main contribution of this dissertation is the development of the GridWeaver programming model and the active library that realizes it. In order to have GridWeaver automate communication, we introduce communication plan generation algorithms. We use different algorithms depending on whether communication is necessary to conduct a compact or a non-compact stencil. Additionally, we introduce terminology for describing and comparing programming models. We use this terminology to compare GridWeaver to existing programming models for stencil computations. We also describe how proxy applications can be used to evaluate programming models, and we use the CGPOP and SWM proxy applications to conduct case studies with GridWeaver. To maintain the original performance of the proxy-applications we introduce a source-to-source translation tool that inlines GridWeaver library calls.

References

[1] *Programming with POSIX Threads.* Addison-Wesley, Boston, MA, USA, 1997. (48)

[2] Design and implementation of components in the Earth System Modeling Framework. 19(3):341–350, 2005. (94)

[3] International Exascale Software Project. http://www.exascale.org/iesp/, 2011. (38)

[4] MPAS: Model for Prediction Across Scales. http://mpas.sourceforge.net/, December 2011. (6, 30, 37, 43, 44, 48, 49)

[5] Design and testing of a global cloud resolving model. http://kiwi.atmos.colostate.edu/gcrm/, July 2012. (2, 3, 24, 96, 118)

[6] Spherical geodesic grids: A new approach to modeling the climate. http://kiwi.atmos.colostate.edu/BUGS/geodesic/, July 2012. (5)

[7] Center for Computational Geophysics. http://earthscience.rice.edu/centers/ccg/, 2013. (1)

[8] CGPOP miniapp website. http://www.cs.colostate.edu/hpc/cgpop/, 2013. (19)

[9] Chombo: Mapped multiblock grids. http://commons.lbl.gov/display/chombo/Mapped+Multiblock+Grids, 2013. (42)

[10] Community Earth System Model (CESM). http://www.cesm.ucar.edu/, 2013. (1, 16)

[11] Global cloud resolving models. http://www.scidacreview.org/0904/html/hardware1.html, 2013. (13)

[12] Gordon user guide. http://www.sdsc.edu/us/resources/gordon/, 2013. (78, 85, 86)

[13] Hopper, NERSC's Cray XE6 system. http://www.nersc.gov/users/computational-systems/hopper/, 2013. (22)

[14] Image: NCEP T62 Gaussian grid. http://en.wikipedia.org/wiki/File:NCEP_T62_gaussian_grid.png, 2013. (12)

[15] *Intel 64 and IA-32 Architectures Software Developer Manuals (Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C)*. Intel, 2013. (115)

[16] ISTeC Cray High Performance Computing (HPC) system. http://istec.colostate.edu/istec_cray/, 2013. (78, 85, 86)

[17] Lynx; Computational Information Systems Laboratory. http://www2.cisl.ucar.edu/resources/lynx, 2013. (22)

[18] National Institute for Computational Sciences: Kraken. http://www.nics.tennessee.edu/computing-resources/kraken, 2013. (17, 22)

[19] NCAR news release: NCAR adds resources to TeraGrid. http://www.ucar.edu/news/releases/2007/teragrid.shtml, 2013. (22)

[20] NetCDF (Network Common Data Form) – website. http://www.unidata.ucar.edu/software/netcdf/, 2013. (19)

[21] OpenACC: Directives for accelerators. http://www.openacc-standard.org/, 2013. (30, 48)

[22] OpenCL overview. http://www.khronos.org/opencl/, 2013. (30, 48)

[23] SLOCCount tool website. http://www.dwheeler.com/sloccount/, 2013. (5)

[24] Spherical geodesic grids: A new approach to modeling the climate. http://kiwi.atmos.colostate.edu/BUGS/geodesic/, 2013. (12, 106)

[25] Unified Z-grid Icosahedral Model (UZIM). http://earthsystemcog.org/projects/dcmip-2012/uzim, 2013. (13)

[26] US energy information administration (EIA): Frequently asked questions. http://www.eia.gov/tools/faqs/faq.cfm?id=97&t=3, 2013. (38)

[27] Yellowstone; Computational Information Systems Laboratory. http://www2.cisl.ucar.edu/resources/yellowstone, 2013. (78, 85, 86)

[28] Mametjanov A., Lowell D., Ma C., and Norris B. Autotuning stencil-based computations on GPUs. Technical Report ANL/MCS-P2094-0512, Argonne National Laboratory, May 2012. (116)

[29] Ramprasad Venkataraman Aaron Becker and Laxmikant V. Kale. Patterns for overlapping communication and computation. In *Workshop on Parallel Programming Patterns*, ParaPLOP 2009, June 2009. (114)

[30] Tilak Agerwala and Siddhartha Chatterjee. Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro*, 25(3):58–69, 2005. (29, 30, 49)

[31] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification. `http://research.sun.com/projects/plrg/`. (30)

[32] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, August 2001. (33)

[33] Neculai Andrei. A simple three-term conjugate gradient algorithm for unconstrained optimization. *Journal of Computational and Applied Mathematics*, 241:19–29, March 2013. (23)

[34] ANSI/ISO/IEC. *Database Languages: SQL*, 1999. (38)

[35] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009. (14, 29, 38, 49)

[36] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991. (43)

[37] Saman Babaie-Kafaki, Reza Ghanbari, and Nezam Mahdavi-Amiri. Two new conjugate gradient methods based on modified secant equations. *Journal of Computational and Applied Mathematics*, 234(5):1374–1386, July 2010. (23)

[38] John Backus. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. (29)

[39] M. Barad, P. Colella, D. T. Graves, T. J. Ligocki, D. Modiano, P. O. Schwartz, and B. Van Straalen and. EBChombo software package for cartesian grid, embedded boundary applications. Technical report, Lawrence Berkeley National Laboratory, February 2000. (30, 43)

[40] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th Interntional Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004. (33)

[41] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with Hierarchically Tiled Arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM. (30)

[42] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216, New York, NY, USA, 1995. ACM. (30)

[43] Robert A. Bosch. Integer programming and Conway's game of Life. *SIAM Review*, 41(3):pp. 594–604, 1999. (106)

[44] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(34):421 – 444, 1998. (87)

[45] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991. (106)

[46] DavidL. Brown, William D. Henshaw, and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations. In *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 177–184. Springer Berlin Heidelberg, 1997. (43, 44, 45, 46, 47, 48)

[47] Choudary C., Godwin J., Holewinski J., Karthik D., Lowell D., Mametjanov A, Norris B., Sabin G., and Sadayappan P. Stencil-aware GPU optimization of iterative solvers. Technical Report ANL/MCS-P3008-0712, Argonne National Laboratory, July 2012. (116)

[48] D. Callahan, B.L. Chamberlain, and H.P. Zima. The Cascade High Productivity Language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, 2004. (32)

[49] Alessandra Carbone. *Pattern Formation in Biology, Vision and Dynamics*. World Scientific, 2000. (106)

[50] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *International Journal High Performance Computing Applications*, 21(3):291–312, August 2007. (30, 32)

[51] Bradford L. Chamberlain, Sung eun Choi, Steven J. Deitz, and Lawrence 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*, P-PHEC '04, 2004. (55)

[52] C. Chen, R.C. Beardsley, and G. Cowles. An unstructured grid, finite-volume coastal ocean model (FVCOM) system. *Oceanography*, 19(1):78–89, 2006. (1)

[53] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: a parallel DSL for image analysis and visualization. In *Proceedings of the*

33rd ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI '12, pages 111–120, New York, NY, USA, 2012. ACM. (30)

[54] Hui-Hsien Chou and James A. Reggia. Emergence of self-replicating structures in a cellular automata space. *Physica D: Nonlinear Phenomena*, 110(34):252 – 276, 1997. (106)

[55] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Journal of Computer Science*, 26:205–210, June 2011. (6, 30, 37, 38, 43, 44, 45, 46, 47, 48, 49, 50)

[56] P Colella, D T Graves, N D Keen, T J Ligocki, D F Martin, P W Mccorquodale, D Modiano, P O Schwartz, T D Sternberg, and B Van Straalen. Chombo Software Package for AMR Applications: Design Document. Technical report, Lawrence Berkeley National Laboratory, 2009. (6, 37, 38, 42, 43, 45, 47, 48, 49)

[57] Phil Colella. Defining software requirements for scientific computing. Slide of 2004 presentation included in David Pattersons 2005 talk, URL: `http://www.lanl.gov/orgs/hpc/salishan/salishan2005/davidpatterson.pdf`, 2004. (38)

[58] WR Collins and KW Miller. Defining and implementing Fortran generic abstract data types. *Information and Software Technology*, 33(4):281 – 291, 1991. (113)

[59] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005. (30)

[60] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys*, 32(1es), March 2000. (32)

[61] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Gluck, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries, 1998. (6, 31)

[62] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998. (30, 31, 32, 48)

[63] P.J. Darwen and D.G. Green. Viability of populations in a landscape. *Ecological Modeling*, 85(23):165 – 171, 1996. (106)

[64] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, February 2009. (115)

[65] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of 2008 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC08, 2008. (115)

[66] E. F. D'Azevedo, V. L. Eijkhout, and C. H. Romine. Conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical Report 56, LAPACK Working Note, August 1993. (19)

[67] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004. (33)

[68] Daniel C. Dennett. Artificial life as philosophy. *Artificial Life*, 1(3):291–292, 1994. (106)

[69] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat 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. (6, 30, 37, 38, 42, 43, 44, 45, 46, 48, 49, 50)

[70] Chen Ding and Ken Kennedy. Inter-array data regrouping. In *Twelfth International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1999. (114)

[71] J. J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Software - Practice and Experience*, 9:219–226, 1979. (115)

[72] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The international exascale software project roadmap. *International Journal on High Performance Computing Applications*, 25(1):3–60, February 2011. (38)

[73] Q. Du, M. Gunzburger, and L. Ju. Voronoi-based finite volume methods, optimal Voronoi meshes, and PDEs on the sphere. *Computer Methods in Applied Mechanics and Engineering*, pages 3933–3957, 2003. (43)

[74] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004. (30)

[75] Niandong Fang and Helmar Burkhart. Structured parallel programming using MPI. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, HPCN 1996, pages 840–847, London, UK, 1996. Springer-Verlag. (30, 48)

[76] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1,12, 2008. (16)

[77] Luciano Floridi. *The Blackwell Guide to the Philosophy of Computing and Information*. Wiley-Blackwell, 1st edition, 2004. (106)

[78] M. Gardner. The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, October 1970. (10, 106)

[79] Al Geist and Robert Lucas. Major computer science challenges at exascale. *International Journal High Performance Computing Applications*, 23(4):427–436, November 2009. (38)

[80] 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. (6, 30, 37, 38, 42, 43, 44, 45, 47, 48, 49, 50)

[81] Tom Goodale, Gabrielle Allen, Joan Lanfermann, Gerd Masso, Thomas Radke, Edward Seidel, and John Shalf. The Cactus framework and toolkit: Design and applications. In *Proceedings of the 5th International Conference on High Performance Computing for Computational Science*, HPSC 2012, pages 197–227, 2003. (6, 37, 43, 45, 47, 48, 49)

[82] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *In IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 106–111. IEEE Computer Society Press, 1998. (33)

[83] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 39–52, New York, NY, USA, 1999. ACM. (31)

[84] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. (116)

[85] Ross Heikes, Joon-Hee Jung, Konor C.S., and Randall D. Continuing development of models based on spherical geodesic grids. Department of Atmospheric Science at Colorado State University; `http://kiwi.atmos.colostate.edu/rr/groupPIX/ross/pde_sphere_2007_heikes.pdf`, 2007. (27)

[86] Ross Heikes and David 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. (2, 12, 54, 106)

[87] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009. (13, 20)

[88] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952. (23)

[89] G.M. Hopper. Automatic coding for digital computers. In *High Speed Computer Conference*. Remington Rand Incorporated, 1955. (37)

[90] Tim J. Hutton. Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, 8(4):341–356, 2002. (106)

[91] IEEE and The Open Group. *Make*, 6th edition, 2004. (38)

[92] Elizabeth R. Jessup, Carolyn J. C. Schauble, and Gitta Domik. *Introduction to High-Performance Scientific Computing*. The MIT Press, 1996. (2)

[93] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories Murray Hill, New Jersey 07974, 1979. (38)

[94] P. Jones. Parallel Ocean Program (POP) user guide. Technical Report LACC 99-18, Los Alamos National Laboratory, March 2003. (1, 16, 23, 85, 118, 119)

[95] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel Distributed Processing*, IPDPS '10, pages 1–12, 2010. (115)

[96] Shoaib Kamil, Cy Chan, Samuel Williams, Leonid Oliker, John Shalf, Mark Howison, and E. Wes Bethel. A generalized framework for auto-tuning stencil computations. In *In Proceedings of the Cray User Group Conference*, CUG '09, 2009. (6, 30, 37, 42, 43, 44, 45, 46, 47, 48, 49)

[97] Dave Kennison. Grids that NCAR graphics can contour (non-uniform grids). `http://ngwww.ucar.edu/grids.html`, 2013. (6, 11)

[98] Kurt Keutzer and Tim Mattson. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, June 2009. (14)

[99] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming*, ECOOP '97. SpringerVerlag, 1997. (35, 82)

[100] Alan Kirman. The economy as an evolving network. *Journal of Evolutionary Economics*, 7(4):339–353, 1997. (106)

[101] C.H. Koelbel, D. B. Loveman, R. S. Schreiber, L. Steele Jr. Guy, and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, September 1996. (56)

[102] Markus Kowarschik and Christian Wei. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies — Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*, pages 213–232. Springer, 2003. (114)

[103] Christopher D. Krieger, Andrew Stone, and Michelle Mills Strout. Mechanisms that separate algorithms from implementations for parallel patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, March 2010. (9, 15)

[104] Sriram Krishnamoorthy, Muthu Baskaran, and Uday Bondhugulaj Ramanujam. Effective automatic parallelization of stencil computations. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, PLDI 2007, 2007. (115)

[105] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. (114, 115)

[106] Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Notes*, 35(1):109–122, December 1999. (31)

[107] Eugene Loh. The ideal HPC programming language. *Queue*, 8(6):30:30–30:38, June 2010. (30)

[108] Chi-keung Luk, Bradley C Kuszmaul, and Charles E Leiserson. The Pochoir stencil compiler. *Artificial Intelligence*, 36:117–128, 2011. (6, 30, 37, 38, 42, 43, 44, 45, 46, 47, 48, 49, 50, 115)

[109] Ewing Lusk, Steven Pieper, and Ralph Butler. More scalability, less pain. *SciDAC Review*, 17, 2010. (29)

[110] Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *Parallel Processing Letters*, 17(1):89–102, 2007. (13, 30)

[111] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC06 Conference Tutorial*. IEEE, Novemeber 2006. (13)

[112] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi 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. (6, 30, 37, 38, 42, 43, 44, 45, 46, 47, 48, 49, 50)

[113] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore compilation strategies and challenges. *Signal Processing Magazine*, 26(6):55–63, 2009. (115)

[114] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Compututer Surveys*, 37(4):316–344, December 2005. (38)

[115] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 293–304, New York, NY, USA, 2006. ACM. (106)

[116] Ross J Murray. Explicit generation of orthogonal grids for ocean models. *Journal of Computational Physics*, 126(2):251–273, 1996. (13, 55, 56)

[117] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. (30, 48)

[118] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. (30, 32)

[119] Robert W. Numrich and John Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24:4–17, August 2005. (32)

[120] Chuck Pheatt. Intel threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, April 2008. (30)

[121] William M. Putman and Shian-Jiann Lin. Finite-volume transport on various cubed-sphere grids. *Journal of Computational Physics*, 227(1):55–78, November 2007. (11, 12)

[122] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers*, volume 10 of *CPC2000*. Springer Verlag, January 2000. (82)

[123] D.A. Randall, T.D. Ringler, R.P. Heikes, P. Jones, and J. Baumgardner. Climate modeling with spherical geodesic grids. *Computing in Science Engineering*, 4(5):32–41, 2002. (12, 106)

[124] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburg, PA, May 1998. (30, 33)

[125] Paul Rendell. Turing universality of the game of Life. In *Collision-Based Computing*, pages 513–539. Springer London, 2002. (106)

[126] R. K. Rew and G. P. Davis. The unidata netCDF: Software for scientific data access. In *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, pages 33–40, Anaheim, California, American Meteorology Society, February 1990. (19)

[127] Gerald Roth, Gerald Roth, John Mellor-crummey, John Mellor-crummey, Ken Kennedy, Ken Kennedy, R. Gregg Brickner, and R. Gregg Brickner. Compiling stencils in High Performance Fortran. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*. ACM Press, 1997. (1)

[128] Robert Sadourny. Conservative finite-difference approximations of the primitive equations on quasi-uniform spherical grids. *Monthly Weather Review*, 100(2):136–144, February 1972. (54)

[129] Paul S. Schopf. Poseidon ocean model tripolar grid. `http://climate.gmu.edu/poseidon/code/Tripole.html`, 2005. (6, 11, 12)

[130] L.S. Schulman and P.E. Seiden. Statistical mechanics of a dynamical system based on Conway's game of Life. *Journal of Statistical Physics*, 19(3):293–314, 1978. (106)

[131] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag. (38)

[132] Zhen-Jun Shi and Jinhua Guo. A new family of conjugate gradient methods. *Journal of Computational and Applied Mathematics*, 224(1):444–457, February 2009. (23)

[133] Rudy Slingerland and Lee Kump. *Mathematical Modeling of Earth's Dynamical Systems: a Primer*. Princeton University Press, 2011. (1)

[134] R. D. Smith and S Kortas. Curvilinear coordinates for global ocean models. Technical report, Los Alamos National Laboratory, LA-UR-95-1146, 1995. (54)

[135] Andrew Stone, John Dennis, and Michelle Mills Strout. The CGPOP miniapp, version 1.0. Technical Report Technical Report CS-11-103, Colorado State University, July 1 2011. (5, 8, 14, 23, 58, 85)

[136] Andrew Stone and Michelle Mills Strout. Programming abstractions to separate concerns in semi-regular grids. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, June 2013. (9)

[137] Andrew I. Stone, John M. Dennis, and Michelle Mills Strout. Evaluating Coarray Fortran with the CGPOP miniapp. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, October 15, 2011. (5, 8, 14, 16, 85)

[138] Andrew I. Stone, Steven DiBenedetto, Michelle Mills Strout, and Daniel Massey. Scalable simulation of complex network routing policies. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, May 17-19, 2010. (30)

[139] Andrew I. Stone, Steven DiBenedetto, Michelle Mills Strout, and Daniel Massey. Simulating internet scale topologies with metarouting. Technical report, Technical Report CS-10-103 Colorado State University, March 2010. (30)

[140] Michelle Mills Strout, Christopher Krieger, Andrew Stone, Christopher Wilcox, John Dennis, and James Bieman. Evaluating the separation of algorithm and implementation within existing programming models. In *Proceedings of SciDAC*, 2011. (8)

[141] Peter Su and Robert L. Scot Drysdale. A comparison of sequential Delaunay triangulation algorithms. *11th ACM Symposium on Computational Geometry*, 7(56):361 – 385, 1997. (43)

[142] Xin Sui, Donald Nguyen, Martin Burtscher, and Keshav Pingali. Parallel graph partitioning on multicore architectures. In *Languages and Compilers for Parallel Computing (LCPC)*, 2010. (13)

[143] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. OptiML: an implicitly parallel domainspecific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, 2011. (30)

[144] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. (30, 49)

[145] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC programmability using classroom studies. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 10:1–10:7, New York, NY, USA, 2009. ACM. (13)

[146] J.W. Thomas. *Numerical Partial Differential Equations: Finite Difference Methods*. Springer, 1995. (1)

[147] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM. (31)

[148] Hirofumi Tomita, Motohiko Tsugawa, Masaki Satoh, and Koji Goto. Shallow water model on a modified icosahedral geodesic grid by using spring dynamics. *Journal of Computational Physics*, 174(2):579–613, 2001. (2, 54)

[149] Didem Unat, Xing Cai, and Scott B Baden. Mint. In *Proceedings of the International Conference on Supercomputing*, ICS '11, page 214, New York, New York, USA, 2011. ACM Press. (30)

[150] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM. (6, 37, 38, 42, 43, 44, 45, 47, 48, 49, 50)

[151] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. (29)

[152] W3C. *HTML 4.01 Specification*, 1999. (38)

[153] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics. (114)

[154] Zoë J. Wood, Peter Schröder, David Breen, and Mathieu 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. (116, 117)

[155] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *In ACM*, pages 10–11, 1998. (30)

[156] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM. (116)

Appendix A

Notation

The following appendix lists notation used in Chapters 5 and 6. The reader may find it useful to refer to this appendix when reading those chapters. Note that we use Greek letters to represent abstractions (tuples) and functions that return abstractions; we use lowercase letters to represent identifiers, indices, vectors, matrices, and functions that return identifiers, indices, vectors or matrices; and we use uppercase letters to represent sets and lists, and functions that return sets or lists.

Subgrids

σ	A subgrid $\sigma = (s, n, m)$ where s is a value identifying a subgrid, n is the width of the subgrid and m is the height of the subgrid. **(Equation 5.1)**
$I(\sigma)$	Function to calculate index set of a subgrid (indices are in the grid domain). **(Equation 5.2)**
$H(\sigma)$	Function to calculate set of indices for halo of a subgrid (indices are in the grid domain). **(Equation 5.3)**
$B(\sigma)$	Function to calculate set of indices for border of a subgrid (indices are in the grid domain). **(Equation 5.4)**

Neighborhoods

ν	A neighbor $\nu = \langle i, j \rangle$. **(Equation 6.8)**
N	A neighborhood, which is a list of neighbors ordered in a clockwise fashion. **(Equation 6.9)**
$r\left(N, \nu, n\right)$	Rotation function: given a neighbor ν return the neighbor that is visited by doing n clockwise rotations in the neighborhood N. **(Equation 6.11)**
$I_{lcl}\left(\langle i, j \rangle, d\right)$	A depth-d domain of local indices for a stencil operation centered at point $\langle i, j \rangle$.**(Equation 6.1)**

Border mappings

β	A border map $\beta = (\rho_{tgt}, \rho_{src})$. **(Equation 5.12)**
β'	A border map cognizant of rotation (used in Chapter) $\beta = (\rho_{tgt}, \rho_{src}, n)$ where n is specifies how much neighbors to rotate through in some neighborhood N where $0 \leq n \leq \lvert N \rvert$. **(Equation 6.12)**
ρ_{tgt},ρ_{src}	Target and source rectangles of a mapping. **(Equations 5.5 and 5.6)** $\rho_{tgt} = (s_1, p_i, p_j, q_i, q_j)$ $\rho_{src} = (s_2, v_i, v_j, w_i, w_j)$
$o(\rho)$	Orientation of some rectangle ρ. **(Equation 5.10)**
$m(\beta, s, \langle i, j \rangle)$	A border mapping function. The function uses β to determe what some point $(s, \langle i, j \rangle)$ in a subgrid's halo should map to. **(Equation 5.12)**

Grids

γ	A grid, $\gamma = (S, B)$. **(Equation 5.14)**
S	A set of subgrids.
B	A set of border maps**.**
$I\left(\gamma\right)$	A function that calculates the indices in grid γ. **(Equation 5.15)**

Decompositions

D A decomposition, which is a set of blocks.

θ A block $\theta = (g, l, p, s, \langle u_i, u_j \rangle, \langle v_i, v_j \rangle)$ where g is a global block identifier, l is a local block identifier, p is a process identifier (MPI rank), s, is a subgrid identifier, and $\langle u_i, u_j \rangle$and$\langle u_i, u_j \rangle$are points in subgrid s. **(Equation 5.16)**

θ' Updated version of a block (from Section 6.4 on). $\theta = (g, l, p, s, \langle u_i, u_j \rangle, \langle v_i, v_j \rangle, G)$. **(Equation 6.13)**

G A list of ghost-nodes for the block.

$I(\theta)$ A function that calculates the global indices of a block θ. **(Equation 6.3)**

$I_{lcl}(\theta)$ A function that calculates the local indices of a block θ. **(Equation 6.4)**

$H_{lcl}(\theta, d)$ A function that calculates the local indices of a halo of depth d around block θ. **(Equation 6.6)**

$t(s, \langle i, j \rangle, G)$ Translate a grid index $(s, \langle i, j \rangle)$ into its position in the list G. **(Equation 6.5)**

Communication plans for halos

π A communication plan to populate halos, $\pi = (M_r, M_s)$where M_r is a set of receiving messages and M_s is a set of sending messages. **(Equation 5.18)**

μ A single message to populate halos $\mu = (l, \rho, p)$ where l is a local block identifier (that contains data to send if $\mu \in M_s$ or data to receieve if $\mu \in M_R$, p is a process ID, and ρ is a rectangle contained either contained in $i(\theta)$ if $\mu \in M_s$ or $h(\theta, d)$ if $\mu \in M_R$. **(Equation 5.19)**

Communication plans for ghost lists

$\pi'(p)$	A communication plan to populate ghost lists, $\pi(p) = (M'_r,' M_s)$where M'_r is a set of receiving messages for processor p and M'_s is a set of sending messages for processor p. **(Equation 6.15)**
μ'	A single message in M'_r, or M'_s to populate halos $\mu = (p, G)$ where p is the processor the message is being received from or sent to, and G is the list of ghost nodes that are transfered. **(Equation 6.14)**
$m(\gamma, \theta, \langle i, j\rangle, \langle i', j'\rangle)$	Maps a local index $\langle i, j\rangle$ in the halo $h_{lcl}(\theta, d)$ for a block θ to a global index in grid γ. **(Equation 6.7)**

Data Objects

δ	A data object $\delta = (D, d)$. **(Equation 5.17)**

Appendix B

Application programming interface

In this section we list routines (subroutines and functions) that exist in the GridWeaver library. Using these routines users may define grids and construct and operate on data objects.

Initialization routines

The following routines initialize and configure GridWeaver.

`gridweaver_initialize()`

Initialize the GridWeaver library. GridWeaver requires that this subroutine be executed on all MPI ranks exactly once prior to invoking any other subroutine or function in this API.

`turnOffSyntaxHighlighting()`

GridWeaver routines that output to stdout use syntax highlighting to distinguish different sorts of text, for example the name of a property versus its value. Although syntax highlighting helps programmers scan this text it may not be diserable in some cases, for example if the text is piped to a file. In such cases syntax highlighting may be turned off by calling this subroutine.

Subgrid routines

Subgrids represent two-dimensional index space. Subgrids can be connected to one another to form a grid. Subgrids are environmental objects, that is they have a unique identifier and are exportable and importable from/to files. The following routines construct and query data about subgrids.

subgrid_new(name, width, height)

This function initalizes and returns a new subgrid of a given **width** and **height**. GridWeaver will store the subgrid in the global object environment using the specified **name**. To instantiate a new subgrid first declare it, for example:

```
type(SubGrid) ::  sg,
```

then assign the subgrid the result from calling this function, for example:

```
sg = subgrid_new("sg", 10, 10).
```

subgrid_width(sg)

Return how many elements span the x dimension of subgrid **sg**.

subgrid_height(sg)

Return how many elements span the t dimension of subgrid **sg**.

Neighborhood routines

Neighborhoods define the internal connectivity of nodes within a subgrid.

neighbor_new(name, dx, dy)

For grids that use non-compact stencils it is necessary to have information about the connectivity pattern that exists within each subgrid. To do this the user may specify what edges surround each point with this function. This function specifies that for each point within a sugrid it should have an edge to a neighboring point located at an index **dx** points from the given point in the x direction and **dy** points from a given point in the y direction. Note that **dx** and **dy** should be integer values between -1 and 1 (inclusive).

Grid routines

Grids are collections of subgrids that connect to one another via border mappings. Grids are environmental objects, that is they have a unique identifier and can be exported and

imported from files. The following routines construct grids, connect subgrids to one another via border mappings, and query information about grids.

grid_new(name)

This function initalizes and returns a new grid. GridWeaver will register the grid in the bal object environment with the specified **name**. To instantiate a new grid first declare it, for example:

```
type(Grid) ::  g,
```

then assign the grid the result from calling this function, for example:

```
g = grid_new("g").
```

grid_addSubgrid(g, sg)

Specify that subgrid **sg** should be included in grid **g**.

grid_addBorder(g, srcX1, srcY1, srcX2, srcY2, srcSG
tgtX1, tgtY1, tgtX2, tgtY2, tgtSG, rotation)

Add a connectivity relationship between two subgrids. Specifically, create a border mapping between the two rectangular regions (**srcX1**, **srcY1**) to (**srcX2**, **srcY2**) in subgrid **srcSG** and (**srcX1**, **srcY1**) to (**srcX2**, **srcY2**) in subgrid **srcSG**. The source rectangle should be in the halo region of subgrid **srcSG** , and the target rectangle should be in the border region of subgrid **tgtSG**. Note that the two rectangular regions must be the same shape (although they may be rotated, that is either the width and height of each reactangle are the same or the width of one rectangle is the height of the other and the height is the width of the other). The **rotation** value specifies how edges traversing from one subgrid to the next should rotate.

grid_numSubGrids(g)

This function returns the number of subgrids included in grid **g**. Subgrid ID's (SGIDs) range between [1, numSubgrids()].

grid_getSubgrid(g, sgid)

This function returns the subgrid with the specified ID. Subgrid ID's (SGIDs) range between [1, numSubgrids()].

grid_twist(g, cell1X, cell1Y, cell1SG,
cell2X,cell2Y,cell2SG, n, resX,resY,resSG)

The twist function takes two adjacent cells and a twist amount n and returns the location of the node that is visited by twisting clockwise n times.

Distribution routines

Distributions define how to allocate grid data across multiple processors. We assume that processors are indexed using MPI ranks and ranks span values in the inclusive range: [0, numRanks]. Distributions are environmental objects, that is they have a unique identifier and can be exported and imported from files. The following functions construct, modify, and query distributions.

distribution_new(name)

This function initalizes and returns a new distribution. Gridweaver will register the distribution in the global object environment with the specified **name**. To instantiate a new distribution first declare it, for example:

```
type(Distribution) ::  dist,
```

then assign the grid the result from calling this function, for example:

```
g = grid_new("dist").
```

distribution_applyFillBlock(dist, g, blkH)

Calculate a fill-block distribution, storing the result in distribution **dist**, for grid **g** where the height of each block is **blkH**. We illustrate a fill-block distribution in Figure 5.3. The calculated distribution will use all available MPI ranks.

distribution_applyBlockFill(dist, g, blkW)

Calculate a block-fill distribution, storing the result in distribution **dist**, for grid **g** where the width of each block is **blkW**. We illustrate a block-fill distribution in Figure 5.3. The calculated distribution will use all available MPI ranks.

distribution_applyBlockCyclic(dist, g, blkW, blkH)

Calculate a block-cyclic distribution, storing the result in distribution **dist**, for grid **g** where the width of each block is **blkW** and the height of each block is **blkH**. We illustrate a block-cyclic distribution in Figure 5.3. The calculated distribution will use all available MPI ranks.

distribution_applyBlankDist(dist, g, blkW, blkH)

Initialize a distribution, storing the result in **dist**, and initializing each blocks so that no process owns it. Each block will have a width of **blkW** and height of **blkH**. In order for the distribution to be of any use the user will have to modify it by repeatedly calling the `distribution_setProcForBlock` function.

distribution_setProcForBlock(dist, gbid, rank)

For distribution **dist** assign an MPI **rank** to a given block **gbid**. Note that **rank** must be in the interval [0, `numTasks()`] and gbid must be between [1, `distribution_numBlocks(dist)`].

distribution_gbidAt(dist, sg, x, y)

Return the global block ID (gbid) at a specified (**x**, **y**) coordinate in subgrid **sg**. The resulting value will be in the interval [1, `distribution_numBlocks(dist)`].

distribution_visualize(dist, dirName)

Visualize the distribution **dist**. GridWeaver will store the resulting visualization in a series of image and HTML files under the directory **dirName**.

distribution_width(dist)

For a given distribution **dist** return the width of its blocks.

distribution_height(dist)

For a given distribution **dist** return the height of its blocks.

distribution_numLocalBlocks(dist)

For a given distribution **dist** return the number of blocks that the MPI rank calling the function contains.

distribution_numBlocks(dist)

For a given distribution **dist** return the number of blocks it globally contains.

distribution_lbid2gbid(dist, lbid)

In distribution **dist** given a local block ID **lbid** return its corresponding global block ID.

distribution_gbid2lbid(dist, gbid)

In distribution **dist** given a gobal block ID **gbid** return its corresponding local block ID.

distribution_gbid2proc(dist, gbid)

In distribution **dist** given a global block ID **gbid** return the MPI rank that associated with the block.

distribution_blockLowX(dist, gbid)

In distribution **dist** given a global block ID **gbid** return the lowest X coordinate associated with the block.

distribution_blockLowY(dist, gbid)

In distribution **dist** given a global block ID **gbid** return the lowest Y coordinate associated with the block.

distribution_blockHighX(dist, gbid)

In distribution **dist** given a global block ID **gbid** return the largest X coordinate associated with the block.

distribution_blockHighY(dist, gbid)

In distribution **dist** given a global block ID **gbid** return the largest Y coordinate associated with the block.

distribution_pos2BlockPos(dist, x, y, sg, blkX, blkY, gbid)

For a given distribution **dist** determine what block contains the coordinate (**x**, **y**) in subgrid **sg**. The function stores the coordinates of the resulting block in **blkX** and **blkY**, and the block's global ID in **gbid**.

Communication plan routines

Data objects use communication plans to determine how to populate halos.

schedule_new(name)

This function initalizes and returns a new schedule. Gridweaver will register the schedule in the global object environment with the specified **name**. To instantiate a new schedule first declare it, for example:

```
type(Schedule) ::  sched,
```

then assign the grid the result from calling this function, for example:

```
sched = schedule_new("sched").
```

schedule_calculate(sched, dist, depth)

Calculate a schedule **sched** for stencil computations of a specified depth for a given distribution **dist**. Data objects can use this schedule to perform communication.

Data object routines

Data objects contain values that associate with each cell of a grid. Data objects use distribution objects to determine how to distribute data and schedule objects to determine how to communicate; users can apply operations to update the values in data objects.

data_new(sched)

This function initalizes and returns a new data object that will communicate using the schedule **sched**. To instantiate a new data object first declare it, for example:

```
type(DataObj) ::  data,
```

then assign the grid the result from calling this function, for example:

```
data = data_new(sched).
```

data_input(data, fileID)

Read values for a data object from the file with the Fortran ID **fileID**.

data_output(data, fileID)

Write values for a data object to the file with the Fortran ID **fileID**.

data_print(data, fileID)

Output the contents of the data object to the file with the Fortran ID **fileID** in a human-readable format. Note that in Fortran stdout always has a **fileID** of 6.

data_printForProcs(data, out)

Output the values for all local blocks in data object **data** for each processor contains independently. Values will be output to the file with the Fortran ID **fileID** in a human-readable format. Note that in Fortran stdout always has a **fileID** of 6.

data_printForProc(data, proc, out)

Output the values for all local blocks in data object **data** for processor **proc**. Values will be output to the file with the Fortran ID **fileID** in a human-readable format. Note that in Fortran stdout always has a **fileID** of 6.

data_apply(data, in1, func) ...
data_apply(data, in1, in2, ..., in10, func)

Apply a stencil function **func** that will update values in data object **data** using input from data objects **in1** through **in10**.

data_applyNC(data, in1, func) ...
data_applyNC(data, in1, in2, ..., in10, func)

Apply a non-compact stencil function **func** that will update values in data object **data** using input from data objects **in1** through **in10**.

data_sum(data)

Calculate and return the sum of all values stored in the data object **data**.

data_initializeSeqVals(data)

Initialize values in data object starting from 1 and increasing sequentially. Users may find it useful to initialize data like this for debugging purposes.

data_forceUpdate(data)
data_forceUpdate(data1, data2, ..., dataN)]

update halos and ghost cells for the specified data object(s).

Environment routines

The GridWeaver model includes an environment of objects such as subgrids, grids, and distributions. Users may import/export these objects to files. The following functions input and output the environment of objects to files or present the environment to the user by printing it to `stdout`.

environment_print(fileID)

Output the contents of GridWeaver's environment to the file with the Fortran ID **fileID** in a human-readable format. Note that in Fortran stdout always has a **fileID** of 6.

environment_output(filename)

Output the environment to a file with a given **filename** in a binary format.

environment_input(filename)

Input the environment to a file with a given **filename** in a binary format.

environment_getGrid(name)

Return the instance of the grid stored in the environment using the identifier **name**.

environment_getSubgrid(name)

Return the instance of the subgrid stored in the environment using the identifier **name**.

environment_getDistribution(name)

Return the instance of the distribution stored in the environment using the identifier **name**.

environment_getSchedule(name)

Return the instance of the schedule grid stored in the environment using the identifier **name**.

environment_clear()

Remove all objects from the environment.

Appendix C

Supplementary figures

This appendix includes Figures, which the reader may find useful, but do not properly fit in any chapter. Specifically, this appendix includes the following figures:

- **Cell-based representation of mini-icosahedral grid (Figure C.1)** — a representation of the mini-icosahedral grid from Figure 6.1. Readers can print, cut, and fold this figure into a 3D model of the grid.
- **Neighbors of node 3e in cell representation of mini-icosahedral grid (Figure C.2)** — an illustration of what neighbors can be visited at various depths around element 3e of the mini-icosahedral grid.
- **Ghost nodes accessible from a subgrid in icosahedral grid (Figure C.3)** — illustrates the nodes visited when performing a breadth first search from a subgrid on an icosahedral grid.
- **Code comparing direct vs indirect access to stencil data (Figure C.4)** — we use this code to generate data for Figure 4.4. It compares the performance of a three point stencils on one-dimensional data that accesses data directly against a version that accesses data through a randomly permuted index array.
- **Code to measure cost of function overhead (Figure C.5)** — we use this code to generate data in Figure 7.3. It compares the performance of a three point stencil on one-dimensional data that accesses data directly against a version that accesses data via a function.
- **Code for GridWeaver's stencil application routine (Figure C.6)** — is code used within GridWeaver's runtime library to apply stencil operations on data objects. This figure demonstrates the library overhead GridWeaver introduces.

- **Code for Conway's game of life on the mini-icosahedral grid (Figure C.7)** — A complete code listing that implements Conway's game of life on an icosahedral grid using GridWeaver.

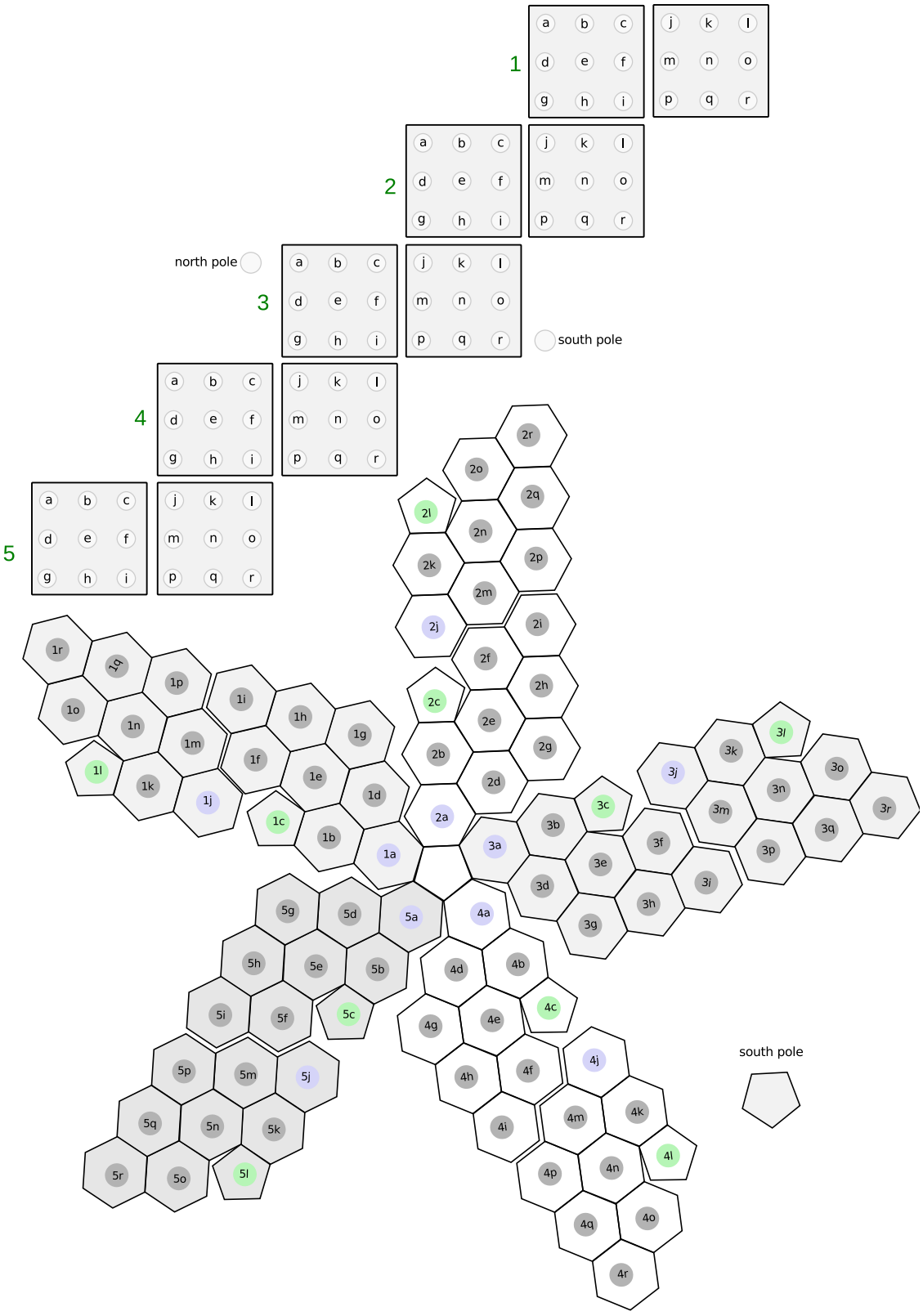


Figure C.1: Cell-based representation of mini-icosahedral grid.

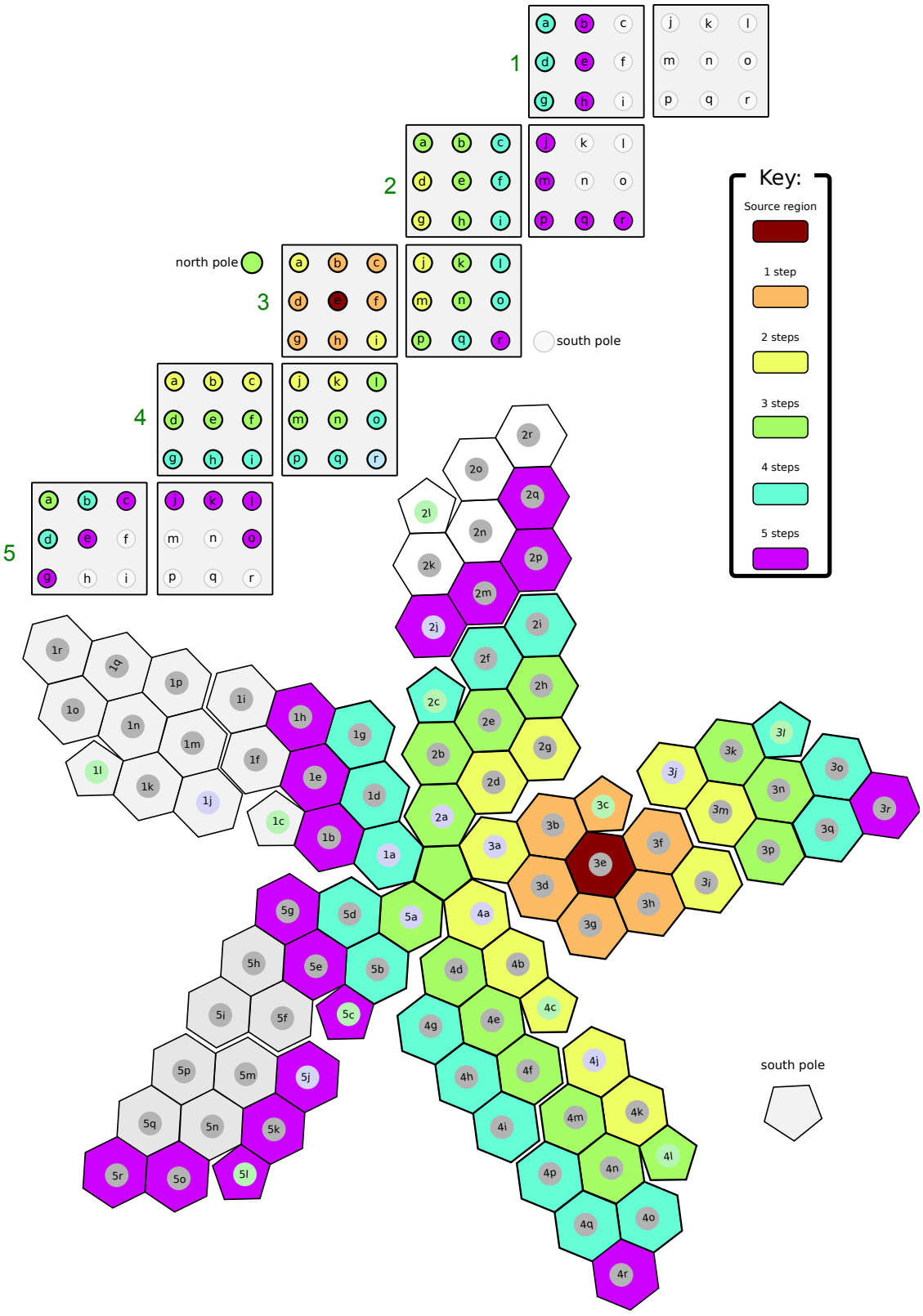


Figure C.2: Neighbors around node 3e in cell-based representation of mini-icosahedral grid.

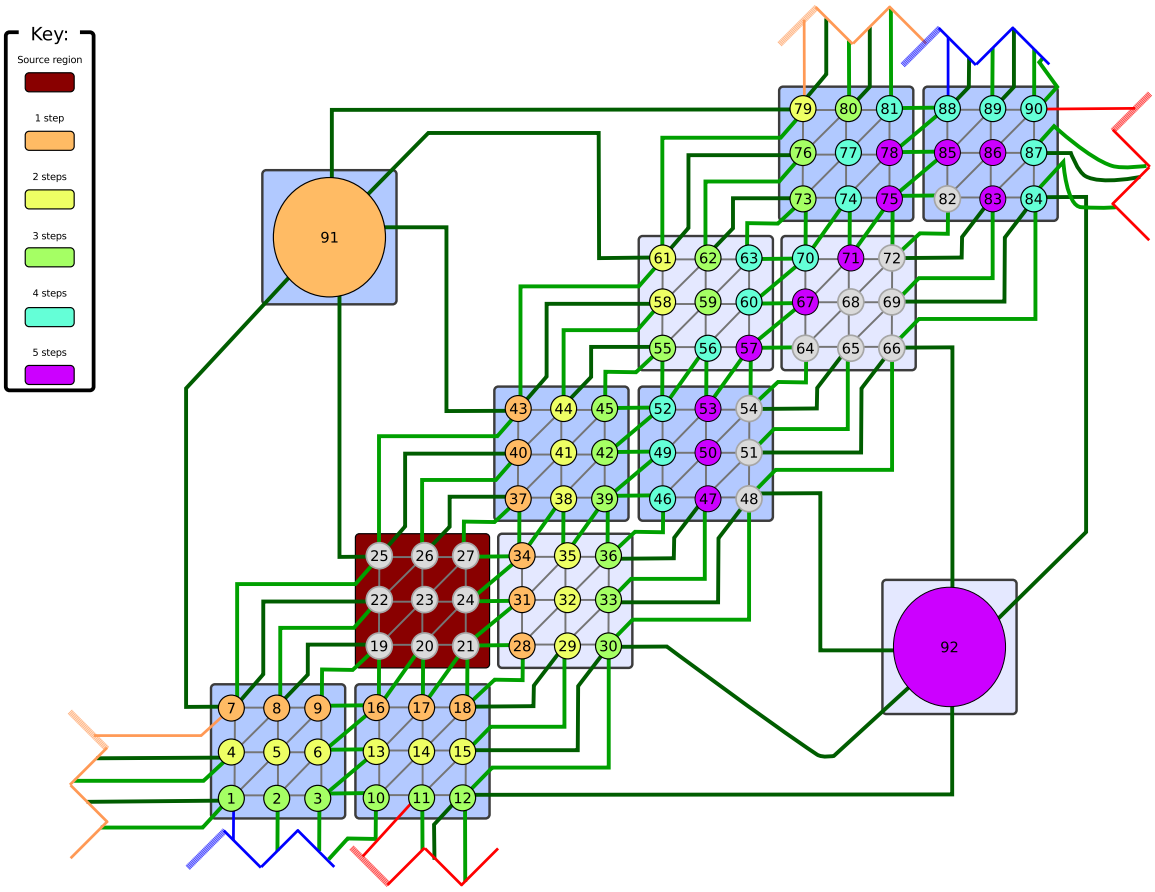


Figure C.3: Ghost nodes accessible from a subgrid in icosahedral grid

```c
#include <stdio.h>
#include <time.h>

#define MAX_SIZE    1000000
double A[MAX_SIZE];   int IDX[MAX_SIZE];
#define NUM_TESTS   7
const long TEST[] = {1e6, 5e6, 1e7, 5e7, 1e8, 5e8, 1e9};

int main(int argc, char *argv[]) {
  clock_t startTime; double sum, tmp; int rndIdx1,rndIdx2,i,j;

  // Fill IDX with indices ranging in the interval [0..MAX_SIZE)
  for(i = 0; i < MAX_SIZE; i++) { IDX[i] = i; }

  // Randomly swap elements in A
  srand(time(NULL));
  for(i = 0; i < MAX_SIZE; i++) {
    rndIdx1 = rand() % MAX_SIZE;  rndIdx2 = rand() % MAX_SIZE;
    tmp = IDX[rndIdx1];
    IDX[rndIdx1] = IDX[rndIdx2];  IDX[rndIdx2] = IDX[rndIdx1];
  }

  // Print table header
  printf("TEST-SIZE␣\tDIRECT␣(sec)\tINDIRECT␣(sec)\n");
  printf("-----------------------------------------------\n");

  // Run experiments using sizes in SIZES array
  for(i = 0; i < NUM_TESTS; i++) {
    // Access data directly
    startTime = clock();
    for(j = 1; j < TEST[i] - 1; j++) {
      A[j % MAX_SIZE] = (A[(j-1) % MAX_SIZE] + A[(j  ) % MAX_SIZE] +
                         A[(j+1) % MAX_SIZE]) / 3.0;
    }
    printf("%10d:\t", TEST[i]);
    printf("%4.2f\t", (double)(clock() - startTime) / CLOCKS_PER_SEC);

    // Access data through an index array
    startTime = clock();
    for(j = 1; j < TEST[i] - 1; j++) {
      A[IDX[j % MAX_SIZE]] = (A[IDX[(j-1) % MAX_SIZE]] +
                              A[IDX[(j  ) % MAX_SIZE]] +
                              A[IDX[(j+1) % MAX_SIZE]]) / 3.0;
    }
    printf("%4.2f\n", (double)(clock() - startTime) / CLOCKS_PER_SEC);
  }

  return 1;
}
```

Figure C.4: Code that measures cost of accessing data directly vs through an index array. We used this code to generate the data in Figure 4.4.

```c
#include <stdio.h>
#include <time.h>

#define MAX_SIZE    100000
double A[MAX_SIZE];
#define NUM_TESTS   7
const long TEST[] = {1e6, 5e6, 1e7, 5e7, 1e8, 5e8, 1e9};

double getValue(int idx) __attribute__ ((noinline));
double getValue(int idx) { return A[idx % MAX_SIZE]; }

double setValue(int idx, double val) __attribute__ ((noinline));
double setValue(int idx, double val) { A[idx % MAX_SIZE] = val; }

int main(int argc, char *argv[]) {
  clock_t startTime;
  double sum;
  int i,j;

  // Print table header
  printf("TEST-SIZE \tDIRECT (sec)\tINDIRECT (sec)\n");
  printf("-----------------------------------------------\n");

  // Run experiments using sizes in SIZES array
  for(i = 0; i < NUM_TESTS; i++) {
    printf("%10d:\t", TEST[i]);

    // Access data directly
    startTime = clock();
    for(j = 1; j < TEST[i] - 1; j++) {
      A[j % MAX_SIZE] = (A[(j-1) % MAX_SIZE] +
                         A[(j  ) % MAX_SIZE] +
                         A[(j+1) % MAX_SIZE]) / 3.0;
    }
    printf("%4.2f\t", (double)(clock() - startTime) / CLOCKS_PER_SEC);

    // Access data through function
    startTime = clock();
    for(j = 1; j < TEST[i] - 1; j++) {
        setValue(j, (getValue(j-1) + getValue(j) + getValue(j+1)) / 3.0);
    }
    printf("%4.2f\n", (double)(clock() - startTime) / CLOCKS_PER_SEC);
  }

  return 1;
}
```

Figure C.5: Code that measures cost of access of data directly vs through non-inlined function. We used this code to generate the data in Figure 7.3.

```
1  ! This function impersonates as an array; it stores data in the array
2  ! pointed to by gInputData1 and reindexes a location <x,y> into an index
3  ! within a block.
4  real(8) function AFunc1(x,y)
5      integer, intent(in) :: x,y
6      AFunc1 = gInputData1%vals(x-blkXOffset, y-blkYOffset, currentLbid)
7  end function
8
9  ! Apply the stencil function func; store results into the data self,
10 ! and use values from the data object in1.
11 subroutine data_apply(self, in1, func)
12     type(DataObj), target, intent(inout) :: self, in1
13     interface
14         real(8) function func(A, i, j)
15             integer, intent(in) :: i, j
16             interface
17                 real(8) function A(x, y)
18                     integer, intent(in) :: x, y
19                 end function
20             end interface
21         end function
22     end interface
23
24     type(SubGrid) :: sg
25     type(Neighbor), pointer :: n
26     integer :: lbid, gbid, blkI, blkJ, neigh, blkW, blkH
27
28     blkW = distribution_width(self%dist)
29     blkH = distribution_height(self%dist)
30     gInputData1 => in1
31
32     ! Iterate over all points in all local blocks
33     do lbid=lbound(self%vals,3), ubound(self%vals,3)
34         gbid = distribution_lbid2gbid(self%dist, lbid)
35         blkXOffset = distribution_blockLowX(self%dist, gbid) - 1
36         blkYOffset = distribution_blockLowY(self%dist, gbid) - 1
37
38         do blkJ=1,blkH
39             do blkI=1,blkW
40                 ! Apply the stencil function
41                 self%vals(blkI, blkJ, lbid) =    &
42                     func(AFunc1,                 &
43                          blkI + blkXOffset,      &
44                          blkJ + blkYOffset)
45             end do
46         end do
47     end do
48 end subroutine
```

Figure C.6: Code for GridWeaver's stencil application subroutine. Notice that the function `func` is called once per point in the grid and whenever `func` accesses a grid-point it does so through the `AFunc1` function.

```
1  program IcosahedralGameOfLife
2    include 'gridweaver.h'
3    implicit none
4
5    integer :: mpierr, i
6    integer, parameter :: N  = 3
7
8    type(Subgrid)      :: sg1L, sg2L, sg3L, sg4L, sg5L
9    type(Subgrid)      :: sg1R, sg2R, sg3R, sg4R, sg5R
10   type(Subgrid)      :: sgNP, sgSP
11   type(Neighbor)     :: n1, n2, n3, n4, n5, n6
12   type(Grid)         :: g
13   type(Distribution) :: dist
14   type(Schedule)     :: sched
15   type(DataObj)      :: earth
16
17   ! Initialize MPI and GridWeaver
18   call MPI_INIT(mpierr)
19   call gridweaver_initialize()
20
21   ! ===================
22   ! Set up environment:
23   ! ===================
24
25   ! Define a 6-point neighborhood
26   n1 = neighbor_new("up",    0,  1); n2 = neighbor_new("uprt", 1,  1)
27   n3 = neighbor_new("rt",    1,  0); n4 = neighbor_new("dn", 0,  -1)
28   n5 = neighbor_new("lfdn", -1, -1); n6 = neighbor_new("lf", -1, 0)
29
30   ! Initialize subgrids
31   sg1L = subgrid_new("sg1L", N, N); sg1R = subgrid_new("sg1R", N, N)
32   sg2L = subgrid_new("sg2L", N, N); sg2R = subgrid_new("sg2R", N, N)
33   sg3L = subgrid_new("sg3L", N, N); sg3R = subgrid_new("sg3R", N, N)
34   sg4L = subgrid_new("sg4L", N, N); sg4R = subgrid_new("sg4R", N, N)
35   sg5L = subgrid_new("sg5L", N, N); sg5R = subgrid_new("sg5R", N, N)
36   sgNP = subgrid_new("sgNP", 1, 1); sgSP = subgrid_new("sgSP", 1, 1)
37
38   ! Aggregate subgrids into a grid
39   g = grid_new("g")
40   call grid_addSubgrid(g, sg1L); call grid_addSubgrid(g, sg1R)
41   call grid_addSubgrid(g, sg2L); call grid_addSubgrid(g, sg2R)
42   call grid_addSubgrid(g, sg3L); call grid_addSubgrid(g, sg3R)
43   call grid_addSubgrid(g, sg4L); call grid_addSubgrid(g, sg4R)
44   call grid_addSubgrid(g, sg5L); call grid_addSubgrid(g, sg5R)
45   call grid_addSubgrid(g, sgNP); call grid_addSubgrid(g, sgSP)
46
47   ! Connect top and left borders
48   call grid_addBorder(g,  1, N+1,  N, N+1,  sg1L,    &
49                           1, N,    1, 1,    sg2L, 0)
50   call grid_addBorder(g,  0, 0,    0, N-1,  sg2L,    &
51                           N, N,    1, N,    sg1L, -1)
52
53   call grid_addBorder(g,  1, N+1,  N, N+1,  sg2L,    &
54                           1, N,    1, 1,    sg3L, 0)
```

```
 55   call grid_addBorder(g,  0, 0,    0, N-1,  sg3L,    &
 56                           N, N,    1, N,    sg2L, -1)
 57
 58   call grid_addBorder(g,  1, N+1,  N, N+1,  sg3L,    &
 59                           1, N,    1, 1,    sg4L, 0)
 60   call grid_addBorder(g,  0, 0,    0, N-1,  sg4L,    &
 61                           N, N,    1, N,    sg3L, -1)
 62
 63   call grid_addBorder(g,  1, N+1,  N, N+1,  sg4L,    &
 64                           1, N,    1, 1,    sg5L, 0)
 65   call grid_addBorder(g,  0, 0,    0, N-1,  sg5L,    &
 66                           N, N,    1, N,    sg4L, -1)
 67
 68   call grid_addBorder(g,  2, N+1,  N, N+1,  sg5L,    &
 69                           1, N,    1, 1,    sg1L, 0)
 70   call grid_addBorder(g,  0, 0,    0, N-1,  sg1L,    &
 71                           N, N,    1, N,    sg5L, -1)
 72
 73   ! Connect right and bottom borders
 74   call grid_addBorder(g,  N+1, 2,  N+1, N+1,  sg1R,    &
 75                             N, 1,    1,   1,  sg2R, -1)
 76   call grid_addBorder(g,    1, 0,    N,   0,  sg2R,      &
 77                             N, N,    N,   1,  sg1R, 1)
 78
 79   call grid_addBorder(g,  N+1, 2,  N+1, N+1,  sg2R,    &
 80                             N, 1,    1,   1,  sg3R, -1)
 81   call grid_addBorder(g,    1, 0,    N,   0,  sg3R,      &
 82                             N, N,    N,   1,  sg2R, 1)
 83
 84   call grid_addBorder(g,  N+1, 2,  N+1, N+1,  sg3R,    &
 85                             N, 1,    1,   1,  sg4R, -1)
 86   call grid_addBorder(g,    1, 0,    N,   0,  sg4R,      &
 87                             N, N,    N,   1,  sg3R, 1)
 88
 89   call grid_addBorder(g,  N+1, 2,  N+1, N+1,  sg4R,    &
 90                             N, 1,    1,   1,  sg5R, -1)
 91   call grid_addBorder(g,    1, 0,    N,   0,  sg5R,      &
 92                             N, N,    N,   1,  sg4R, 1)
 93
 94   call grid_addBorder(g,  N+1, 2,  N+1, N+1,  sg5R,    &
 95                             N, 1,    1,   1,  sg1R, -1)
 96   call grid_addBorder(g,    1, 0,    N,   0,  sg1R,      &
 97                             N, N,    N,   1,  sg5R, 1)
 98
 99   ! Connect east and west borders
100   call grid_addBorder(g,  N+1, 1,  N+1, N,   sg1L,    &
101                             1, 1,    1, N,   sg1R, 0)
102   call grid_addBorder(g,    0, 1,    0, N,   sg1R,    &
103                             N, 1,    N, N,   sg1L, 0)
104
105   call grid_addBorder(g,  N+1, 1,  N+1, N,   sg2L,    &
106                             1, 1,    1, N,   sg2R, 0)
107   call grid_addBorder(g,    0, 1,    0, N,   sg2R,    &
108                             N, 1,    N, N,   sg2L, 0)
```

```
109
110   call grid_addBorder(g,  N+1, 1,  N+1, N,   sg3L,    &
111                             1, 1,    1, N,   sg3R, 0)
112   call grid_addBorder(g,    0, 1,    0, N,   sg3R,    &
113                             N, 1,    N, N,   sg3L, 0)
114
115   call grid_addBorder(g,  N+1, 1,  N+1, N,   sg4L,    &
116                             1, 1,    1, N,   sg4R, 0)
117   call grid_addBorder(g,    0, 1,    0, N,   sg4R,    &
118                             N, 1,    N, N,   sg4L, 0)
119
120   call grid_addBorder(g,  N+1, 1,  N+1, N,   sg5L,    &
121                             1, 1,    1, N,   sg5R, 0)
122   call grid_addBorder(g,    0, 1,    0, N,   sg5R,    &
123                             N, 1,    N, N,   sg5L, 0)
124
125   ! Connect north and south borders
126   call grid_addBorder(g,   1, N+1,  N, N+1, sg1R,  &
127                            1,   1,  N,   1, sg2L, 0)
128   call grid_addBorder(g,   1,   0,  N,   0, sg2L,  &
129                            1,   N,  N,   N, sg1R, 0)
130
131   call grid_addBorder(g,   1, N+1,  N, N+1, sg2R,  &
132                            1,   1,  N,   1, sg3L, 0)
133   call grid_addBorder(g,   1,   0,  N,   0, sg3L,  &
134                            1,   N,  N,   N, sg2R, 0)
135
136   call grid_addBorder(g,   1, N+1,  N, N+1, sg3R,  &
137                            1,   1,  N,   1, sg4L, 0)
138   call grid_addBorder(g,   1,   0,  N,   0, sg4L,  &
139                            1,   N,  N,   N, sg3R, 0)
140
141   call grid_addBorder(g,   1, N+1,  N, N+1, sg4R,  &
142                            1,   1,  N,   1, sg5L, 0)
143   call grid_addBorder(g,   1,   0,  N,   0, sg5L,  &
144                            1,   N,  N,   N, sg4R, 0)
145
146   call grid_addBorder(g,   1, N+1,  N, N+1, sg5R,  &
147                            1,   1,  N,   1, sg1L, 0)
148   call grid_addBorder(g,   1,   0,  N,   0, sg1L,  &
149                            1,   N,  N,   N, sg5R, 0)
150
151   ! Connect to NP
152   call grid_addBorder(g,  0, N,  0, N, sg1L,    &
153                           1, 1,  1, 1, sgNP, 1)
154   call grid_addBorder(g,  0, 0,  0, 0, sgNP,    &
155                           1, N,  1, N, sg1L, -1)
156
157   call grid_addBorder(g,  0, N,  0, N, sg2L,    &
158                           1, 1,  1, 1, sgNP, 1)
159   call grid_addBorder(g,  1, 0,  1, 0, sgNP,    &
160                           1, N,  1, N, sg2L, -1)
161
162   call grid_addBorder(g,  0, N,  0, N, sg3L,    &
```

```
                          1, 1,  1, 1, sgNP, 1)
  call grid_addBorder(g,  2, 1,  2, 1, sgNP,    &
                          1, N,  1, N, sg3L, -1)

  call grid_addBorder(g,  0, N,  0, N, sg4L,    &
                          1, 1,  1, 1, sgNP, 1)
  call grid_addBorder(g,  2, 2,  2, 2, sgNP,    &
                          1, N,  1, N, sg4L, -1)

  call grid_addBorder(g,  0, N,  0, N, sg5L,    &
                          1, 1,  1, 1, sgNP, 1)
  call grid_addBorder(g,  1, 2,  1, 2, sgNP,    &
                          1, N,  1, N, sg5L, -1)

  ! Connect to SP
  call grid_addBorder(g,  N+1, 1,  N+1, 1, sg1R,    &
                            1, 1,    1, 1, sgSP, -1)
  call grid_addBorder(g,    1, 0,    1, 0, sgSP,    &
                            N, 1,    N, 1, sg1R,  1)

  call grid_addBorder(g,  N+1, 1,  N+1, 1, sg2R,    &
                            1, 1,    1, 1, sgSP, -1)
  call grid_addBorder(g,    0, 0,    0, 0, sgSP,    &
                            N, 1,    N, 1, sg2R,  1)

  call grid_addBorder(g,  N+1, 1,  N+1, 1, sg3R,    &
                            1, 1,    1, 1, sgSP, -1)
  call grid_addBorder(g,    0, 1,    0, 1, sgSP,    &
                            N, 1,    N, 1, sg3R,  1)

  call grid_addBorder(g,  N+1, 1,  N+1, 1, sg4R,    &
                            1, 1,    1, 1, sgSP, -1)
  call grid_addBorder(g,    1, 2,    1, 2, sgSP,    &
                            N, 1,    N, 1, sg4R,  1)

  call grid_addBorder(g,  N+1, 1,  N+1, 1, sg5R,    &
                            1, 1,    1, 1, sgSP, -1)
  call grid_addBorder(g,    2, 2,    2, 2, sgSP,    &
                            N, 1,    N, 1, sg5R,  1)

  ! Initaialize a GOL board
  dist = distribution_new("dist")
  call distribution_applyFillBlock(dist, g, N)
  sched = schedule_new("sched")
  call schedule_calculate(sched, g, dist, 1)
  earth  = data_new(sched)
  call data_fillRandom0sAnd1s(earth)

  ! Print starting state
  call data_print(earth, 6)

  ! Run for n turns
  do i=1,n
    call data_forceUpdate(earth)
```

```
217     call data_apply1(earth, earth, golStencil)
218   end do
219
220   ! Print resulting state
221   call data_print(earth, 6)
222 contains
223
224   !  .=================================================================,
225   !  |                Rules for Conway's game of life                  |
226   !  |   From: <http://en.wikipedia.org/wiki/Conway's_Game_of_Life>    |
227   !  |-----------------------------------------------------------------|
228   !  | (1) Any live cell with fewer than two live neighbours dies,     |
229   !  |       as if caused by under-population.                         |
230   !  |                                                                 |
231   !  | (2) Any live cell with two or three live neighbours lives on    |
232   !  |       to the next generation.                                   |
233   !  |                                                                 |
234   !  | (3) Any live cell with more than three live neighbours dies,    |
235   !  |       as if by overcrowding.                                    |
236   !  |                                                                 |
237   !  | (4) Any dead cell with exactly three live neighbours becomes    |
238   !  |       a live cell, as if by reproduction.                       |
239   !  `=================================================================='
240   real(8) function golStencil(A, i, j)
241     integer, intent(in) :: i, j
242     interface
243       real(8) function A(x,y)
244         integer, intent(in) :: x,y
245       end function
246     end interface
247     integer :: nNeighbors
248
249     ! Count number of neighboring cells containing life
250     nNeighbors = &
251       A(i,j+1) + A(i+1,j+1) + A(i+1,j) + A(i,j-1) + A(i-1,j-1) + A(i-1,j)
252
253     ! Update cell using the game-of-life rules:
254     if(A(i, j) == 1) then       ! Check rules (1), (2), and (3)
255       if(nNeighbors < 2)                          golStencil = 0
256       if(nNeighbors == 2 .or. nNeighbors == 3)    golStencil = 1
257       if(nNeighbors > 3)                          golStencil = 0
258       return
259     else                        ! Check rule (4)
260       if(nNeighbors == 3)  golStencil = 1
261       return
262     end if
263     golStencil = 0
264   end function
265 end program
```

Figure C.7: Code to execute Conway's game of life on a mini-icosahedral grid.