AUTONOMOUS HIERARCHICAL ADAPTIVE MESH REFINEMENT FOR MULTISCALE SIMULATIONS

BY

HENRY JOEL NEEMAN

B.S., State University of New York at Buffalo, 1987
B.A., State University of New York at Buffalo, 1987
M.S., University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

THIS PAGE INTENTIONALLY LEFT BLANK.

AUTONOMOUS HIERARCHICAL ADAPTIVE MESH REFINEMENT FOR MULTISCALE SIMULATIONS

Henry Joel Neeman, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1996
Michael Heath, Michael Norman, Advisors

Modern high resolution numerical simulations of multiscale physical phenomena require enormous computer resources; however, these resources are largely wasted on subdomains whose solutions do not require such high resolutions. Adaptive mesh refinement (AMR) addresses this problem by providing a means to perform high resolution computation only in areas that require it. In the AMR strategy discussed, a nested heirarchy of overlaying grids of increasingly fine resolution — in both space and time — permits high resolution computation in some areas and low resolution in others, either as a set of virtual grids, each encompassing the entire domain, or as a means of zooming in on a subdomain of interest. However, this AMR strategy is both subtle and cumbersome to code, and its data requirements are difficult to manage in a general way. To address this shortcoming, the *Hierarchical Adaptive Mesh Refinement* (HAMR) system provides support not only for AMR, but also for autonomous data management, thereby decoupling the numerical techniques of a simulation from the adaptive grid hierarchy to which it it applied.

For Teri

Acknowledgements

This work was supported in part by National Science Foundation grant ASC 9318185 and NASA grant NAG 5-2493.

I am grateful to the many people who helped me throughout this endeavor, not only for their encouragement and feedback, but also for ideas on how to make things work, or work better. Many thanks to my advisors, Michael Norman, Michael Heath and Dennis Gannon[1] not only for all of their help but also for their patience over the long haul. Much appreciation also to the rest of my committee, Paul Saylor, Faisal Saied and Donald Hearn, and to Eric Golin, who was unable to remain on my committee because of a changes of jobs, but who provided able assistance on my prelim.

An enormous number of people have provided feedback, support or just listened to me rant. I'd like to express my appreciation to them all, but I'm sure I'll forget a few of them, and they have my apologies in advance. For technical assistance, support and encouragement, my thanks to Dinshaw Balsara, George Baxter, Noam Ben-Ami, Jim Bottum, Steve Brandt, Jim Browne, Karen Camarda, Albert Cheng, Matt Choptiuk, Bob Fiedler, Mike Folk, Jill Hanson, John Jaynes, Susan John, Ben Johnson, B.I. Jun, Tejas Katwala, Paul and Margaret Klock, Pat Moran, Ruth Ann Nichols, Michele Plante, Ray Plante, Harold Ravlin, Barry Sanders, Ed Seidel, Crystal Shaw, Larry Smarr, Chris Song, Doug Swesty, John Towns, Robert Wilhelmson, Marianne Winslett, Jill and Peter White, and David Wojtowicz.

[1] Dennis was unable to attend my defense because of inclement weather, and so does not appear on the official forms, but he has acted as advisor during the course of my studies and research.

Among the people whose support was crucial to this endeavor are the staff of the National Center for Supercomputing Applications, particularly Beth McKown, Shirley Shore, Debbie Carrier and especially Jean Soliday. Without them, I would not have been able to take on this project.

Several people provided detailed technical information and helped me to develop the techniques described in this dissertation. My thanks to Peter Anninos, Marsha Berger, Brian Jewett, Joan Masso, Nelson Max, Manish Parashar, John Shalf, Paul Walker, and Yu Zhang.

If any one person has made HAMR possible, it is Greg Bryan. Not only has he provided me with the physics to showcase my project, he has also contributed keen insight into the details of adaptive mesh refinement techniques. And always with a smile. Thank you, Greg.

I also want to thank my family. On the Neeman side, my thanks for love and encouragement to my parents, Moshe and Renate, to Jenifer and Ed, to Ed and Maria, and to Lisa and Abby. And for welcoming me to the Murphy side, thanks to Pat, Johanna and Marilu, as well as Dolly, Alec and Kathy.

Finally, the person I want to thank the most, without whom I could not have mustered the confidence to tackle this project, nor the stamina to see it through, is my beloved wife, Teri Murphy.

Contents

Chapter 1

Introduction

Many modern numerical simulations of multiscale physical phenomena require enormous computer resources, in both memory storage and computing time, because their domains are discretized on high resolution meshes. However, these resources are often largely wasted on subdomains whose solutions do not require the maximum resolutions. Adaptive mesh refinement (AMR) is a class of strategies that address this problem by performing high resolution computation only in areas that require it.

Among the possible reasons for avoiding uniformly high resolution meshes are:

- some areas have small gradients, so the solution varies little among neighboring cells;
- in some areas the solution can be computed with sufficient accuracy on a low resolution mesh;
- there may be many redundant phenomena in the domain, only a few of which need to

be studied at high resolution;

- various phenomena of interest in the simulation may occur at widely varying time and length scales.

The literature for adaptive mesh refinement is extensive, dating back over more than twenty years and continuing today as a rich field of research. AMR strategies have been developed for elliptic, parabolic and hyperbolic systems. The many approaches vary considerably, in both philosophy and implementation. AMR strategies have been successfully applied to

- computational fluid dynamics (CFD),
- computational astrophysics,
- structural dynamics,
- magnetics,
- thermal dynamics,
- microwave theory

and many other areas of numerical research.

However, little work has been done in generalizing the adaptive strategies into flexible, modular systems that promote relatively quick and simple "plug-and-play" approaches for creating new adaptive simulations, and this lack is particularly noticeable in the context of simulations on structured meshes. Such general-purpose systems require a number of important properties, in order to provide maximal benefit to the research community:

- *Minimal knowledge of the system* to add new applications.
- *A simple interface to data and methods.*
- *Sophisticated memory management*, to relieve the scientist of the burden of allocating and deallocating grid space as the collection of grids evolves.
- *An extensive library* of commonly used subroutines.
- *Geometric flexibility* in mesh type, coordinate system, and staggered grid positioning, for simulations that have variables at various positions in and around each cell — for example, velocities at the nodes, energy fluxes at the edge centers and densities at the cell centers.
- *Expandability*
  - to adjust to the needs of various numerical schemes;
  - to incorporate new types of grids and new coordinate systems;
  - to be portable to many platforms — written in commonly available languages like C and Fortran;
  - to apply to many hardware architectures.

- *Algorithm flexibility*, because some sophisticated simulations require algorithms that are rather complex.

- *Clustering optimization*, in which cells to be refined are decomposed into conformally rectangular grids that make the best possible use of the capabilities of the platform.

In the AMR strategy discussed in this dissertation, developed by Marsha Berger and collaborators, a nested heirarchy of overlaying grids of increasingly fine resolution — in both space and time — permits high resolution computation in some areas and low resolution in others. At each level of resolution, a set of subgrids covers those portions of the domain that require at least that resolution, and each of these subgrids is in turn completely contained in some subset of the grids at the immediately coarser level. In some cases, the types of phenomena encountered at each level are of widely varying scales, so the refinement ratio may vary from level to level. The strategy can be thought of either as a set of increasingly fine virtual grids, each encompassing the entire domain, or as a means of zooming in on a subdomain of interest. Most importantly, it addresses the concerns listed above.

The showcase for the research conducted in this dissertation is an implementation of this AMR strategy, called the Hierarchical Adaptive Mesh Refinement (HAMR) system, which not only implements Berger's AMR strategy, but also addresses the problem of applying this strategy to existing simulation kernels in a manner that is both intuitive and convenient. Unlike other implementations of Berger's strategy, HAMR is a general-purpose approach that can be applied to a wide variety of numerical schemes, because it makes very few simplifying assumptions about applications, their numerical techniques, and the data they require.

The design of HAMR is driven in large part by a desire to decouple the physics of a simulation from the computing environment in which it operates, and especially from the particulars of how data are allocated, how they are managed, and how they interact with one another and with the methods that operate on them. The value of this approach is that it isolates the scientific researcher from details that are irrelevant to the subject of the experiment, and that can detract from the reasons for embarking on the research in the first place. In today's rapidly evolving research climate, too many scientists are having to learn too much about too many topics that have little direct bearing on the actual nature of the phenomena they are studying, because the methodologies with which they are presented, while compellingly powerful, are often subtle and difficult to manage.

Berger's AMR strategy is such an extreme example of this situation that it has been largely ignored as a practical approach to improving simulation efficiency. Yet the power of the strategy is not in dispute: Berger, her collaborators, and the few other researchers who have used her techniques have reported outstanding improvements in performance, especially in experiments in three dimensions — precisely the kinds of experiments that contemporary research attempts to address. But despite widespread acknowledgement of the importance both of adaptive techniques in general and of Berger's strategy in particular, its use has remained confined to a small group of scientific researchers.

The motivation for the development of HAMR is to ameliorate precisely this situation.

1.1 Structure of the Dissertation

This dissertation is composed of eight chapters, of which this chapter is the first. The next three provide background on numerical issues and the adaptive strategy of concern, while the final four discuss the associated doctoral research.

Chapter 2 describes the computational context in which this research was performed. Specifically, it covers the geometric issues associated with structured simulations, such as mesh types, coordinate systems and staggerings; finite difference methods, including their basic properties, as well as start-state issues such as initial and boundary conditions; and finally, legacy codes, which present a significant implementation challenge in the design of general-purpose AMR environments. The purpose of Chapter 2 is to lay down a clear, well-defined set of motivations, which will explain both the design of Berger's AMR strategy and its implementation in HAMR.

Chapter 3 examines a variety of related methodologies and research, including AMR strategies on unstructured grids, traditional multigrid methods, moving mesh methods, tree-based AMR approaches, and moving local uniform mesh refinement. This survey of related literature will show the place of Berger's AMR strategy in the battery of numerical techniques, with respect to the variety of approaches to numerical simulation, and more specifically to the field of adaptive methods.

Chapter 4 provides an overview of the AMR strategy of Berger and collaborators. This overview is presented primarily from a theoretical point of view, but includes some discussion of implementation issues and results of experiments using the strategy. Specifically, the chapter examines the premise of Berger's strategy; the layout of grid hierarchies; interpretations of AMR, including virtual grid and zoom interpretations; an overview of the AMR algorithm, including boundary value collection, evolution, flux-corrected conservation, projection, refinement, selection, clustering and regridding; the evolution of Berger's AMR strategy over the last decade and a half, including changes in allowed mesh types, overlapping grids, rotated grids, the clustering algorithm and staggerings; research that has employed Berger's strategy; and the relative lack of popularity of this strategy among computational scientists. Thus, Chapter 4 is designed to clarify not only what Berger's AMR is and how it works, but also how it has been used and how the numerical simulation research community has received it.

Chapter 5 describes a theoretical framework underlying autonomous data management for grid hierarchies. It delineates a data structure for representing grid hierarchies, including the scope and extent of data items as well as the categories of data types that are required; attributes of data and methods, including several attribute categories, rules governing certain attributes, and the means by which attributes are attached to the appropriate data items; the specification, which describes the data, methods and their relationships; the declaration, a user-produced description of this same information; modules, which encapsulate various operations and categories of data; data management, of data items, of strata and of the grid hierarchy. In this way, Chapter 5 clarifies the theoretical underpinnings upon which the implementation, HAMR, is built. Furthermore, these theoretical discussions motivate the explanation of the implementation itself, in the chapter that follows.

Chapter 6 describes the Hierarchical Adaptive Mesh Refinement system, and is divided into four sections, each of which corresponds to one of the major components of HAMR. The first section discusses HAMR data types, including element types, parameter types, and type attributes. These types and their attributes apply well to the wide variety of data needs, and to the myriad relationships these data can exhibit, for many classes of computational simulations. The second section discusses the low level function library, including operations on structured types, on dimensional types, on methods, and on spatial types. These operations provide the computational foundation for the bulk of the functionality of both data management and the AMR algorithms. The third section describes HAMR's implementation of the autonomous data management concepts laid out in Chapter 5, including the declaration, its parser, its data structure, the specification, the grid hierarchy data structure, data item macros, and predefined data items. By encapsulating the data management within these constructs, these operations can be completely decoupled from the application itself. The final section describes the implementation of AMR algorithms for Berger's strategy, including control, integration, boundary collection, extrapolation, refinement, regridding, truncation error estimation, selection, clustering, correction and projection. These algorithms do not merely implement Berger's AMR; rather, they expand and improve on it, by providing maximal generality and code reusability.

Chapter 7 presents the results of simulations performed using HAMR.

Chapter 8 presents conclusions and a summary.

Chapter 2

Computational Context

There are a variety of computational issues involved in numerical simulation techniques that affect the manner in which adaptive mesh refinement must be designed and implemented. Both the AMR scheme and the nature of the simulation itself impose restrictions on the general AMR framework, and on the engineering details of software construction. Among the computational issues that must be addressed are the geometries on which simulations are performed, the finite difference schemes used, and the characteristics and requirements of legacy codes.

2.1 Data Geometries

Numerical data come in a variety of geometries. Among the geometric degrees of freedom are mesh types, coordinate systems and staggerings.

2.1.1 Types of Meshes

A *field* is a mapping from one space to another; for example, the mapping $T : \Re^3 \rightarrow \Re$ describes temperature T in 3D space. A *mesh* is a collection of points or *nodes* and some notion of connectivity among them. A *grid* is a set of discrete approximations of fields mapped onto a mesh. (In the literature, *mesh* and *grid* are often used interchangeably. In this dissertation, they refer to different concepts, to reduce ambiguity.)

Meshes come in several types (Figure 2.1). Each mesh type has its own means of specification, and for each there are associated strategies for refinement.

First, some meshes are *empirical*; that is, they are collections of nodes that have no intrinsic connectivity among them. Empirical meshes can be refined by generating new nodes in areas that require higher resolution. Alternatively, a connectivity can be imposed on a empirical mesh by means of Delauney Triangulation [WS90], and then refinement methods appropriate for unstructured meshes (see below) can be used.

Most numerical simulations use meshes with intrinsic connectivity. Of these, the most generic in construction, and the most complicated to specify, is the *unstructured* mesh, which is a collection of nodes and a description of the connectivity between them. This connectivity is generally specified by the subgroup of nodes that delimit each *cell* in the mesh.

Unstructured meshes have arbitrary connectivity and geometry. Typically, each cell is the simplex [Hof89], or minimal polytope, of the mesh's dimensionality (for example, [ZSZZ90], [HST90], [RMC89]) — lines in one dimensional space, triangles in two dimensions,

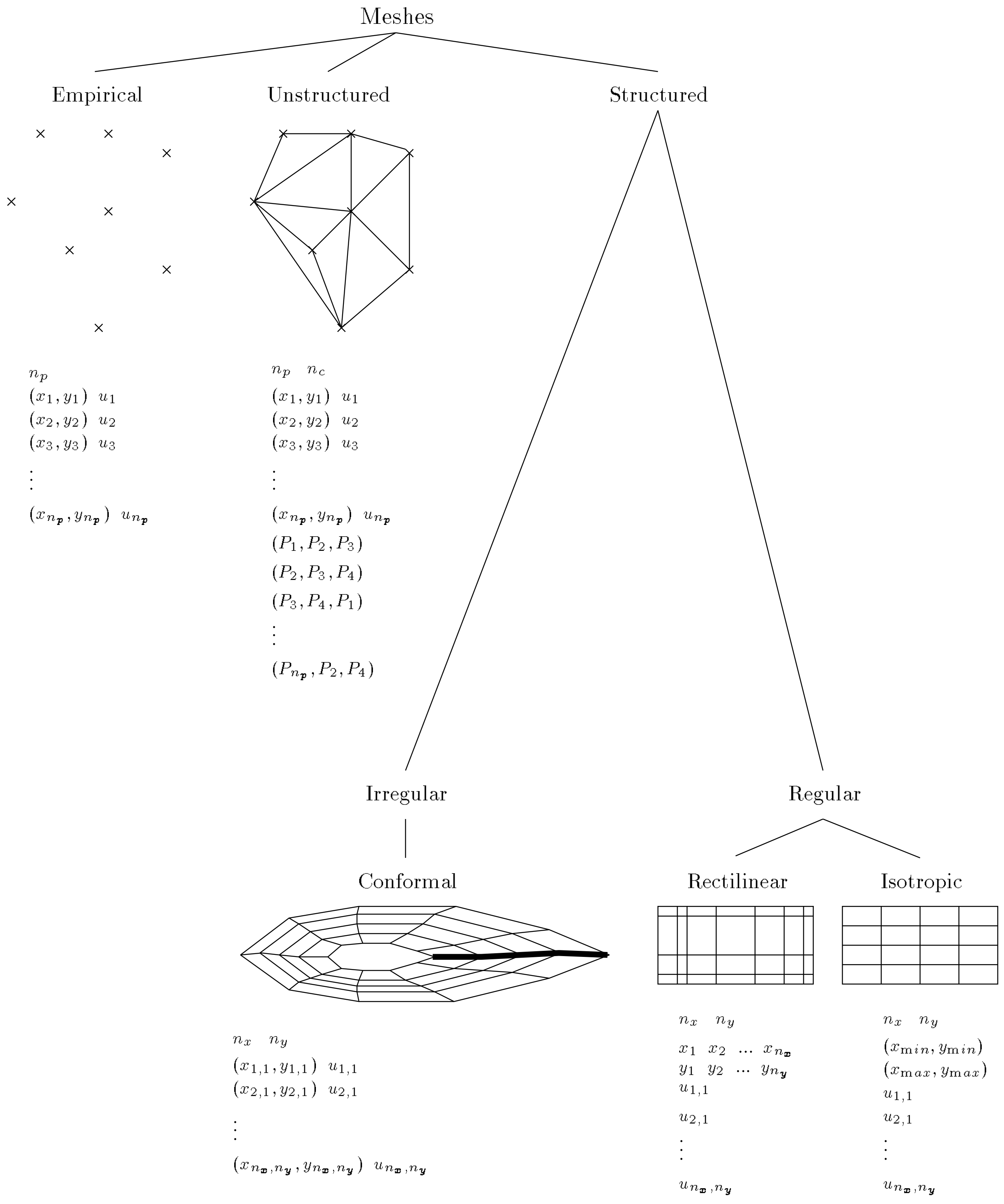


Figure 2.1: Hierarchy of mesh types

tetrahedra in three dimensions — but this property does not always apply (for example, [SC89], [MGS88]). In fact, in some cases a single mesh can consist of various different simplicial complexes.

Mesh refinement on unstructured meshes is typically achieved by placing one or more new nodes on the surface of, or inside, each cell that is to be refined — and perhaps adjusting the positions of the nodes — and then connecting the nodes of the cell to the new node(s), thereby creating a new set of finer cells from the coarse cell (for example, [LR88], [MF88], [TRS90]).

A *structured* mesh, unlike an unstructured mesh, has its connectivity implicit in the specification of the mesh; that is, the order in which the nodes are specified determines their connectivity. Specifically, a structured mesh is an array of nodes, each connected to two others along each axis (except for those on a surface of the mesh, which are connected to fewer). The cells of a structured mesh are implicitly defined by the relationships between nodes: a cell in a k–dimensional mesh is the space bounded by 2^k nodes connected as a (conformal) cube. Thus, structured meshes are arranged in the same manner as arrays in computer memory.

Structured meshes are more difficult to refine than unstructured meshes, and are thus the subject of this dissertation. This difficulty arises from the three possible approaches to structured mesh refinement:

- cells can be added to the existing mesh, in which case its implicit connectivity is lost, requiring that the associated structured mesh solution method be abandoned;

- nodes can be shifted about the physical domain to follow the most significant phenomena, an approach which does not allow the total number of cells to grow, and which is applicable to only the most general structured mesh geometries;

- multiple structured submeshes of varying resolutions can be added and their interactions controlled, requiring considerable additional coding to manage both the refinement and the associated data structure, but allowing both arbitrary refinement and the use of solution methods applicable to simple, fixed geometries.

The most general kind of structured mesh is the *conformal* mesh. It has the connectivity of a structured mesh, but its geometry — that is, the placement of its nodes and the shapes of its cells in physical space — is arbitrary. It is specified by the positions of its nodes. Conformal meshes range from slight perturbations off the Cartesian axes (for example, [MLP90], [Kim90], [DT90]) to almost totally deformed (for example, [ALP90], [SPB89]).

Less general than conformal meshes are *regular* meshes. These meshes have mesh lines parallel to the coordinate axes, and thus their geometric properties are far simpler than those of conformal meshes. The two major types of regular meshes are *rectilinear* and *isotropic* meshes. Rectilinear meshes have arbitrary spacings along each axis, so that cells may vary in size (for example, [Dem91], [Jia89]). They can be described by the positions of the nodes along each axis. Isotropic meshes are the simplest of all, a special case of rectilinear meshes with all cells of identical geometry; they can be completely described by two diagonally opposite corners of the mesh and the number of nodes along each axis.

In some cases, a simulation will use several interconnected structured meshes, or *blocks*, rather than a single mesh. This approach permits *cavities*, regions inside the physical domain that are not contained in the computational domain. For example, a jet engine can be composed of many blocks, so that the boundaries abut the major components of the engine — the hub, splitter and nacelle — and the domain is the air passages inside the engine [Ste91]).

Topologically, structured meshes are identical; their differences are exclusively geometric. This shared property allows many kinds of structured meshes to be subject to the same kind of mesh refinement strategies, because the meshes can all be treated as conformal cubes of the appropriate dimension. Only the aspects of the refinement strategies that directly relate to physical geometries of the meshes — for example, interpolation — need to be modified to apply the strategies to the different types of structured meshes.

2.1.2 Coordinate Systems

A mesh can be defined on many different kinds of coordinate systems (Figure 2.2), and a general-purpose approach to AMR must address the possibility of incorporating this geometric variety.

The simplest and most common coordinate system is the *Cartesian* or *rectangular* coordinate system. Cartesian coordinates can be specified completely implicitly and are commonly used to describe the real world, so they are a natural basis for numerical simulation.

However, many meshes are instead defined in *polar* coordinates, typically one of three

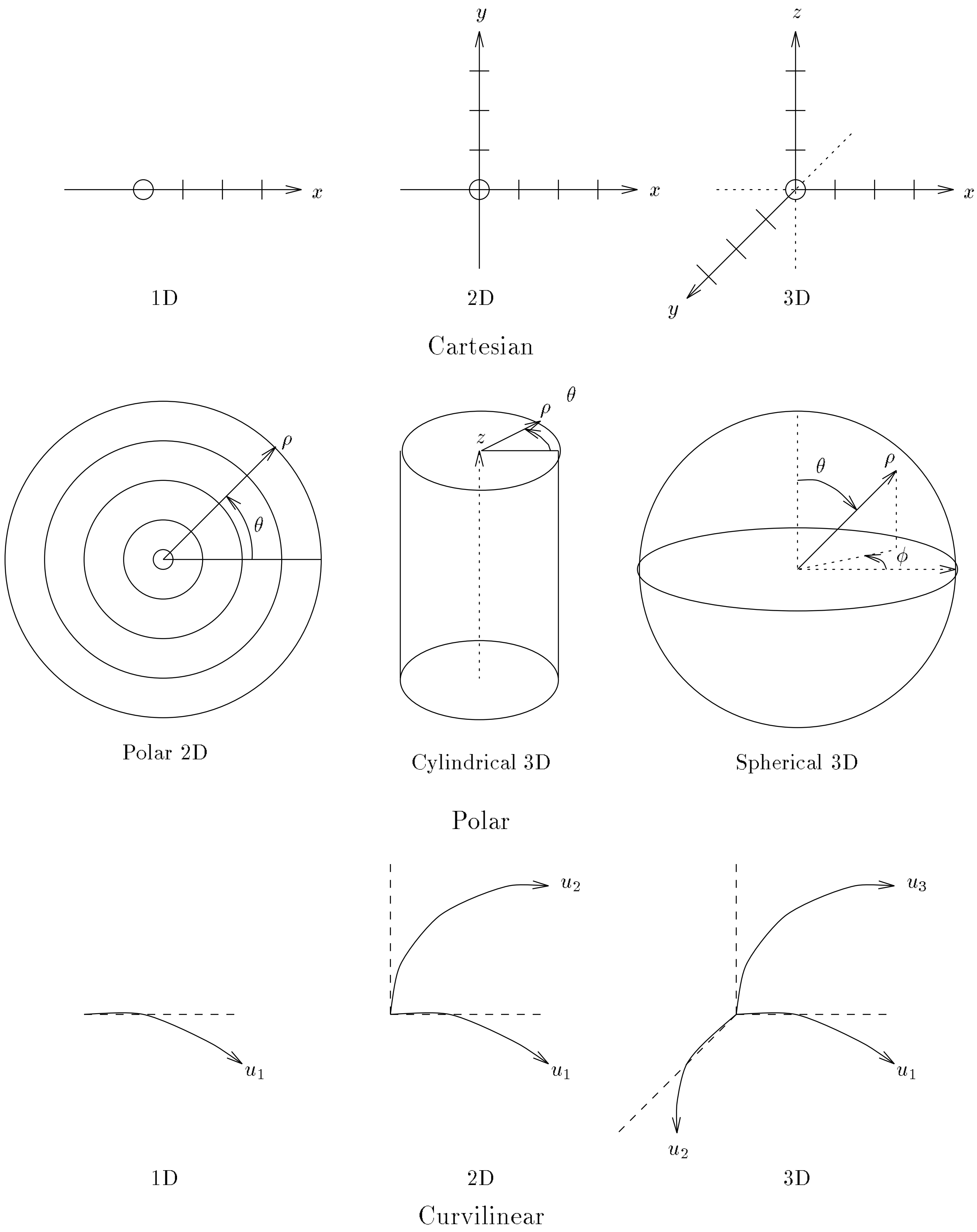


Figure 2.2: Coordinate systems

types. The simplest is two dimensional polar coordinates; in three dimensions, the two polar systems are *cylindrical* (for example, [Bor90]) and *spherical* (for example, [Bar89]). In principle, polar coordinates in dimensions higher than three are a natural extension of 3D polar coordinate systems.

The most general coordinate systems are *curvilinear*. Such systems must be explicitly described and may be only an approximation of the true coordinates desired. Also, it is important to distinguish between curvilinear coordinates and conformal meshes; the latter are typically defined in Cartesian coordinates. In some cases, conformal meshes are transformed into regular meshes as the Cartesian space is mapped into an appropriate curvilinear space (for example, [ALP91]).

A special kind of mesh used in 3D polar coordinates is the $2\frac{1}{2}$D polar mesh. For example, a 2D rectangular plane can be swept about the z–axis, producing a $2\frac{1}{2}$D cylindrical mesh; a 2D polar semicircular plane can be swept about a vertical line, producing a $2\frac{1}{2}$D spherical mesh (Figure 2.3). Computation and refinement occur on the 2D rectangular or polar mesh, but the physical domain is a 3D cylinder or sphere, respectively. (An example of an application that uses $2\frac{1}{2}$D meshes can be found in [SNM89]).

Given this multiplicity of coordinate systems, the best means of achieving a general-purpose AMR system is to isolate the meshes' geometric qualities as much as possible. Specifically, the most general approach to AMR includes AMR algorithms most of which rely on indices within the computational domain, rather than on physical positions. As for AMR algorithms which require physical positions, they must be decoupled from the rest of

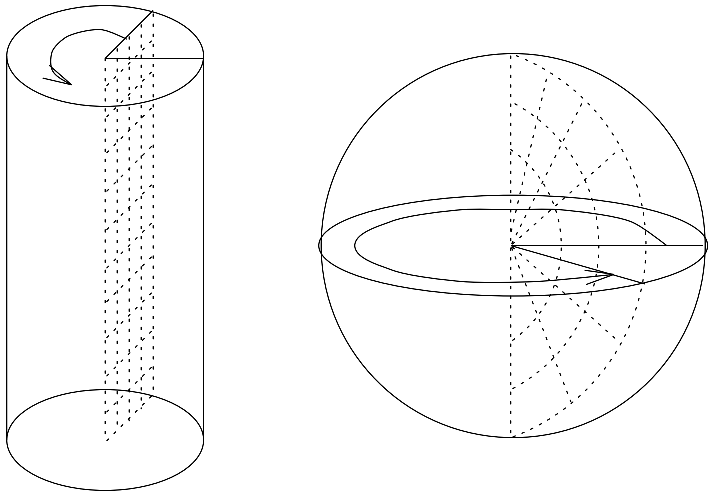


Figure 2.3: Cylindrical and spherical $2\frac{1}{2}$D meshes

the AMR system, and they are rendered most efficient by incorporating whatever geometric simplifications can be reliably assumed.

2.1.3 Location of Variables

Many multivariate applications require *staggered* grids; that is, grids whose variables exist at various loci within and on the interfaces of each cell. In many cases, these loci will either be at the nodes of the cells, or staggered half a cell width along each of some combination of axes; that is, at edge, face and cell centers (Figure 2.4). Thus, in 1D systems, a grid can have variables at the nodes and at the cell centers; in 2D, at the nodes, at the edge centers and the cell centers; in 3D, at the nodes, edge centers, face centers and cell centers, and so on.

In one example [LM92], stream function variables such as velocity are stored at mesh

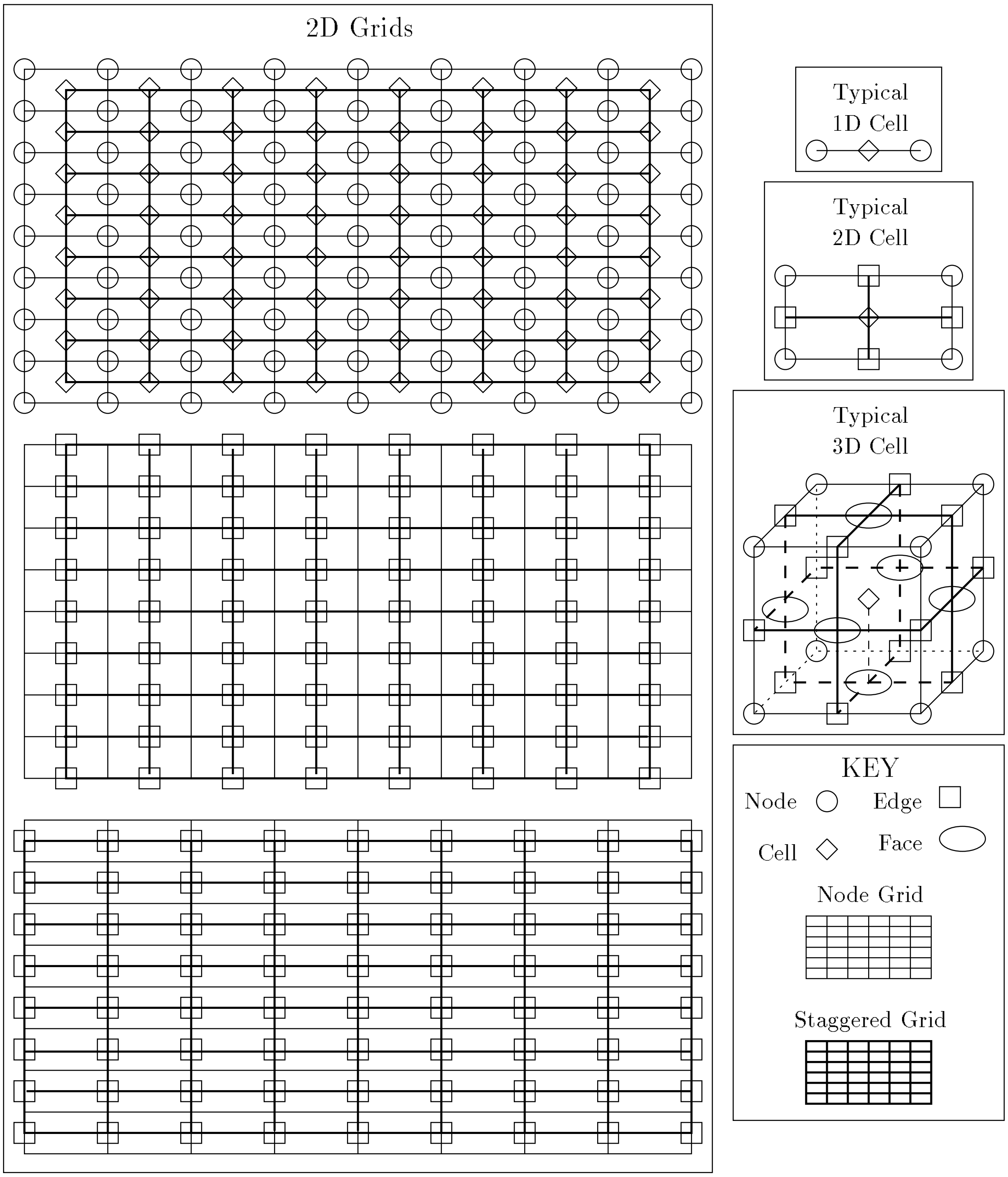


Figure 2.4: Variable loci on staggered grids

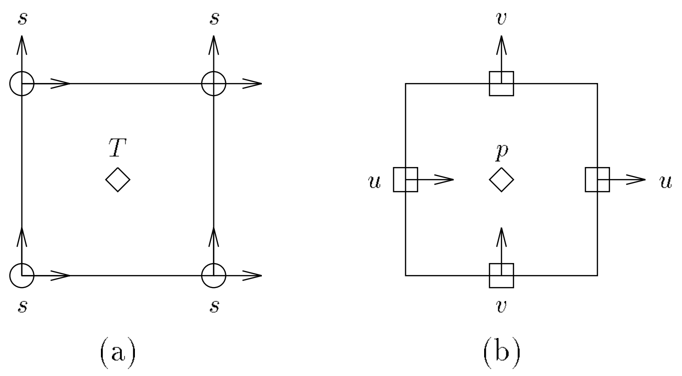


Figure 2.5: Staggered grid example

nodes, while temperature is located at the cell centers (Figure 2.5a). In other examples [TLVR91], [WH92], velocities are stored at cell edges and other quantities (e.g., pressure) at cell centers (Figure 2.5b).

However, meshes are not always staggered by half a zone size along each axis. In some simulations, variables can be at fixed but arbitrary fractions of a zone size within each cell (for example, [NP87]).

The AMR-related problem that these staggerings create arises because many AMR algorithms produce results over all variables. Each such result is located on a specific staggering, and all relevant variables that are on other staggerings must have their results transferred to the staggering of the result. For example, if the region to be refined is selected based on the values of the cell-centered density and the face-centered velocities, then the results for those variables must be merged to produce an overall refinement region, which would be the union of the individual variables' refinement regions.

2.2 Finite Difference Methods

At the heart of every time-dependent numerical simulation is a *solver*, a module that *evolves* the solution vectors of a partial differential equation forward in time by a specified time interval. The solver captures the physical processes that govern the application; the rest of the simulation software exists primarily to support the solver. Among the most popular categories of spatially discretized numerical schemes are *finite difference* methods.

In finite difference methods, the solver is typically a function that maps the initial value at a locus and at several loci surrounding it to a new value at the locus, such as would be required for an initial value partial differential equation. For example, a simple two-dimensional solver might look like so:

```
subroutine solve (u, ni, nj, istart, iend, jstart, jend, old, new)
integer ni, nj, istart, iend, jstart, jend, old, new
real u(ni,nj)
integer i, j

do j = jstart, jend
  do i = istart, iend
    u(i,j,new) =
      F(u(i-1,j-1,old),u(  i,j-1,old),u(i+1,j-1,old),
        u(i-1,  j,old),u(  i,  j,old),u(i+1,  j,old),
        u(i-1,j+1,old),u(  i,j+1,old),u(i+1,j+1,old))
  enddo
enddo
end
```

This example has a three point stencil on each spatial axis and a two point stencil on the time axis, and is referred to as a *forward time, centered space* scheme (Figure 2.6). In many cases,

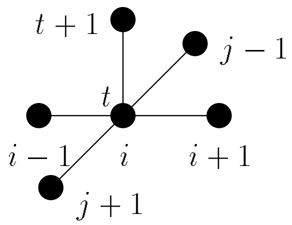


Figure 2.6: Forward time centered space stencil

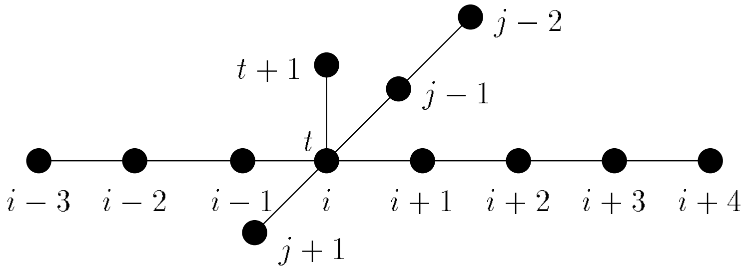


Figure 2.7: Stencil with a variety of component values

the mapping function `F` will actually operate not just over the range `[i-1:i+1][j-1:j+1]` but rather over `[i-s:i+s][j-s:j+s]` for some constant stencil value `s` ≥ 1. In fact, the stencil value can be different along each axis, and indeed on each side of each axis; that is, there may be different values for `sxmin`, `sxmax`, `symin` and `symax` (Figure 2.7).

Finite difference methods are subject to two primary concerns: consistency and stability. *Consistency* means that the truncation error τ — that is, the error inherent in the finite difference approximation of partial derivatives — approaches zero as resolution grows arbitrarily fine; that is, for zone width Δx and time interval Δt,

$$\lim_{\Delta x \to 0, \Delta t \to 0} \tau = 0$$

Sod expresses this as a case in which "... the differential equation fails to satisfy the finite difference method by an arbitrarily small amount" [Sod85]. *Stability* is the condition that the growth of errors in the solution is bounded for sufficiently small Δt. Haltiner and Williams [HW80] give three definitions for stability:

- " ... [The scheme's] solutions remain uniformly bounded functions of the initial state for all sufficiently small Δt ...."
- "... When the corresponding differential solution is bounded, a finite difference scheme is *unstable* if, for a fixed ... conditions, there exist initial disturbances for which the ... solution becomes unbounded ...."
- "... [T]he cumulative effect of all round-off errors remains negligible as n increases."

The Lax Equivalence Theorem [Ric57] implies that a consistent, stable finite difference method will *converge* — that is, approach the true solution — for appropriate initial conditions and discretization.

A very common stability condition is that the ratio $\Delta t/\Delta x$ falls within a particular range, a condition of considerable significance for adaptive mesh refinement, since under this condition stability does not depend on the absolute resolution, but rather on the ratio of temporal to spatial resolution. Of course, an ideal problem is one that is both stable and consistent, and for which τ approaches zero very rapidly relative to Δx and Δt. In practice, however, this case rarely arises, and in fact various subregions of the domain can converge to the true solution at different rates.

In experimental numerical simulation, the analytic solution is not known; if it were, there would be no point in imposing a discretization and a finite difference scheme on the differential equation, when the solution could be obtained directly. However, many researchers use problems with known analytic solutions as tests for examining new numerical techniques, since the numerical result can be directly compared to the true solution.

A finite difference method requires two kinds of input information: the initial condition and the boundary conditions.

2.2.1 Initial Conditions

The *initial condition* provides a starting state for a simulation, from which the progress of its evolution can be examined. At the implementation level, initial conditions fall into two categories: those which are explicitly stored and must be loaded into the solution vector at runtime, and those which can be calculated analytically.

Among explicitly stored initial conditions are phenomena that are observed in nature. For example, McInnes, McBride and Leslie simulated a cold front over southeastern Australia initialized from "... objective analyses produced routinely by the Bureau of Meteorology ...." [MML94]. Another form of explicitly stored initial condition is that which has been computed in a separate numerical simulation. For example, Bertschinger's COSMICS package generates initial conditions for cosmological simulations [Ber95].

As for analytic data, many classic problems have analytic solutions that make them ideal candidates for exploring the accuracy of numerical techniques. For example, the transport equation [MB92]

$$\frac{d\rho}{dt} + \nabla(\rho c) = 0$$

is an excellent initial test of finite difference methods, using the analytic solution at time $t = 0$ for the initial condition. Similarly, initial conditions can be based on scalar values; for example, a *shock tube* begins with each grid point of each variable initialized to one of two scalar values for that variable, depending on which side of the shock interface it is located.

Finally, some of these techniques can be combined. For example, Wilhelmson and collaborators conducted a storm simulation based on observed data from a severe storm that occurred in the southwestern United States, which was then perturbed to produce more appropriate conditions [WJS+90]; Dudhia and Moncrieff simulated a squall line using a vertical sounding translated to obtain horizontally uniform thermodynamic and wind profile, which they then perturbed analytically. [DM89].

2.2.2 Exterior Boundary Conditions

Boundary conditions for finite difference methods are often implemented by providing a set of additional cells surrounding the active computational domain (Figure 2.8). These *ghost boundaries* have values which express the boundary conditions of the differential equation discretized by the finite difference method. Specifically, the ghost boundary values are provided to cover the stencil of the outermost cells of the computational domain, but their values are not generally considered part of the overall solution of the differential equation.

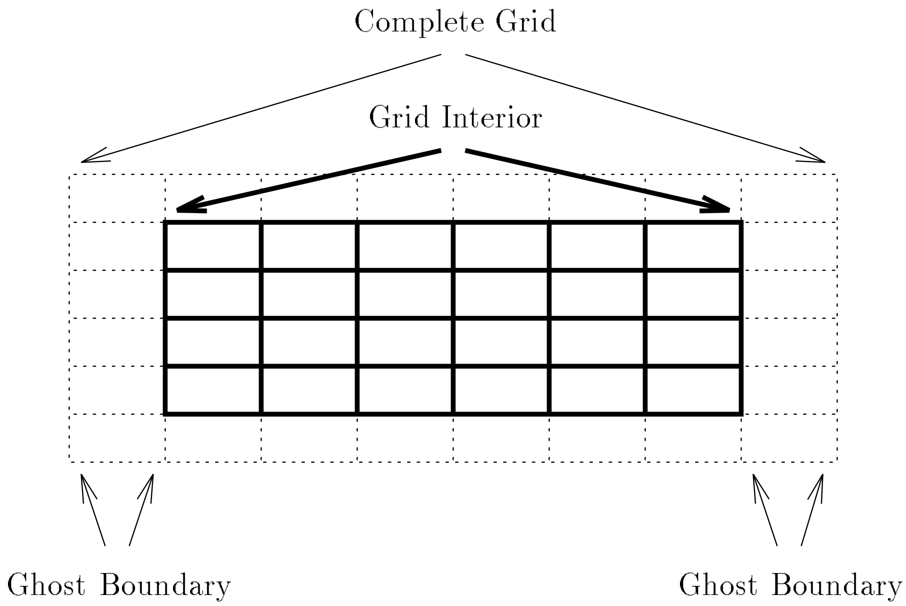


Figure 2.8: Ghost boundary region

Boundary conditions fall into two categories: those on boundary regions that are actually contained inside the overall computational domain, and those that are exterior to the domain. In the context of adaptive mesh refinement, this distinction is critical. Interior boundary conditions are a separate case that will be discussed in the context of AMR. However, exterior boundary conditions are universal; that is, all finite difference methods require them in some form.

Among the most common types of exterior boundary conditions (Figure 2.9) are:

- periodic;
- reflecting;
- inflow;

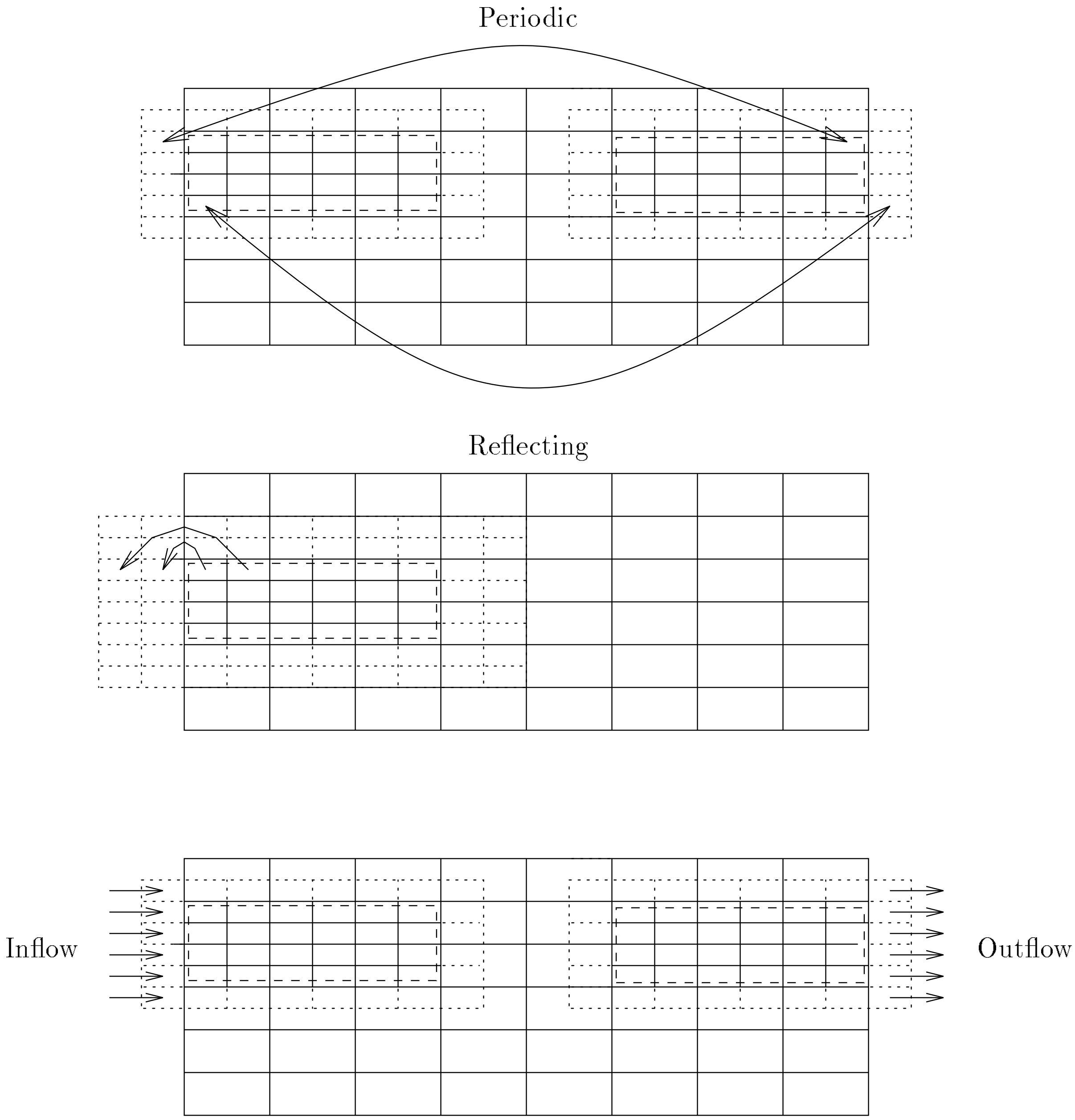


Figure 2.9: Common exterior boundary conditions

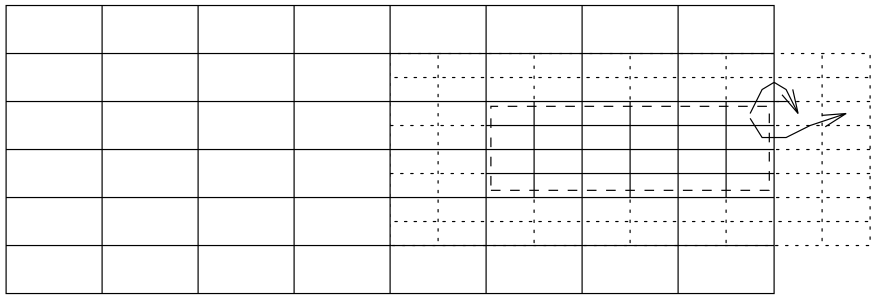


Figure 2.10: Grid with copied extrapolation boundary

- outflow;
- singularity.

Periodic boundaries have continuity of solution values on opposite ends of the mesh; that is, the "rightmost" cell along an axis is computationally adjacent to the "leftmost" cell. Conceptually, a periodic boundary is like an infinite chain of identical domains, in both directions along each periodic axis. *Reflecting* boundaries are mirror images; that is, the boundary value k exterior loci outside the domain interface is identical to the computed value k interior loci inside the domain interface. *Inflow* boundary conditions often have analytic values — for example, the amount of material being ejected from a source — and *outflow* boundaries typically have extrapolated values. For example, the simplest outflow extrapolation is to set the exterior boundary values equal to the computed value closest to the exterior interface (Figure 2.10). Finally, *singularity* conditions are applied to collections of computational values that represent a single physical position; for example, the center of a circle or sphere in polar coordinates (Figure 2.11).

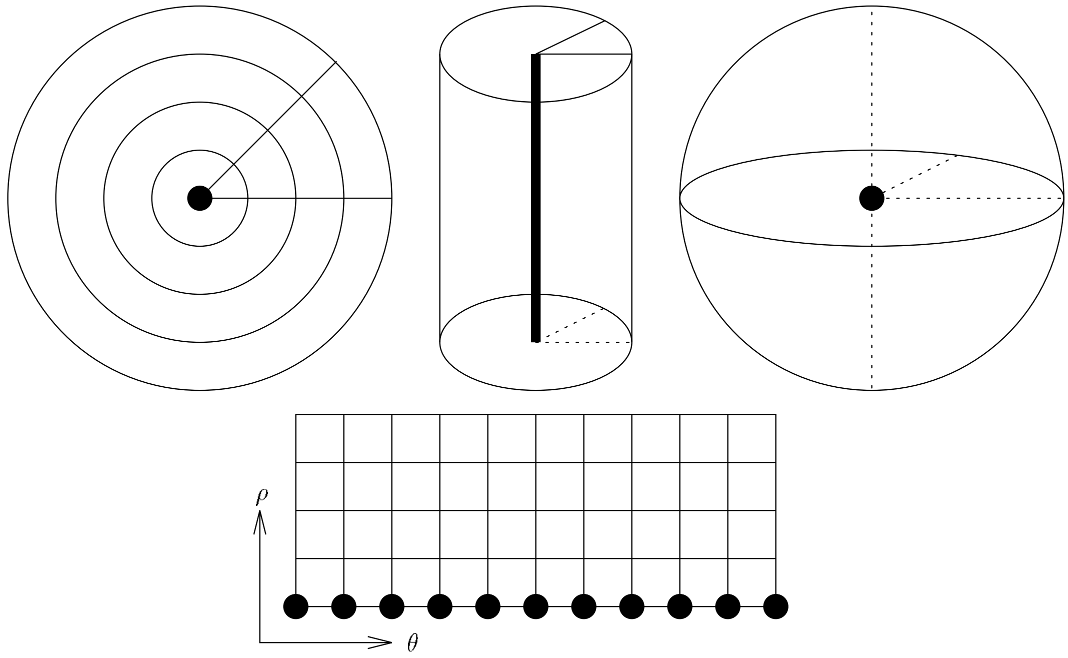


Figure 2.11: Grids with singularities

Many applications have combinations of these types of exterior boundary conditions. For example, Anninos, Norman and Anninos [ANA95] study cosmological sheets with periodic boundaries along the x-axis, a reflecting boundary at $y_{\min}$ and an inflow boundary at $y_{\max}$ (Figure 2.12). In fact, exterior boundary types can be combined within a single boundary. For example, Stone and Norman [SN93] discuss a two-dimensional protostellar jet simulation with outflow boundaries at $x_{\max}$, $y_{\min}$ and $y_{\max}$, and with a reflecting boundary at $x_{\min}$, except for a few zones in the center of that boundary, which are the inflow of the jet (Figure 2.13).

The almost limitless variety of exterior boundary conditions is a significant obstacle to the development of general-purpose AMR frameworks. The problem is not simply that many boundary conditions are entirely application-dependent, and therefore require the incorpo-

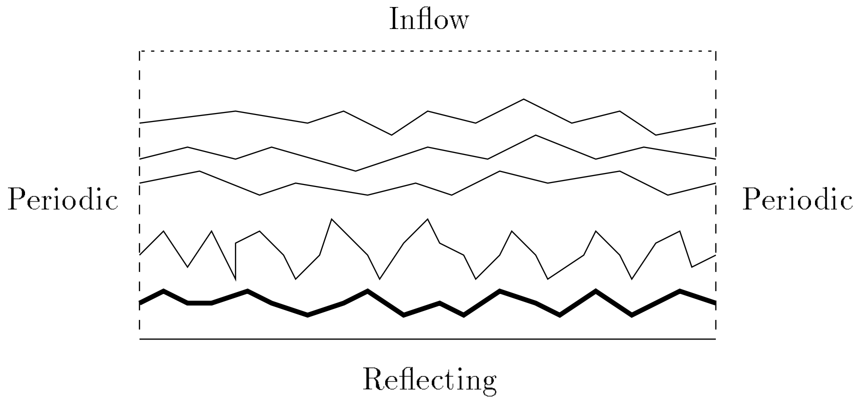


Figure 2.12: Exterior boundaries for cosmological sheets

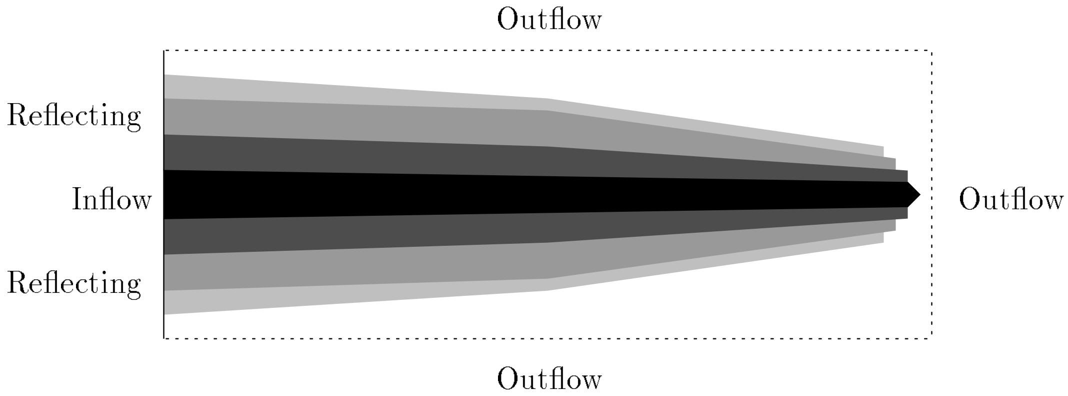


Figure 2.13: Exterior boundaries for a protostellar jet

ration of problem-specific subroutines. A far more serious problem is that some boundary conditions require access to multiple subdomains which may be stored discontiguously. For example, a periodic boundary condition requires access to the subdomain whose position is either laterally or diagonally opposite from the interface associated with the boundary (as can be seen in Figure 2.9), and presumably this second subdomain, which contributes to the boundary values of the first subdomain, should be of the same resolution as the first. Thus, the algorithms which implement these boundary conditions require considerable knowledge about the data management approach of the system.

2.3 Legacy Codes

The use of computers for numerical simulation is a mature field. Over the years, scientists have written, tested and used hundreds of simulation codes. Many of these codes have existed in various forms for years or even decades, and have been altered over time to expand and improve them. In addition, the many experiments that have been conducted using these codes constitute a testbed for their reliability and accuracy, and in some cases the codes have been algorithmically verified. Also, these codes often have multiple contributors, each adding modules or even statements within existing modules. Such codes are called *legacy* or *historical* codes, and more colloquially *dusty decks.*

Understandably, scientists with well-established legacy codes are reluctant to replace them with new, completely rewritten codes. A significant risk in replacing a legacy code is that some subtle aspect of the code might be overlooked or misunderstood, and that over the long term a seemingly minor error might have serious consequences. Also, many of these codes are thousands of lines long; rewriting them from scratch requires a significant commitment of time and effort, which could otherwise be directed toward scientific research.

Thus, legacy codes present inherent obstacles to creating a general-purpose AMR framework. The challenge is to develop the AMR system in such a way that an existing code can be incorporated in a simple, consistent manner, with minimal reinstrumentation. An ideal AMR system would allow a legacy code to be incorporated with no recoding whatsoever, or with the recoding fully automated, but such an approach is unrealistic in the near term. Still, a near-ideal AMR system would be able to incorporate a legacy code with only a few hours of additional programming.

2.4 Summary

A variety of computational issues contribute to the requirements of adaptive mesh refinement systems, especially those intended to be useful over a wide range of research topics, applications, platforms and numerical schemes. Each of these issues presents a significant challenge to those who would design such an AMR system.

Geometric and topological issues, such as mesh types, coordinate systems and staggerings, require a broad-based approach to infrastructure design, the first two because the adjustments they necessitate must be decoupled as much as possible from other aspects of the system, and the last because the various staggerings may need to interact, and so the notion of staggering must be fully embedded within the system.

The issues surrounding finite difference methods have perhaps the most significant impact on the design of general-purpose AMR systems, because such methods are the reason that structured AMR systems are designed in the first place. Both the abstract mathematical underpinnings of these methods and the practical implementation details have fundamental repercussions in the nature of multipurpose adaptive environments.

Finally, a great many of the simulations in use today are codes with long histories, which would be difficult to redesign or to rewrite from scratch. Therefore, a general-purpose AMR system must allow existing codes to be incorporated with minimal recoding by the application scientist.

Chapter 3

Related Methodologies and Research

Unlike adaptive techniques for unstructured meshes, which are relatively simple to devise and to implement, all of the existing AMR strategies on structured meshes are complex and subtle, and they suffer from a variety of disadvantages that may make them appear unsuitable or unattractive for many applications. But their study, and the study of related techniques such as traditional multigrid methods, can provide insight into both the requirements of structured AMR and the means by which it can best be achieved.

3.1 AMR Strategies on Unstructured Grids

The literature for computational simulation of physical phenomena contains a wealth of material about adaptive mesh refinement on unstructured meshes, as noted earlier. An obvious possibility, then, is to reformulate existing structured simulations in an unstructured context, and then applying the unstructured AMR techniques.

However, it is entirely unclear whether such reformulations are ideal or even practical. An illustrative example of this dilemma is provided by computational fluid dynamics (CFD) research, because little of the unstructured AMR literature describes the use of such grids for CFD. Mavriplis [Mav90] points out

> [The use of unstructured mesh techniques] in the field of computational fluid dynamics (CFD) constitutes a relatively recent phenomenon. This situation is probably due to the large overheads generally incurred with unstructured mesh techniques, ... The advantages of unstructured meshes lie in the ability they afford for flexibly discretizing arbitrarily complex geometries, and in the ease with which they lend themselves to adaptive meshing techniques, ...

Mavriplis' point is well taken, but with the advent of the kinds of AMR strategies that this dissertation addresses, unstructured CFD techniques — and particularly unstructured AMR for CFD — may not be necessary. This point leads to an important criterion for deciding whether a "plug-in" AMR system is easy to use: does it require that the scientist completely reformulate the simulation of interest?

To adopt Mavriplis' methods, one must restate one's physical properties in the context of an unstructured grid. This requirement is likely to be needlessly burdensome on the scientist and is therefore far from ideal. Rather, an AMR system should permit existing "off the shelf" simulations to be incorporated quickly and with a minimum of recoding by the scientist. Therefore, a superior alternative to Mavriplis' suggestion is to provide an AMR technique which can be applied directly to a structured simulation.

3.2 Multigrid Methods

Multigrid methods are a class of simulation techniques which illustrate important principles related to structured adaptive mesh refinement. These methods solve complex, sophisticated simulations by rapidly reducing the error inherent in many single grid iterative methods. Multigrid methods employ a hierarchy of grids of varying resolution, each grid covering the entire computational domain (Figure 3.1). The underlying premise is that, although iteration on a fine mesh quickly eliminates the high frequency components of error, low frequency components take much longer, resulting in an inefficient or inaccurate solution [SH90]. However, by iterating on meshes of various scales, the smoother error components can be reduced quickly as well.

Multigrid methods have several different scheduling algorithms (Figure 3.2). The simplest schedule is the *V-Cycle*, in which the system cycles from the coarsest to the finest grid and then back. A popular cycling schedule for multigrid methods is the *Full Multigrid V-Cycle* or *FMV* schedule. Here, information is transferred from the finest grid to the coarsest grid, by stepping upwards and downwards in resolution to a final, most accurate set of meshes.

Another popular cycling schedule is the *W-cycle*, named for the letter it resembles. In this schedule, iteration begins at the coarsest mesh, recursively cycles down through increasingly

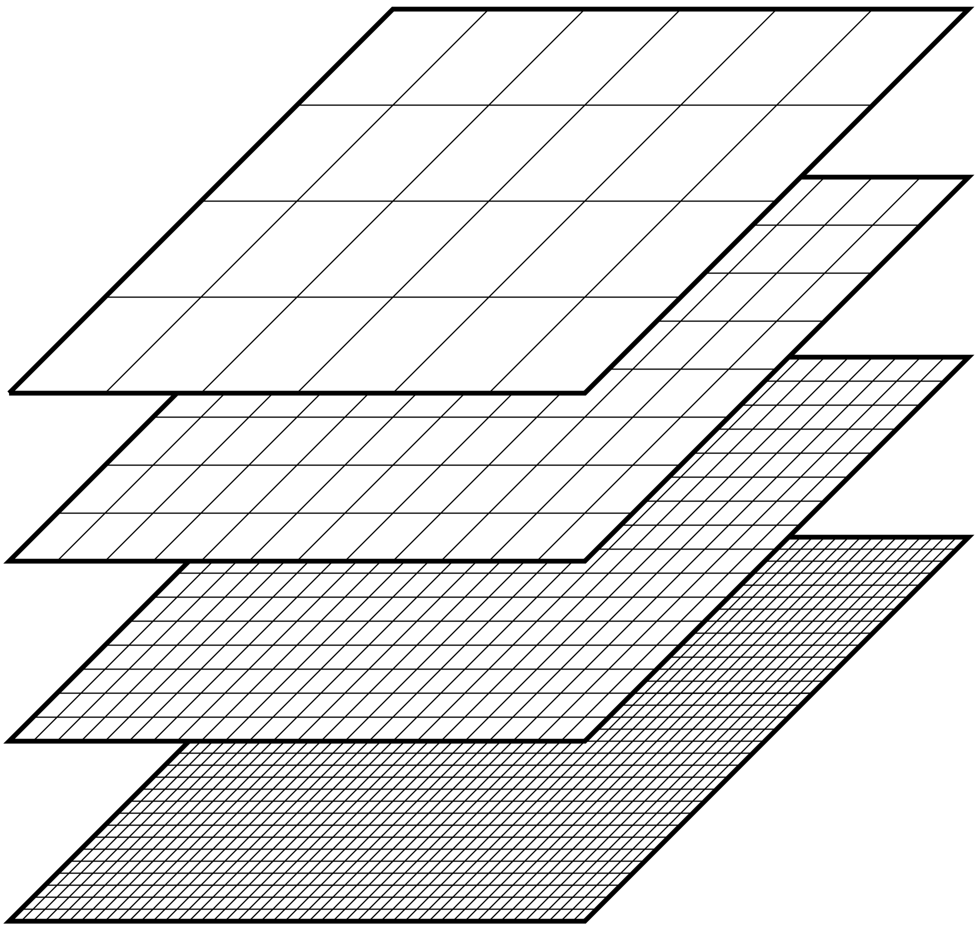


Figure 3.1: Multigrid hierarchy

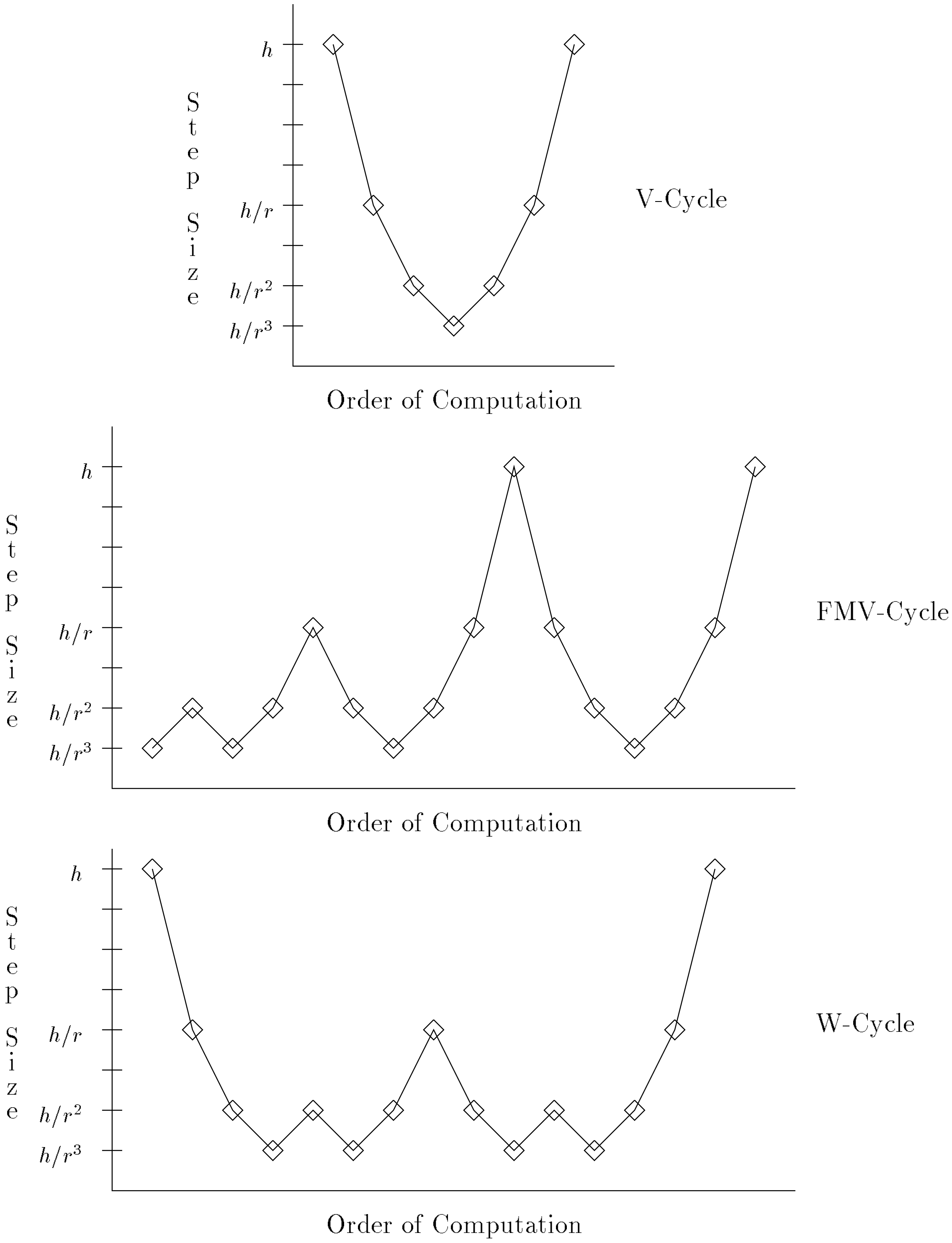


Figure 3.2: Multigrid cycle schedules

fine meshes to the finest, and then slowly moves up and down, interleaving coarser and finer, until it returns to the coarsest mesh. In some cases, the reverse strategy is used, with the majority of iterations being performed on the coarsest mesh (for example, [Mav89]).

Multigrid methods are generally applied to systems of equations on which information is propagated very quickly — for example, elliptic systems — and thus every cell affects every other cell. In contrast, other systems propagate information very slowly; for example, hyperbolic systems typically transfer information only between cells that are very close to one another during any given integration.

Adaptive strategies which rely on multiple grids of varying resolution have much in common with traditional multigrid methods, which are a special case of certain classes of structured AMR strategies. Thus, such a general-purpose AMR implementation is likely to have the added benefit of also being useful for traditional multigrid simulations, and for simulations which employ a static collection of grids of one or many resolutions.

3.3 Moving Mesh Methods

Moving mesh methods are a class of adaptive strategies that refine by redistributing mesh points, rather than by creating new meshes at different resolutions. As Miller says [Mil83],

> The MFE [Moving Finite Element] method was developed to handle those many nonlinear hyperbolic and parabolic problems that develop shocks or other sharp moving fronts.

In MFE, nodes move around the domain and are concentrated on the front; the number of nodes remains constant throughout.

The problems with MFE, and with moving mesh methods in general, are

- the methods are inherently non-uniform, so existing uniform grid solvers must be radically altered;
- the number of mesh points does not grow, but the complexity of the solution can, so a given mesh size may prove insufficient for some evolving simulation, causing some regions which require very high resolution to "steal" nodes from other regions which are then insufficiently resolved.

Thus, while moving mesh methods may be useful for a limited class of applications, they are inappropriate for a considerable portion of the simulation codes currently available. The requirement that uniform mesh codes be entirely reformulated eliminates a great many simulations, particularly those based on legacy codes, which makes moving mesh methods unattractive to many researchers.

3.4 Tree-Based Refinement

Gannon describes a mesh refinement strategy in which each cell of a mesh is split into identical subcells [Gan80]. The description of the mesh is stored in a *quadtree*-like structure (Figure 3.3). Tests of this refinement strategy produce good results. Adjerid and Flaherty

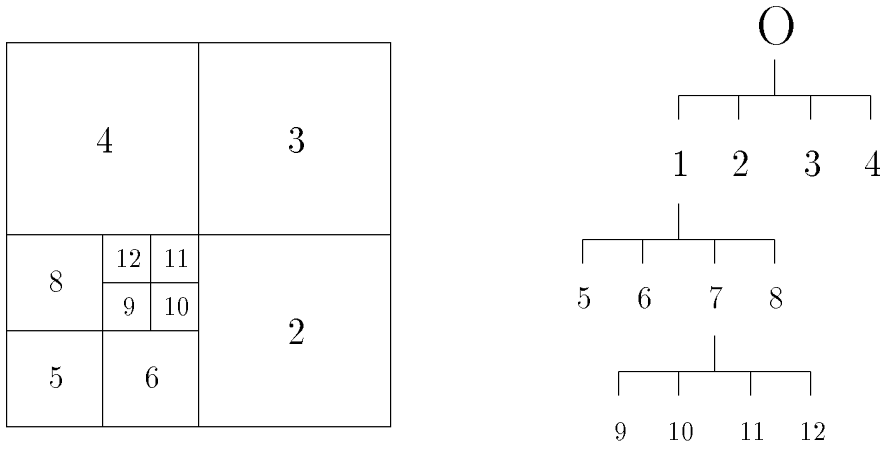
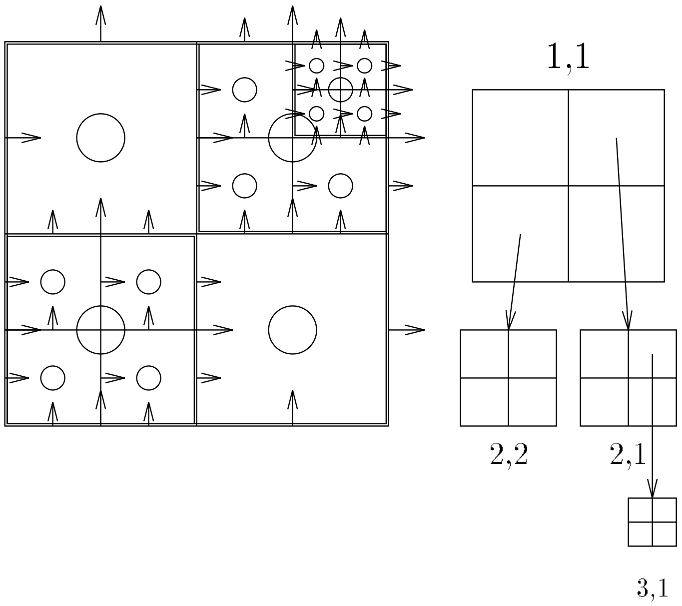


Figure 3.3: Mesh and tree of quadtree-based refinement strategy

[AF88] describe a similar method.

However, the constraint on the refinement ratio, in which cells are exactly halved in each direction, is insufficiently flexible, since it fails to permit the arbitrary scale control necessary for sophisticated multiscale systems. If a simulation depicts phenomena that have a small number of widely disparate length and time scales — for example, a large portion of the universe, a cluster of galaxies and a galaxy — then these scales must be implemented by many intervening levels, requiring considerable storage space and computation time, much of which will be wasted in intermediate and uninteresting scales. On the other hand, if the simulation requires a variety of scales based on optimizing the adaptation, then the appropriate refinement factors must be accessible.

Thompson, Leaf and Van Rosendale describe a similar approach on a staggered, adaptive, multilevel grid, which they use to solve incompressible Navier-Stokes equations [TLVR91]. As in the approach of Adjerid and Flaherty, this strategy employs a quadtree structure

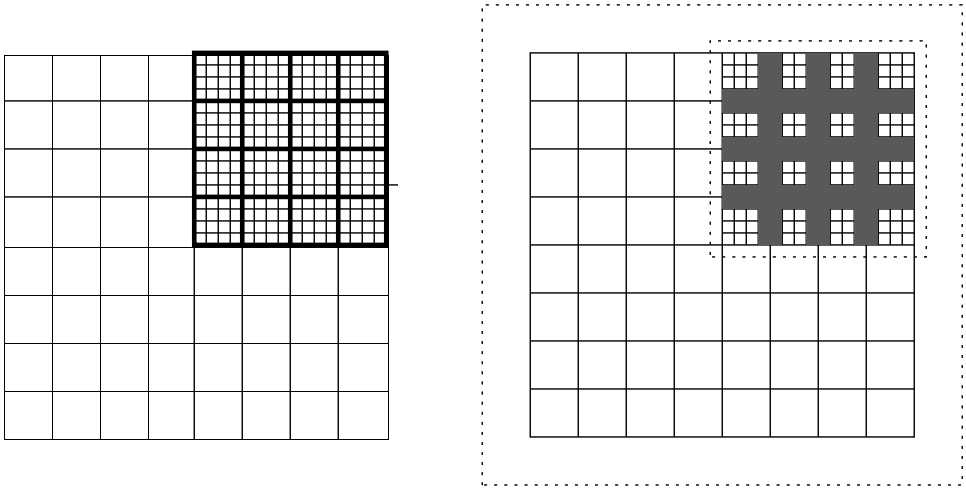


RELATIONS	1,1	2,1	2,2	3,1
NEXT	NIL	2,2	NIL	NIL
PARENT	NIL	1,1	1,1	2,1
NEIGHBOR	NIL NIL NIL NIL	1,1 NIL NIL 1,1	1,1 1,1 NIL NIL	NIL NIL 2,1 2,1
KID	2,1 2,2	3,1	NIL	NIL
AUNT	NIL NIL NIL NIL	NIL NIL NIL NIL	NIL NIL NIL NIL	1,1 NIL NIL 1,1

Figure 3.4: Mesh and tree of Thompson, et al.'s refinement strategy

(Figure 3.4). The strategy begins with a single grid — a *patch* in the authors' terminology — that is adaptively refined in a quadtree-like manner. Each cell to be refined on a given level l becomes a patch on level $l+1$, complete with its own set of interrelationships with its parent, its parent's siblings — the authors call them *aunts* — its own siblings — the authors' term is *neighbors* — and its children. More significantly, each patch has its own ghost boundary space. Thus, after all patches at level l obtain their boundary values based on these interrelationships, the patches can in principle be computed simultaneously, to obtain the new iteration's solution.

However, because the size and arrangement of patches on each level is arbitrary, and each patch contains its own boundary space, the boundary space of the patches grows arbitrarily with the spatial and scale complexity of the phenomena being examined. Thus, considerable



(a) Patch Interfaces (thick lines)   (b) Wasted Boundary Space (shaded)

Figure 3.5: Boundary space in a quadtree

time must be spent passing information between patches. In addition, if boundary space is added to each patch, considerable memory is required to contain all of the boundary space. A great deal of the boundary space can become redundant, if the patches cover a large, contiguous region (Figure 3.5). (The authors avoid the memory usage problem, so the primary concern is communication time, which is of considerable significance on many massively parallel processor and cluster architectures, in which not only the amount of data communicated but also the number of times communication is initiated can significantly increase the overall time to solution. In fact, one of the authors' test cases demonstrates the severity of the communication problem, with sizable regions in which this property holds.)

For many hardware architectures, this approach is not ideal; the main problem is the size of the patches. Most architectures employ some equivalent of pipelining, in the sense that they can most quickly perform a series of operations on large, physically contiguous regions of memory. Thus, maximal optimization is obtained by performing an operation over the largest possible region of memory. For each level, each pipeline is applied to a subset of the patches, one at a time, and the solution method must be able to make the best possible advantage of the pipelining. However, many pipelines produce the best results when applied to memory regions of considerable length. For example, the Connection Machine CM-2 architecture is best applied to an integer multiple of the number of processors, which might well be in the thousands; the Cray C90 is best applied to vectors of length $64n$, $n \geq 1$. Thus, even in integration schemes that can pipeline an entire multidimensional grid, the refinement may need to be very large in order to obtain reasonable speedup from optimization, and in directional sweep schemes, the refinement will have to be excessive. Further, these constraints will be almost entirely hardware dependent, requiring a complete modification of refinement ratios for each new machine. But perhaps most important, these hardware-imposed refinement ratios may have little to do with the scales of the physical phenomena of interest.

However, the value of this AMR strategy is clearer in massively parallel architectures, which may have relatively small local memories associated with many processors. Once all patches have obtained their boundary values from their neighbors, each patch can be solved independently from all of the others, so this approach leads to a natural decomposition that can be quickly and easily distributed among a large number of procesors. However, in these architectures, the number of patches should be np, $n \geq 1$, for p the number of processors, in order to maximally balance the load. Because patch size is fixed for each level, it is difficult to regulate the number of patches at each level; rather, the number of cells to be refined at each level is arbitrary and changes during the simulation. Thus, an ideal balancing of patches within processors — specifically, an equal number in each processor — is difficult. However, this problem is not overwhelming, since it will require at most one undersized (and thus wasteful) subset of patches distributed among a subset of the processors.

3.5 Moving Local Uniform Mesh Refinement

Gropp describes a class of mesh refinement strategies called *moving local uniform mesh refinement* or *MLUMR*. In these strategies, a coarse uniform grid is overlayed with finer uniform grids. The finer grids move, in the sense that the subdomain that they cover changes as the solution evolves. In his original strategy, developed in the late 1970's and early 1980's, Gropp used a single level of refinement created along a shock front [Gro80]. This refinement produced results as accurate as those with a uniform fine mesh but took about half the time and saved considerable memory, if a sufficiently high refinement level was used.

In Gropp's more recent strategy [Gro87], many levels of refinement are possible, but individual grids are important entities in and of themselves. In this approach, each grid has a velocity with which it moves within its parent (Figure 3.6). Thus, regridding need only occur when a grid crosses from one parent to another, or when two grids collide and therefore

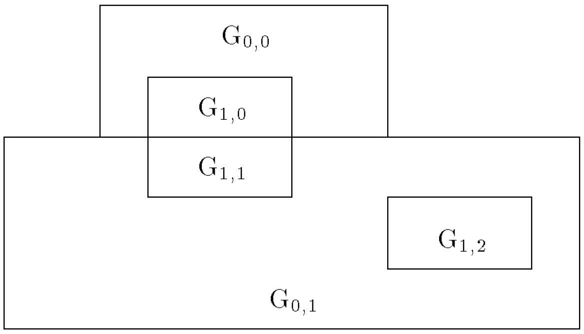


Figure 3.6: Grid nesting in Gropp's MLUMR strategy

join. However, the relationship between a grid and its parent is constantly changing. Gropp uses this refinement strategy in order to take better advantage of pipelining capabilities, by providing maximal data locality.

3.6 Summary

Adaptive mesh refinement techniques provide insight into the computational advantages and disadvantages of structured meshes. Specifically, the simple, implicit connectivity of structured meshes is both boon and bane: it is inherently easier to optimize operations on such meshes, because they guarantee data locality, but it is also far more difficult to adaptively refine them.

Traditional multigrid methods are an instructive example for studying cycling schedules. While these methods are not adaptive, they illustrate the ways in which meshes of varying resolutions can interact with one another, in order to solve unwieldy problems cooperatively. Thus, these methods give rise to a more complete understanding of multiresolution requirements, a crucial issue for some classes of adaptive strategies.

The various structured adaptive refinement strategies that have previously been developed depict many of the issues that any structured AMR strategy must address. The pitfalls include minimizing wasted space, ensuring sufficiently large subgrids to take advantage of optimization capabilities, the need for flexibility in every aspect of the refinement strategy, and the difficulty of adjusting existing simulation kernels to match the requirements of the adaptive approach. To the extent that an AMR strategy addresses these concerns, it may considered effective and efficient.

Chapter 4

Overview of Berger's Adaptive Mesh Refinement Strategy

Since the early 1980's, Marsha Berger has been developing an adaptive mesh refinement strategy for structured meshes based on the notion of multiple, independently solvable grids, all of identical type but each of arbitrary size and shape. She began this work with Joseph Oliger at Stanford University [BO84] and has continued her research at the Courant Institute for Mathematical Sciences at New York University, and the Research Institute for Advanced Computer Science at NASA Ames Research Center, with other collaborators, including Jameson [BJ85a], [BJ85b], Colella [BC89], Bokhari [BB87], Bell, Saltzman and Welcome [BBSW91], Aftosmis and Melton [AMB95], and Rigoutsos [BR90].

4.1 Premise

The underlying premise of Berger's strategy is that all grids of any given resolution that cover a problem domain are equivalent in the sense that, given proper boundary information, they can be solved independently by identical means. In essence, the multigrid concept is adjusted, reducing it from the highly accurate but computationally expensive set of increasingly finely resolved grids, each of which covers the entire domain, to a set of resolution *levels*, each of which employs a disjoint set of subgrids to cover progressively less of the domain. Because the grid nesting is constantly changing, Gropp considers Berger's AMR scheme to be an MLUMR strategy [Gro87]; however, in Berger's AMR the grids do not move as such, but rather are replaced with other grids that may cover only slightly different regions.

4.2 Layout of the Hierarchy

Berger's AMR strategy features a *hierarchy* of resolution *levels*, each of which contains a set of *grids* (Figure 4.1). Berger's original implementation represents the hierarchy as a directed graph that is acyclic on relationships between levels (e.g., from parent to child and vice versa) but cyclic over all relationships (Figure 4.2[1]). Every grid is completely covered by some non-empty set of parent grids; both its active computational interior and its ghost boundaries are covered, except those portions of the ghost boundary that lie on the exterior of the overall computational domain. In addition, the finer grids *abut* the coarser cells, so

[1]The figure is the candidate's reproduction of a portion of Figure 5.1 on page 500 of [BO84].

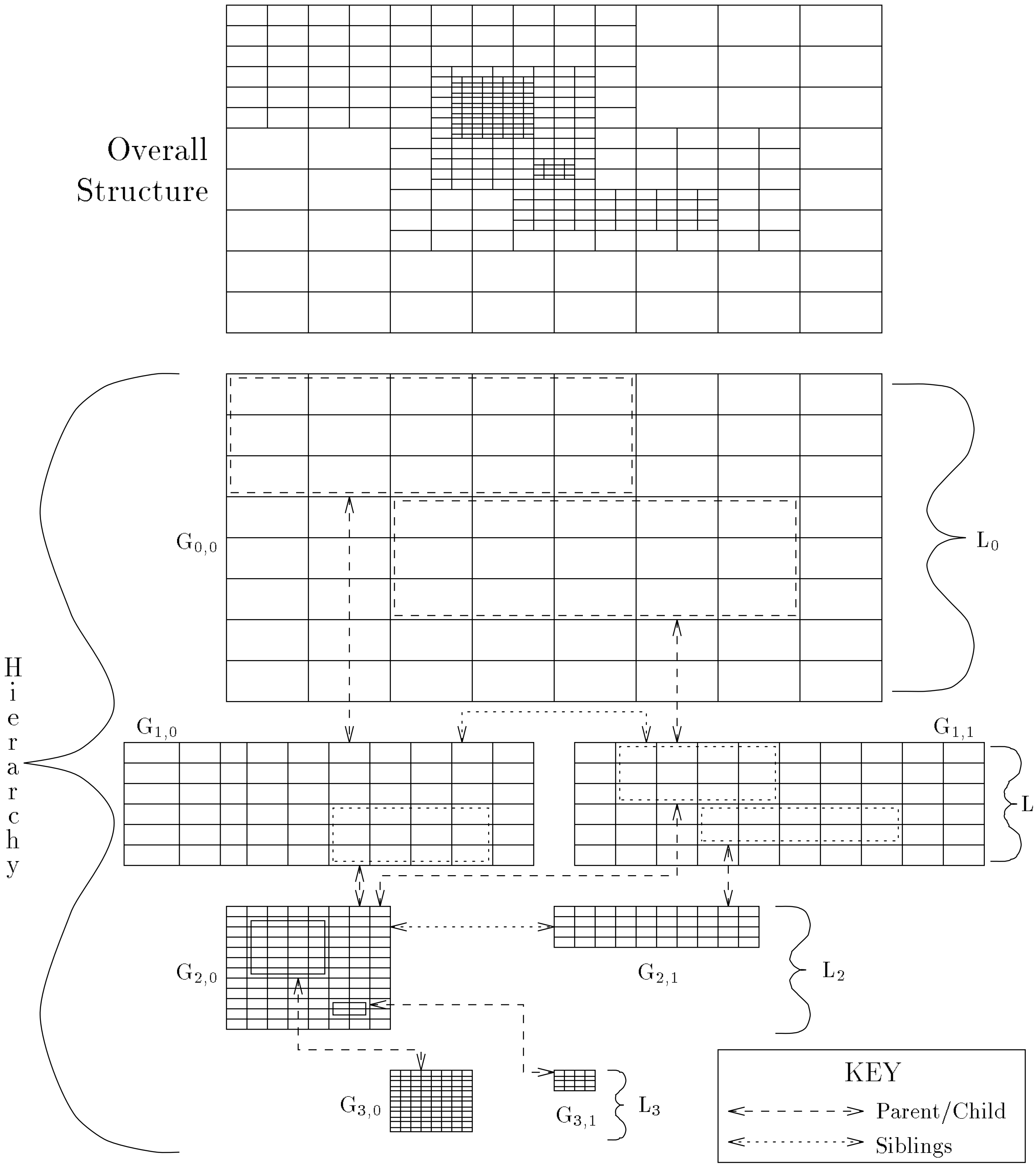


Figure 4.1: A grid hierarchy

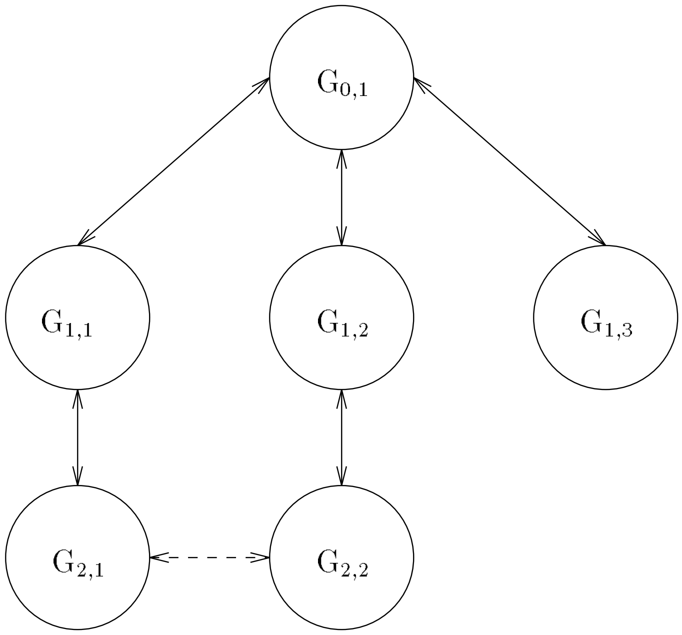


Figure 4.2: Graph representation of grid hierarchy

that the number of non-boundary cells along each axis of each finer grid is an integer multiple of the refinement factor.

At the root level, each grid consists of a computational interior and a ghost boundary region. At all finer levels, each grid consists of: a region of interest that has been refined from the immediately coarser level; a buffer region, which allows the existing grids to continue to cover fully the phenomena of interest, even if they travel, until the next regridding; and the ghost boundary region (Figure 4.3). In some cases, the buffer region may be partially absorbed if the grid abuts another grid at the same level (Figure 4.4). In fact, it is conceptually more appropriate to consider the buffer as surrounding the region of interest, which may be distributed among several grids, rather than as surrounding the subregion of interest

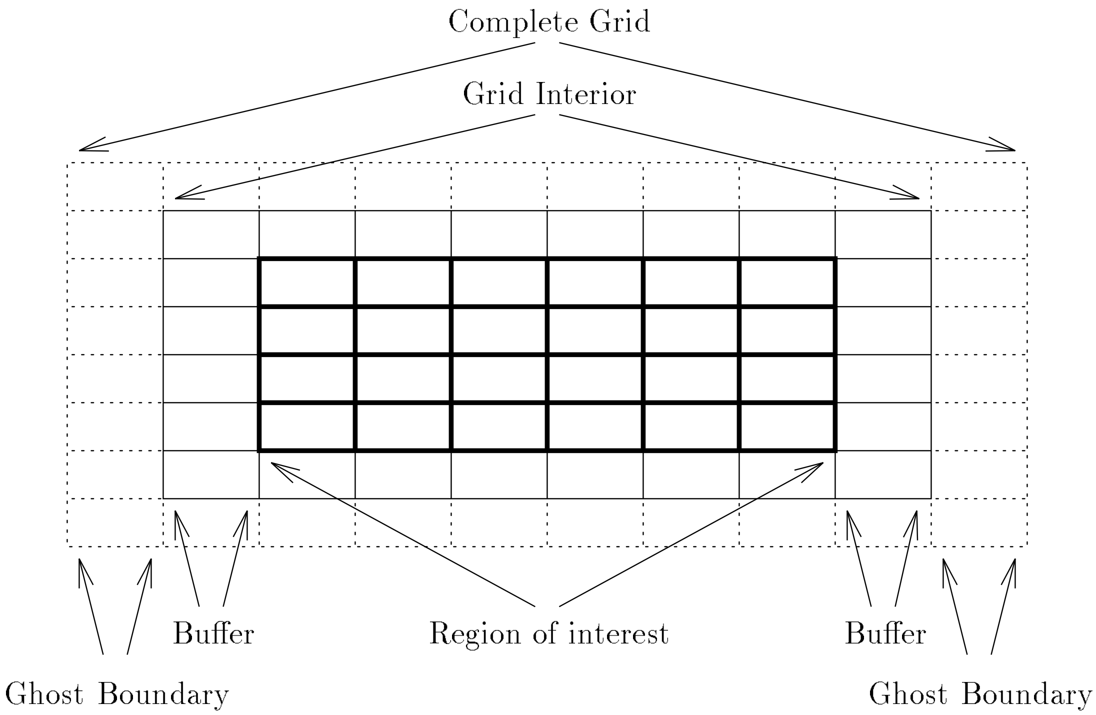


Figure 4.3: Parts of a grid

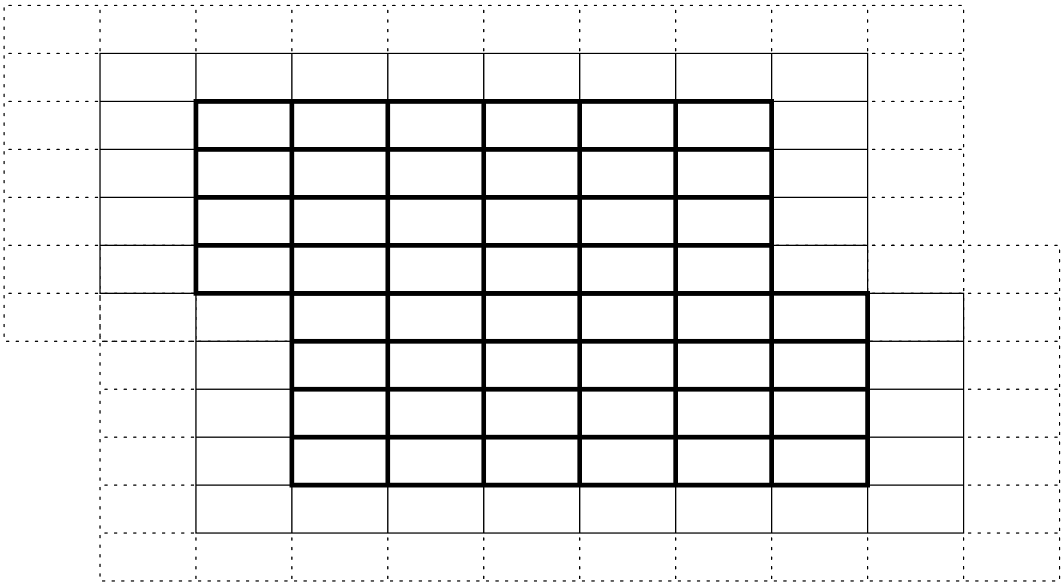


Figure 4.4: Abutting grids with partially absorbed buffer regions

that a specific grid contains. In any case, each grid will retain its full boundary region, even if it is overlapped by other grids at the same level.

4.3 Interpretations of Adaptive Mesh Refinement

Berger's strategy for adaptive mesh refinement can be interpreted in two different ways: either as a series of increasingly fine virtual grids overlaying the same domain, or as zooming in on a particular subdomain of interest. These interpretations give rise to differing approaches in various aspects of the adaptation, particularly in selecting regions for refinement.

4.3.1 Virtual Grids

In the virtual grid case (Figure 4.5), a root level, coarsest set of grids covers the entire domain. Overlayed on the root grids is a virtual finer grid, which also covers the entire domain. However, this finer grid is implemented by a set of non-overlapping subgrids that cover only those subdomains of the domain requiring the higher resolution, according to the refinement criteria. This method of placing finer subgrids over coarser grids can be repeated recursively, either to some predefined maximum resolution, or until the refinement criteria no longer exceed the threshold at some level.

The finer subgrids on each level directly represent the finer virtual grid of that resolution on the subdomains they cover, while the coarser grids indirectly represent the finer virtual grid on the rest of the domain. Solution vector values of the finer virtual grid that are

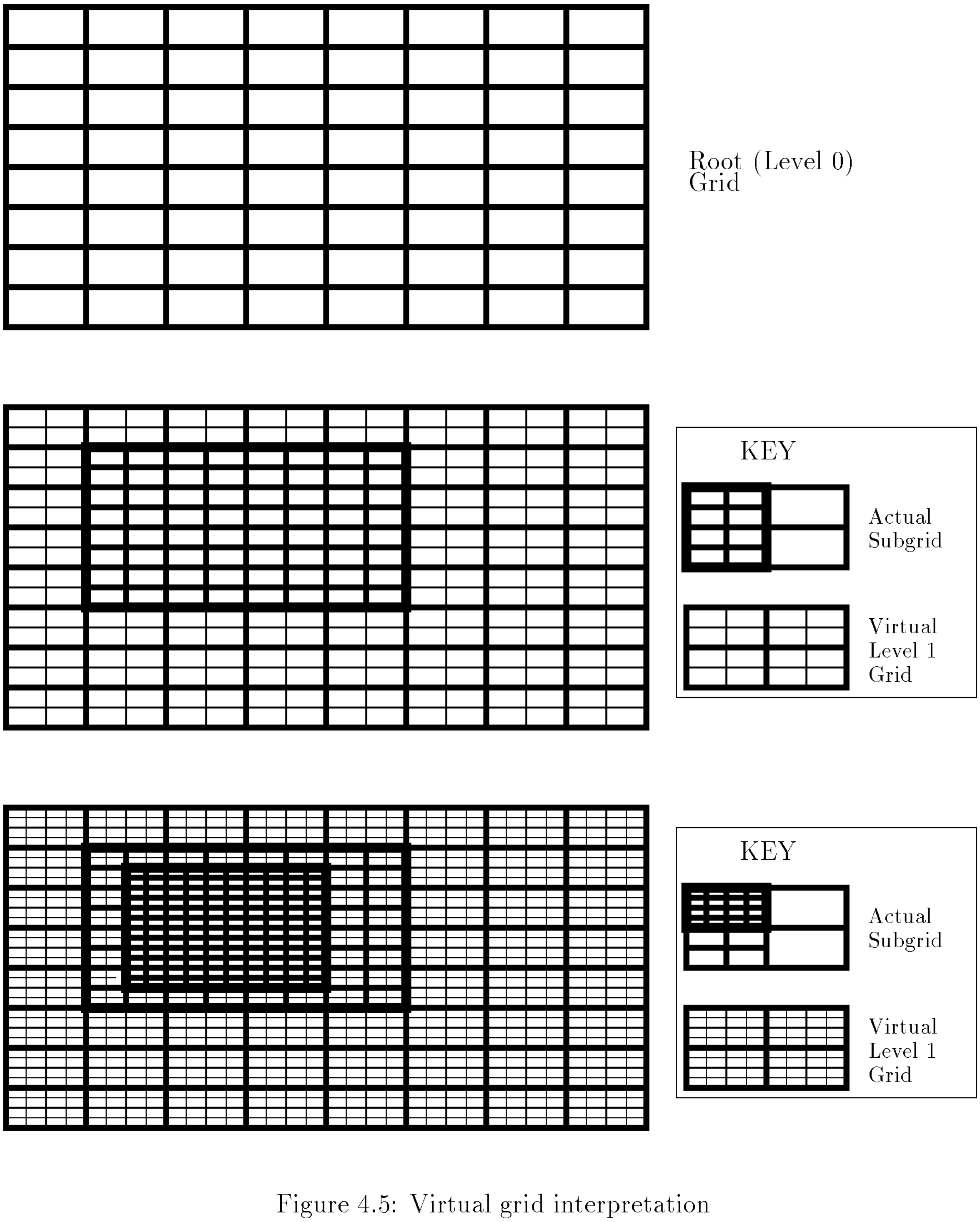


Figure 4.5: Virtual grid interpretation

contained in actual finer subgrids are obtained directly; the rest are obtained by injecting from coarser grids as necessary.

Ultimately, the purpose of this approach is to represent every portion of the domain with the minimal resolution required to satisfy some criteria. Therefore, this interpretation of Berger's strategy is most amenable to selection criteria which are automatic. For example, the collection of grids may cover the domain such that, for all points in the domain $P = (x, y)$, the truncation error $\tau_P < \varepsilon$, where ε is the refinement threshold.

An example of an application for which this interpretation is appropriate is the movement of a shock front through a fluid [BC89]. In this case, the grid loci that contain the shock must be computed at high resolution, in order to ensure low error for all solution values (Figure 4.6).

4.3.2 Zooming Grids

The zooming interpretation approaches adaptive mesh refinement quite differently. Here, the increasingly fine subgrids cover an increasingly small subdomain, or a small collection of disjoint subdomains, within the overall domain, and these subdomains are explored in more detail than the rest of the overall domain. (Figure 4.7). This approach to AMR is difficult to perform automatically, because it often applies to domains that contain many redundant phenomena, all of which would be selected by automatic refinement criteria. It is most naturally achieved by interactive selection of refinement regions.

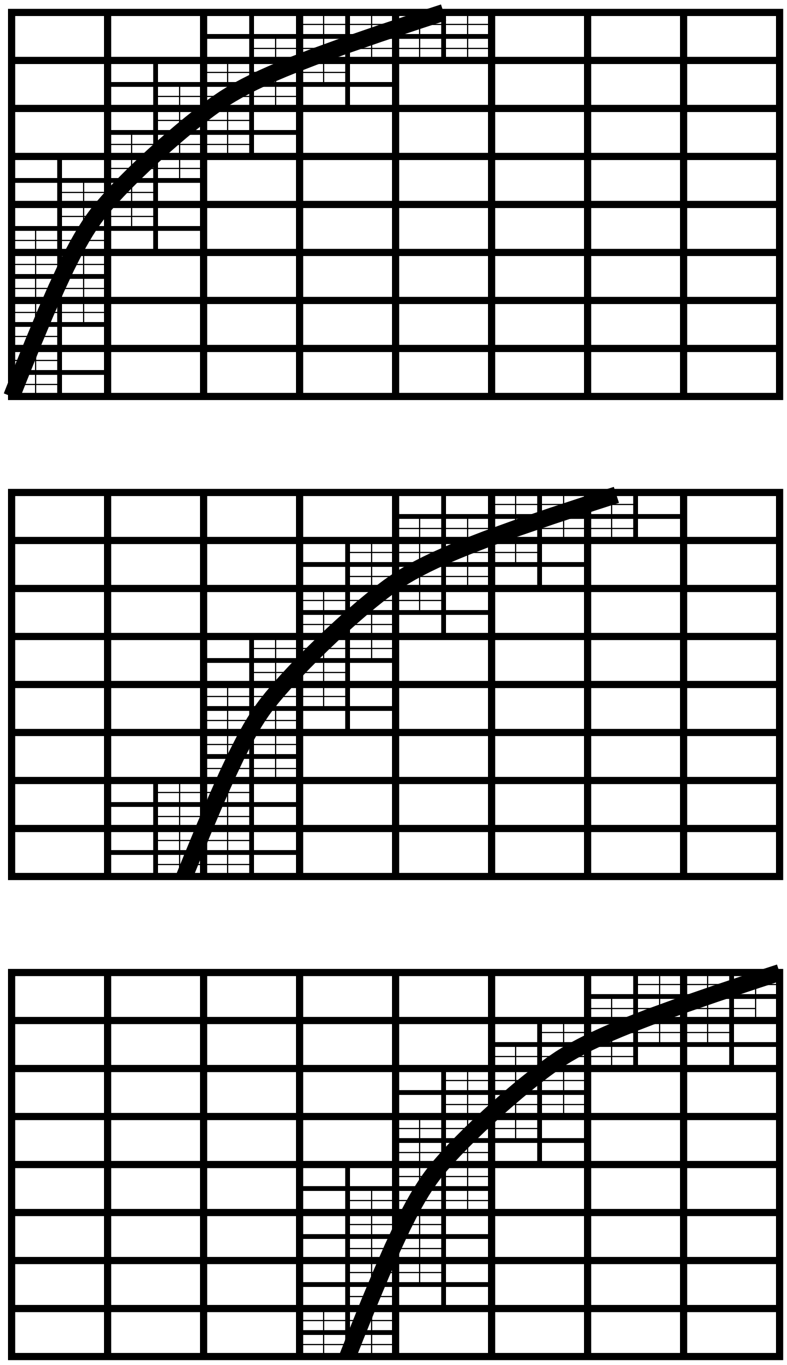


Figure 4.6: AMR on a moving front

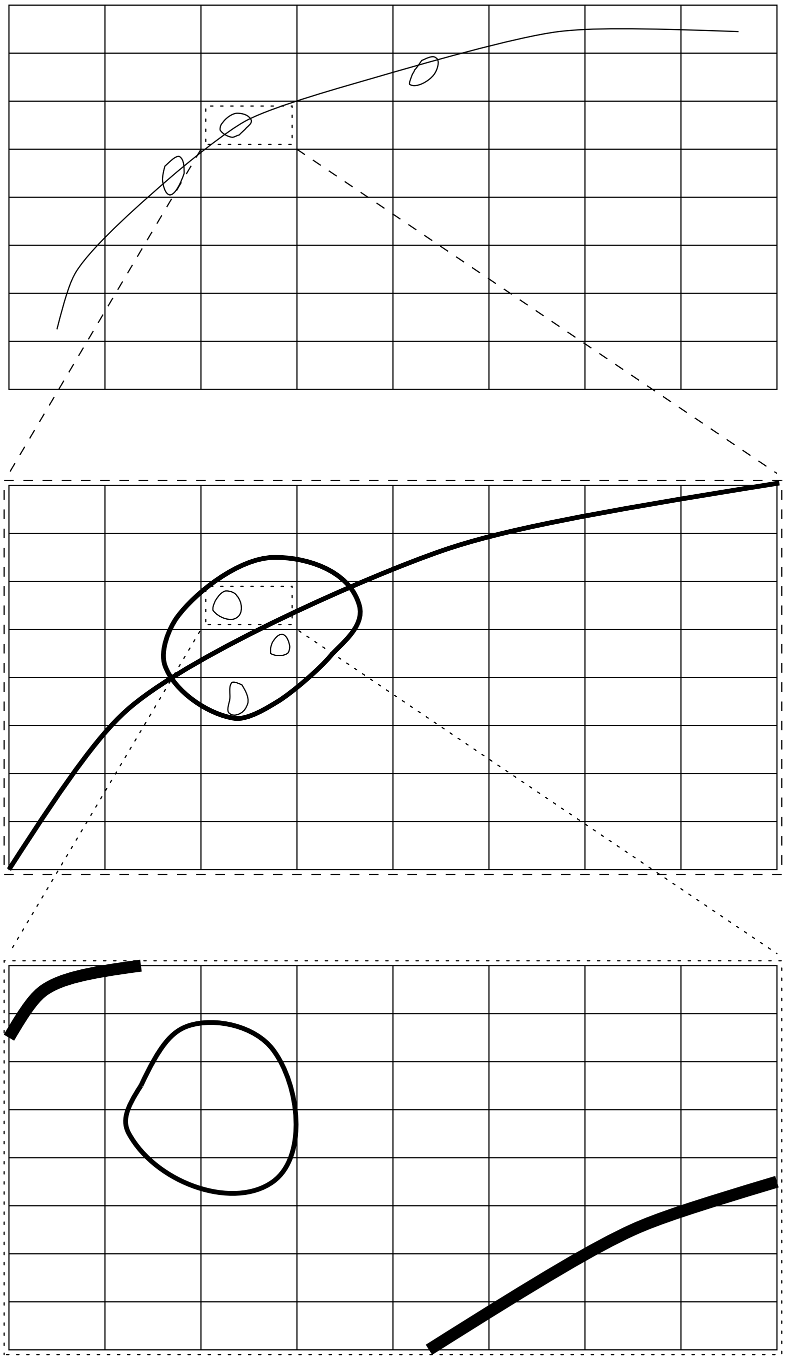


Figure 4.7: AMR zoom

4.4 Berger's AMR Algorithm

Berger's AMR scheme employs the nested hierarchy of grids to cover the appropriate subdomain at each level. The integration algorithm recurses through the levels, advancing each level by the appropriate time interval, then recursively advancing the next finer level by enough iterations at its (smaller) time interval to reach the same physical time as that of the newest solution of the current level:

```
Integrate (level)
begin
  Evolve(level)
  if "level isn't finest" then begin
    for r = 0 to time_refinement_factor - 1 do
      Integrate(level + 1)
  end
end
```

Thus, the order of the integrations is a generalized W-cycle (Figure 3.2), with integrations at each level recursively interleaved between iterations at coarser levels (Figure 4.8). An important effect of the integration order is that, as a general rule, the overwhelming majority of computing time is spent on the finest level, as a direct result of the fact that Berger's AMR refines in time as well as in space: if the refinement factor between a finer level $l+1$ and the next coarser level l is r, then grids on the finer level $l+1$ will be advanced r timesteps for every coarser timestep. For a d-dimensional domain, the grids at level $l+1$ must cover the same portion of the computational domain as only $1/r^d$ coarser cells at level l, in order to consist of the same total number of cells for the level, because every coarse cell covers r^d

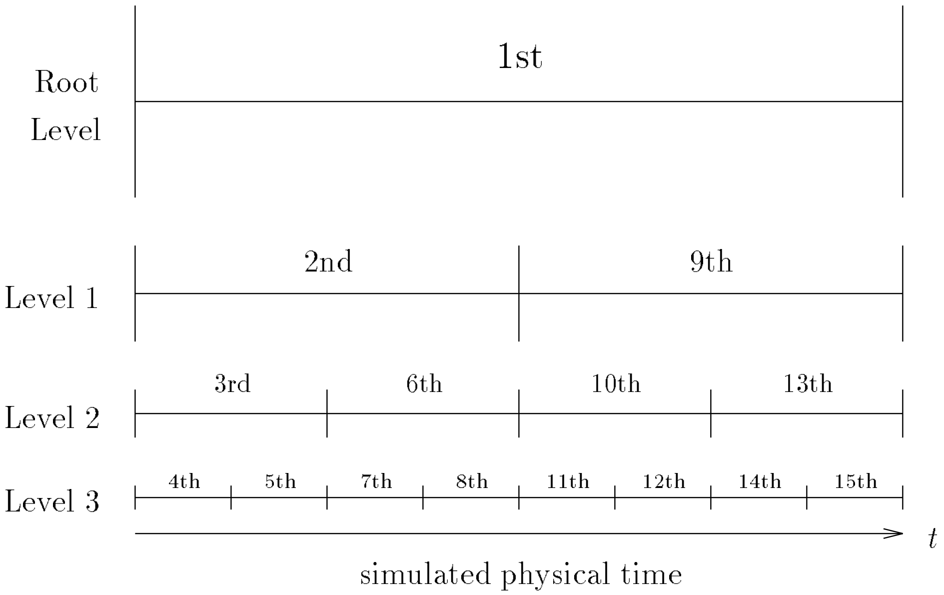


Figure 4.8: Integration order in Berger's AMR scheme

fine cells. Taking into account the r finer timesteps per coarser timestep, it is clear that the timesteps on the finer level will take more computation time than one coarser level timestep unless the finer level includes no more than $1/r^{d+1}$ as much of the computational domain as the coarser level. For example, using a refinement factor of two on a three-dimensional domain, two iterations at level 1 will take more computation than an iteration at the root level (which comprises the entire computational domain) unless the grids at level 1 cover no more than 1/16 of the domain.

Integration requires five operations:

- boundary value *collection*, from parents, siblings, and the exterior of the computational domain, as appropriate;

- *evolution*, to advance the solution in time;

- flux-based *correction*, to ensure conservation at the interfaces between those coarse cells that are overlapped by fine cells and those that are not;

- *projection*, to improve the solution values on coarse cells from the overlapping fine cell values;

- *refinement*, to place grids appropriately for the evolved condition of the solution.

Thus, a more precise expression of the integration algorithm is:

```
Integrate (level)
begin
  if "time to refine" then Refine(level)
  CollectBoundaryValues(level)
  Evolve(level)
  if "level isn't finest existing" then begin
    for r = 0 to time_refinement_factor - 1 do
      Integrate(level + 1)
  end
  IncrementTime(level)
  if "level isn't finest existing" then begin
    CorrectFluxes(level, level + 1)
    Project(level, level + 1)
  end
end
```

4.4.1 Collection of Ghost Boundary Values

Values for the ghost boundary cells surrounding each grid's active region are collected from three sources (Figure 4.9), as appropriate: by injecting from parents on the immediately

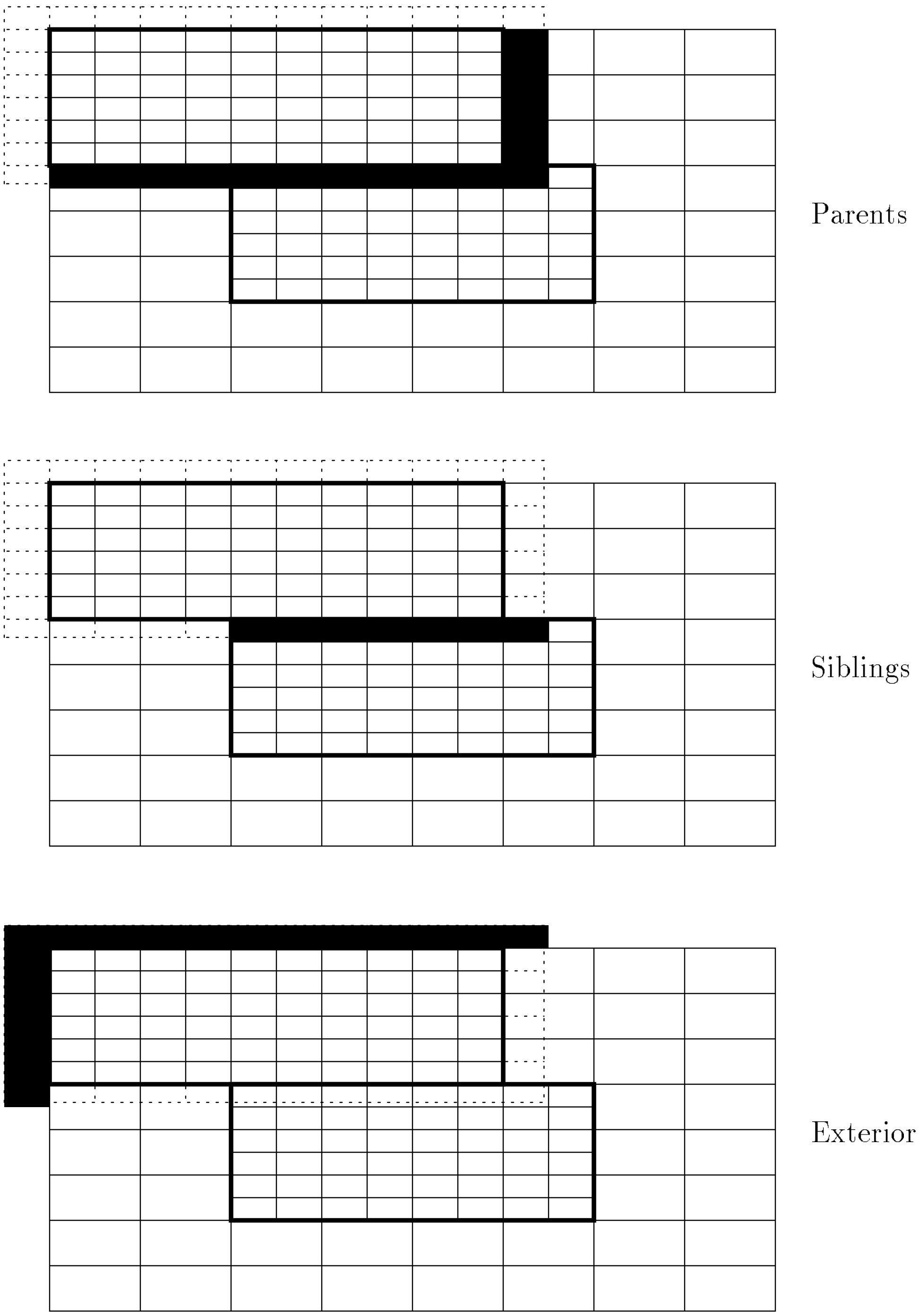


Figure 4.9: Sources of boundary values

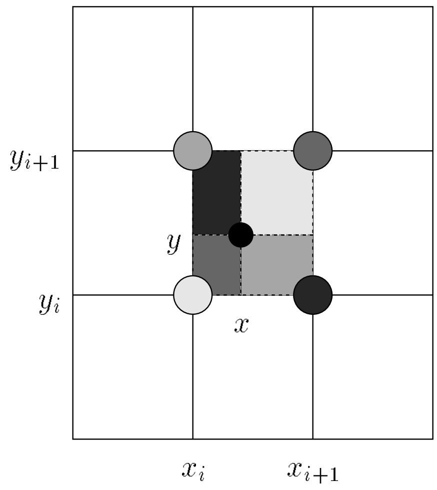


Figure 4.10: 2D Area-weighted linear interpolation

coarser level, by copying from siblings at the same level, and by extrapolating from the exterior of the computational domain.

Injection transfers boundary values to a finer grid based on the values of its coarser parent, typically by an interpolation. For example, a common type of injection is *linear volume-weighted* interpolation, in which the values of the surrounding parental loci are weighted by the diagonally opposite relative volume (shown in 2D in Figure 4.10). There are also a variety of more sophisticated interpolation schemes, including higher order and conservative schemes.

A fundamental principle of boundary collection is that a grid at a non-root level requires maximal *coverage*: the entire active region of the grid, and as much of its boundary region as possible, must be covered by some set of parent grids at the immediately coarser level, except for those boundary regions exterior to the computational domain (Figure 4.11).

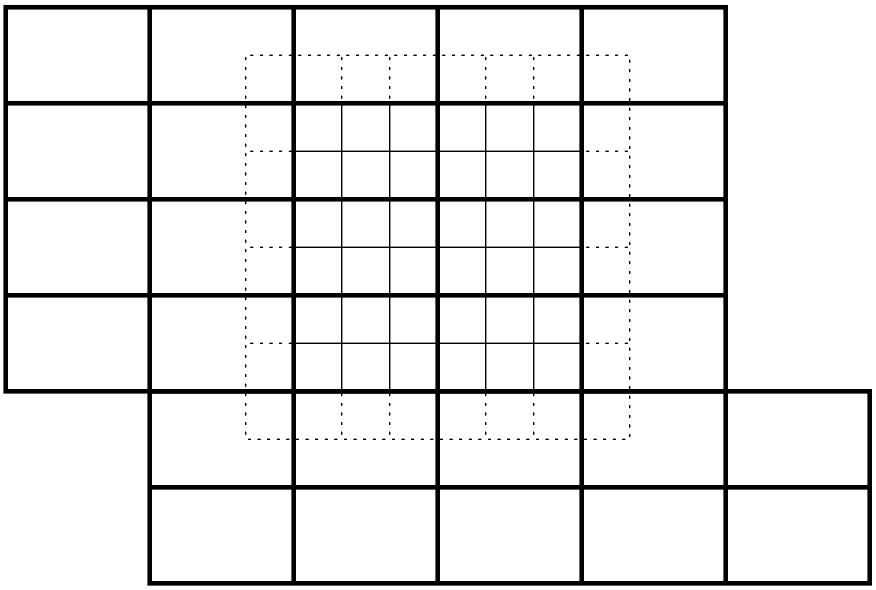


Figure 4.11: Grid coverage

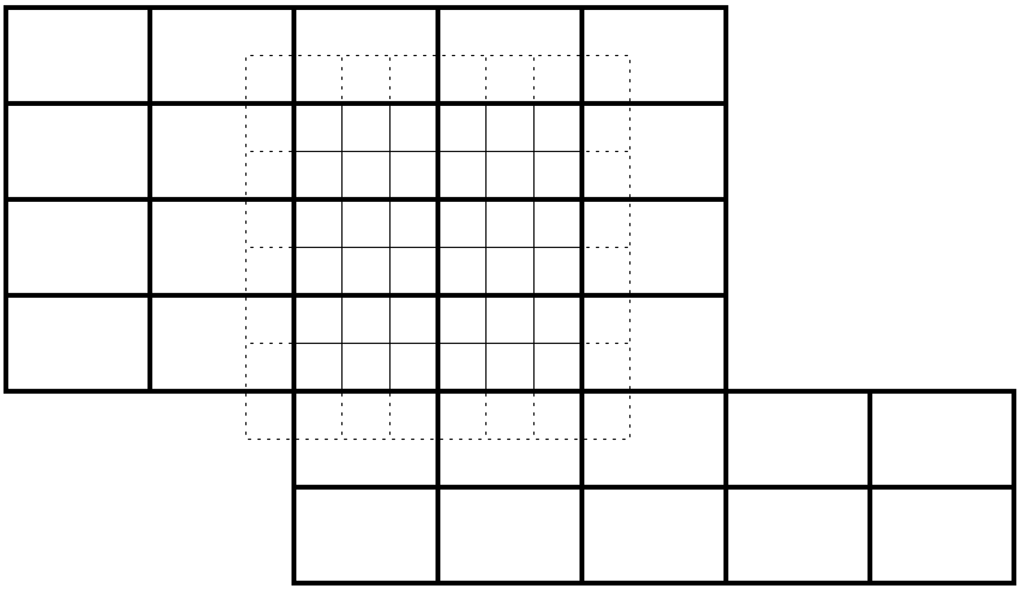
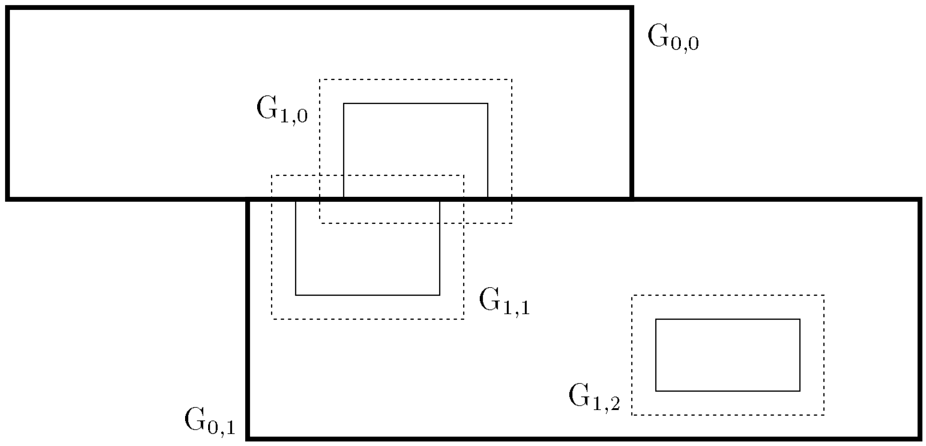


Figure 4.12: Grid with incomplete coverage

Coverage is required because it promotes the principle of the *maximally accurate state*: for every call to the solver, the grid's (non-exterior) boundary is at least as accurate as the immediately coarser level. Coverage is fundamental because, unless it is guaranteed, it is possible for a grid to obtain its input values from much less well resolved levels (Figure 4.12). For a d-dimensional application of maximum depth $l_{\max}$ and a refinement factor of r between each, each covered grid has (non-exterior) boundary values of no worse than $1/r^{d+1}$ as well refined as its interior (computed) values, since the refinement is over d spatial dimensions and one time dimension. This property holds at any level, from coarsest to finest, though at the coarsest level the (non-exterior) boundary resolution is identical to the computed resolution. An uncovered grid, on the other hand, may have (non-exterior) boundary values only $(1/r^{d+1})^{l_{\max}}$ as well resolved as its computed values, because it may need to draw some of its boundary values from as far away as the root level.

When a grid's boundary region is overlapped by another grid's computed interior, the former grid can obtain boundary values from the latter *sibling* grid by simple copying. Naturally, this case is ideal, because it provides the best resolution possible for that ghost boundary cell. However, many of the grids at a particular level will obtain some or all of their boundary values from other than sibling interiors. The only way to guarantee that all (non-exterior) boundary values are obtained from siblings is to refine the entire domain — which would defeat the purpose of AMR. In this context, a grid's *siblings* are only those grids at the same level that overlap the grid's ghost boundary region (Figure 4.13).

Superficially, it might appear that an appropriate source of boundary values would be



$G_{0,0}$ is the parent of $G_{1,0}$'s interior and is a parent of $G_{1,1}$'s boundary.
$G_{0,1}$ is the parent of $G_{1,1}$'s interior and is a parent of $G_{1,0}$'s boundary.
$G_{1,0}$ and $G_{1,1}$ are siblings.
$G_{1,2}$ has no siblings, despite the fact that it shares a parent with $G_{1,1}$.

Figure 4.13: Parents and siblings

grids at finer levels. The finer grids, after all, are even more well resolved than are their parents. However, closer inspection of the AMR algorithm and integration order reveals that drawing boundary values from children is not necessary, and would involve redundant calculation. The reason is that each timestep at a level l is followed recursively by r timesteps at level $l+1$ for some refinement factor r. At the end of these r finer timesteps, the finer grids have caught up in physical time to the coarser level — that is, $t_{l+1} \equiv t_l$ — and then the finer values from level $l+1$ are projected onto the coarser level l. Also, each finer grid is completely covered by some subset of the coarser grids. Therefore, before the boundary values are collected at level l, any boundary values that might have come from grids at level $l+1$ have already been projected onto their parents — which are the siblings of the grid of interest — at the end of the previous level $l+1$ integration.

The final source for boundary values is the exterior of the overall computational domain, which requires *extrapolation*: constructing data that have not otherwise been computed. In many cases, however, exterior boundary values are obtained not by extrapolation as such, but by appropriate copying, in cases such as periodic and reflecting boundaries, or by an analytic procedure, as is sometimes the case with inflow boundaries. For simplicity, however, it is convenient to consider such conditions as degenerate cases of extrapolation, and thus to refer to the process of obtaining any exterior boundary values as extrapolation.

4.4.2 Evolving the Solution

The solver or *kernel* of a simulation evolves the solution by advancing it forward by a specificed time interval. Although the solver is the lynchpin of any numerical simulation, its details are not directly relevant to AMR, because the AMR abstracts it into a "black box" operation; that is, given the correct input, the solver is expected to produce the correct output, and otherwise is not a concern of the AMR strategy.

4.4.3 Flux Correction for Conservation

Berger's AMR strategy permits conservative numerical schemes [BC89], with conservation achieved by a relatively simple adjustment to the numerical solver, and a simple, efficient flux correction step.

Specifically, the solver receives as input not only the variables it needs, such as solution

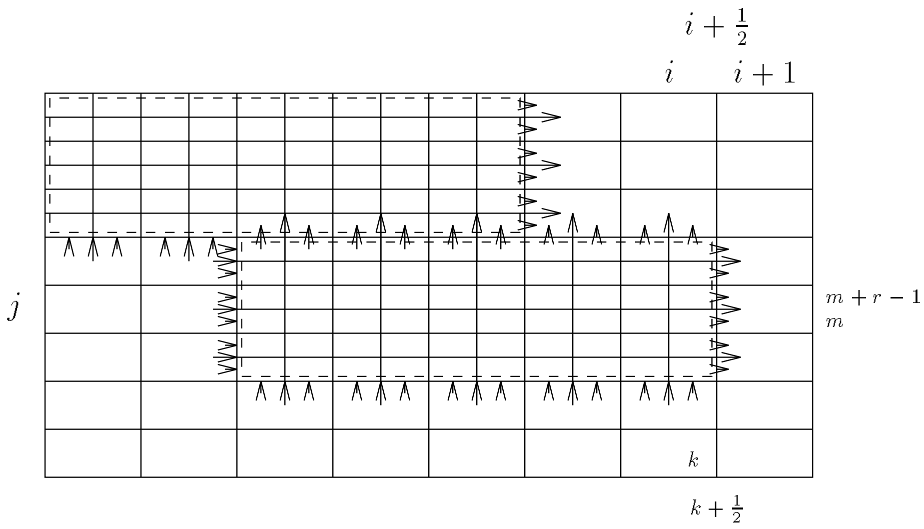


Figure 4.14: Flux correction at fine-to-coarse interfaces

vectors, grid descriptions (for example, the number of nodes along each axis), and integration arguments (for example, the time step interval), but also special arrays for storing the fluxes along the interface of each grid. These fluxes are collected after computing each timestep, in separate arrays for each grid. The fluxes for the fine level are summed over all the fine time steps that constitute a coarse time step, in a separate array, and the difference between the fine fluxes and the coarse fluxes is used to correct the coarse solution values along the fine-to-coarse interfaces.

The flux correction operation (Figure 4.14) is as follows:

- Let r be the refinement ratio.
- Let $\mathrm{u}_{i,j}^{l}$ be the solution at the center of cell i, j at coarse level l.
- Let $f_{k+\frac{1}{2},m}^{l+1}$ be the flux at the interface between cells k, m and $k+1, m$ at fine level

$l+1$.

- Let $f^l_{i+\frac{1}{2},j}$ be the flux at the interface between coarse cells ij and $i+1,j$.

- Let $F^{l+1}_{k+\frac{1}{2},m}$ be the total flux at the interface between fine cells k,m and $k+1,m$ over an entire coarse step — that is, over r fine timesteps.

- Let n^l_y be the number of coarse cells along the y-axis.

Initially, after integrating at the coarse level and storing the $f^l_{i+\frac{1}{2},j}$'s, $j=1\ldots n^l_y$, let

$$F^{l+1}_{k+\frac{1}{2},m} \leftarrow 0,\ l=1\ldots n^{l+1}_y$$

After each fine time step,

$$F^{l+1}_{k+\frac{1}{2},m} \leftarrow F^{l+1}_{k+\frac{1}{2},m} + f^{l+1}_{k+\frac{1}{2},m},\ l=1\ldots n^{l+1}_y$$

When r fine time steps have been performed, bringing the time of the fine level to that of the coarse level,

$$u_{i+1,j} \leftarrow u_{i,j} + (\sum_{p=0}^{r-1} f^{l+1}_{k+\frac{1}{2},m+p} - f^l_{i+\frac{1}{2},j},\ j=1\ldots n^l_y$$

The expressions are analogous for interfaces along other axes. (Actually, Berger gives a similar, more elaborate expression based on unitless flux values rather than actual flux through the specific cell.)

A more intuitive explanation of this flux correction procedure is that the algorithm records the amount of material — for example, mass — that passes through an interface between a fine grid and a coarse grid. When the coarse timestep is performed, the algorithm records the coarse flux, which is the amount of material traveling through the interface during the time interval of the coarse timestep. When each associated fine timestep is performed, the fluxes through the fine cells composing that interface are recorded, and a running total of these fine fluxes is maintained over the r fine timesteps that correspond to the coarse timestep (Figure 4.15). In addition to summing along the timesteps — and thus along the time axis — the fine fluxes are ultimately summed over all of the fine cell interfaces that contribute to the coarse cell interface. Thus, a total of r^d fine fluxes are summed over the r fine timesteps, because there are r^{d-1} fine cell interfaces for each coarse cell interface.

If the coarse cell and the corresponding fine cells arrive at exactly the same result, then the difference between the coarse flux and the sum of the r^d fine fluxes is zero, in which case the correction does not change the value of the abutting coarse cell. In practice, of course, the likelihood of the coarse and fine values being identical is extremely low, and in fact would be considered problematic, in the sense of indicating an unnecessary refinement. The purpose of flux correction is to adjust such a cell, on the assumption that the values on the coarse and fine cells will *not* be identical. Thus, when the fine solution values are projected onto the corresponding coarse cell, the amount of material traveling between that coarse cell and the coarse cell that abuts it is the same as would result if the abutting coarse cell also had corresponding fine cells. In this way, conservation is maintained on the fine-to-coarse interfaces.

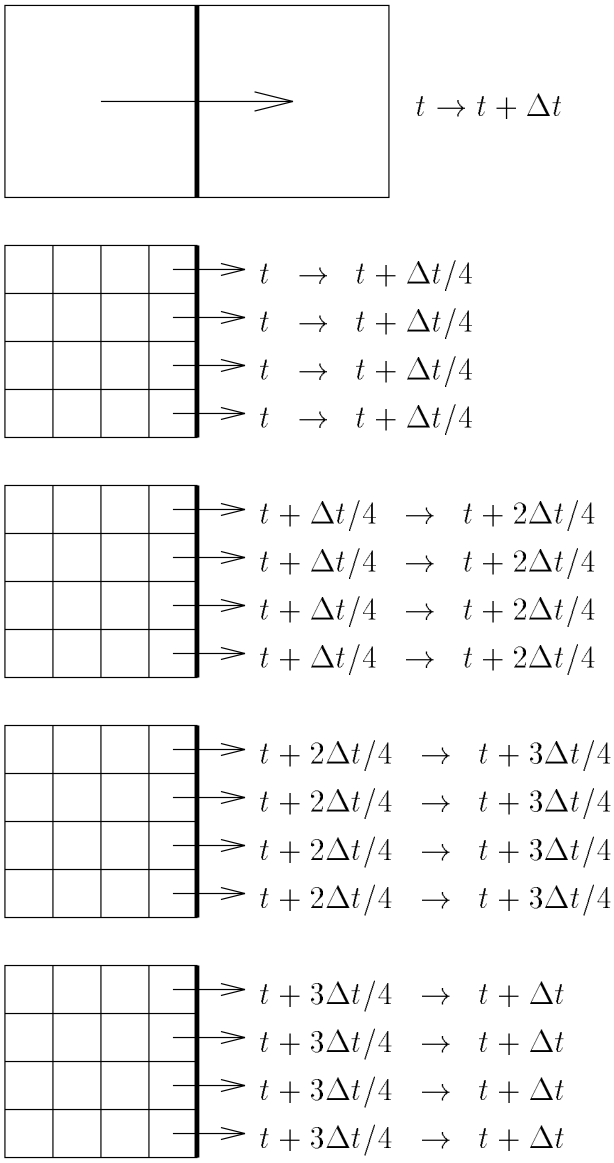


Figure 4.15: Fluxes for correction

4.4.4 Projection from Fine to Coarse Grids

Projection is the process of updating a coarser level, using the more accurate values of the cells at the finer level to replace the covering cells on the immediately coarser level. Projection can be achieved by any of a number of means, including

- copying, for example velocities at nodes;
- summing, for example masses at cell centers;
- averaging, for example densities at cell centers.

An important point here is that projection should occur *after* flux correction, rather than before, because the projected values reflect the most accurate solution available. If projection occurs before correction, then the optimal projected values may be corrected, which is obviously undesirable. If, however, correction occurs first, then any corrected values covering a finer grid will be replaced with the projected values, while those on true fine-to-coarse interfaces will remain appropriately corrected.

4.4.5 Refinement

Refinement covers subdomains with grids of higher resolution. Refinement is probably the most algorithmically complex operation in Berger's AMR strategy. Like integration, it is implemented recursively: a refinement at level l first refines level $l+1$, and so on recursively to the finest level. In this way, proper nesting, or *coverage*, of the refined regions at finer levels is ensured. The basic refinement algorithm is:

```
Refine(level)
begin
  if "level is finest allowed" then return
  if "level isn't finest existing" then Refine(level+1)
  Select(level)
  Expand(level)
  Cluster(level)
  Regrid(level+1)
  if "timestep is initial" and
     "level isn't finest allowed" and "finer level is empty" then
    Refine(level)
end
```

Even this simplified description of the refinement procedure is counterintuitive on its face, but a statement-by-statement examination may prove helpful.

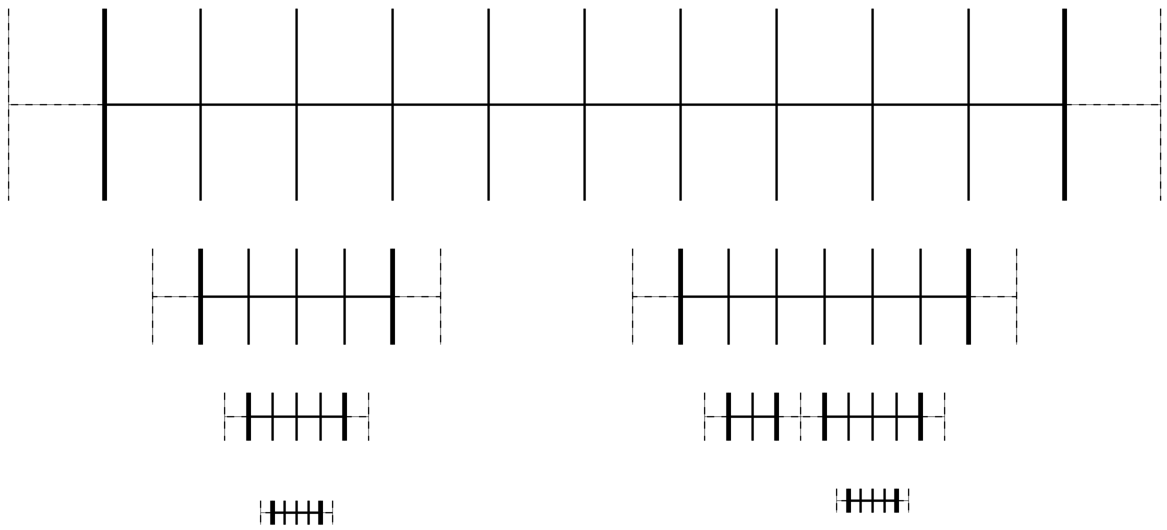
```
if "level is finest allowed" then return
```

This statement is the halting criterion of the recursion; no refinement of the finest level allowed is possible.

```
if "level isn't finest existing" then Refine(level+1)
```

In this statement, the refinement algorithm recursively calls itself on the finer level (Figure 4.16). Inductively, it may be helpful to assume that this statement works properly and produces a properly nested hierarchy for all levels finer than l. Thus, while the existing grids at level $l+1$ have not been replaced, all the grids at $l+2$ through $l_{\max}$ (if they exist) have been, and all of those levels now cover all regions of interest that are contained within the grids at $l+1$.

Before refining level $l + 1$



After refining level $l + 1$

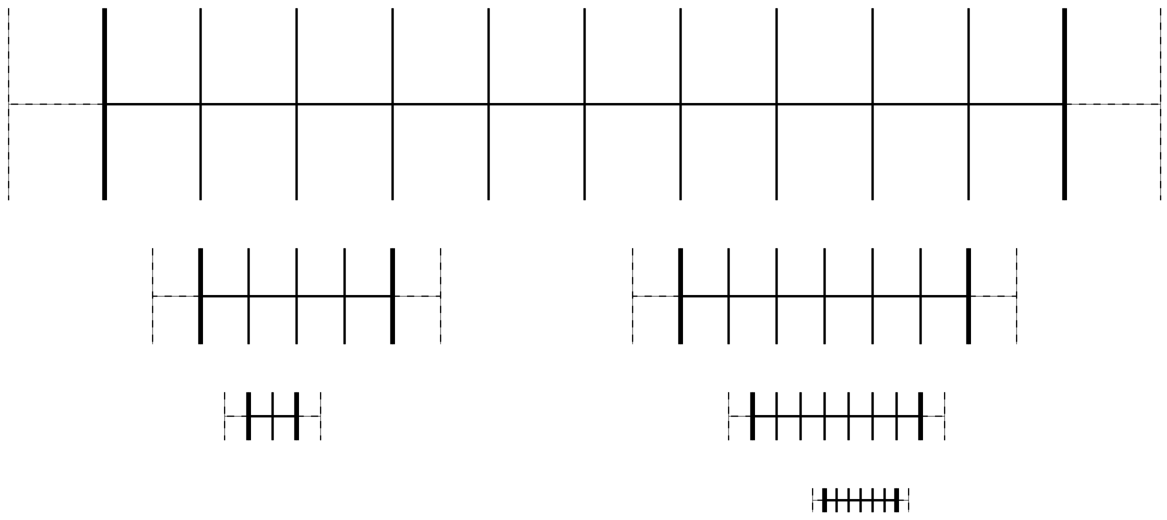


Figure 4.16: Recursive refinement of finer levels

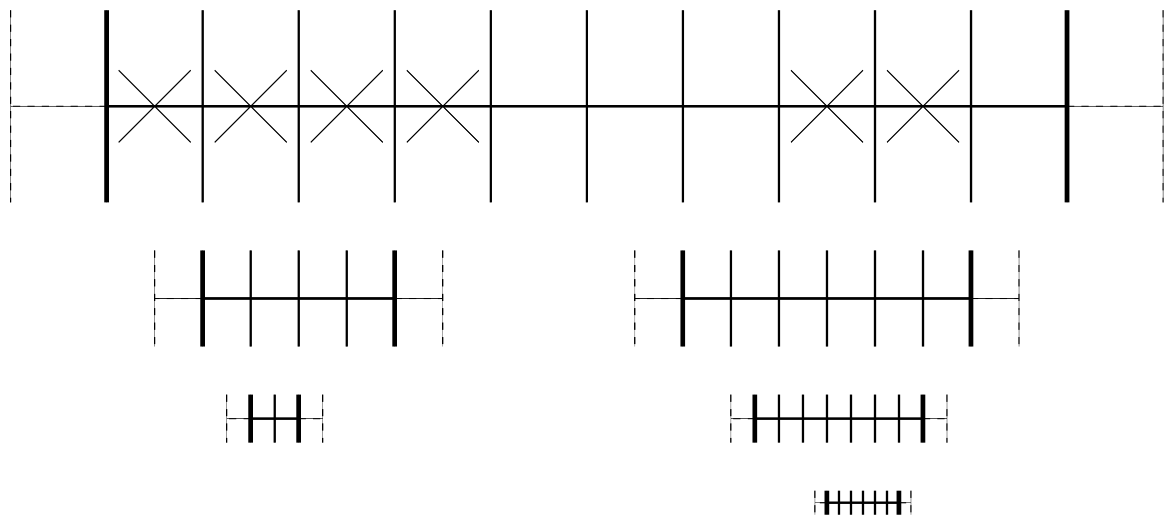


Figure 4.17: Selection of refinement regions

Next, regions of interest on level l are *selected*:

```
Select(level)
```

Selection is described in more detail in in section 4.4.6, but for the moment it is sufficient to note that the selection algorithm produces the appropriate regions of interest to be refined (Figure 4.17).

The selected regions must next be *expanded*:

```
Expand(level)
```

Expansion serves to ensure coverage of the grids at finer levels. Each selected cell is expanded by the number of cells necessary to cover the next finer level's buffer regions and the even finer level's boundary regions. In addition, the refined regions from grids at even finer levels must be covered as well, so their subdomains are mapped onto the selection regions.

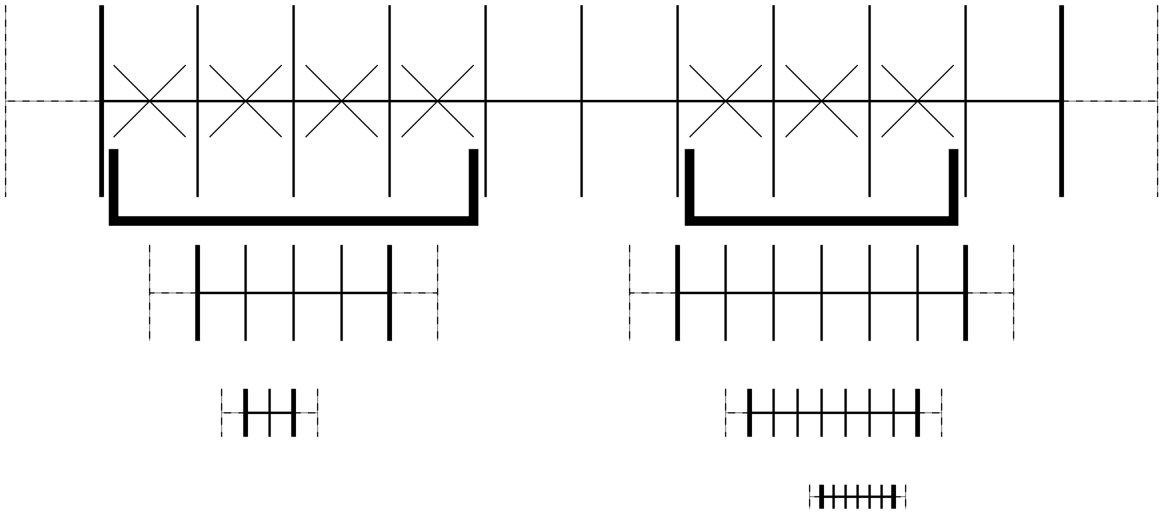


Figure 4.18: Clustering

Then, the expanded refinement regions at level l are *clustered*:

```
Cluster(level)
```

Clustering maps the set of refined loci into a set of rectangular regions, each of which represents the subdomain that a new grid will cover (Figure 4.18).

Next, $l+1$ is regridded:

```
Regrid(level+1)
```

That is, the grids at $l+1$ are replaced by grids that cover both the selected regions of refinement at level l and the newly created grids at $l+2$ through $l_{\max}$ (Figure 4.19).

The final statement of the refinement algorithm

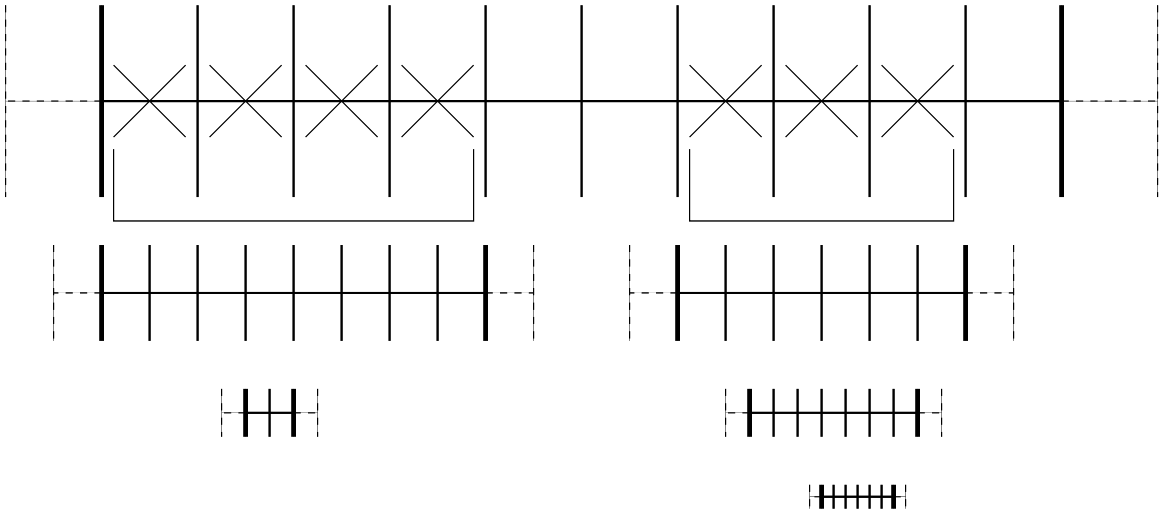


Figure 4.19: Regridding the immediately finer level

```
if "timestep is initial" and
   "level isn't finest allowed" and "finer level is empty" then
  Refine(level)
```

is used only to initialize the hierarchy. If the hierarchy is being initialized, then it is useful and appropriate to create grids of as fine resolution as required. Furthermore, when the refinement algorithm is executed at each level, there are as yet no grids at any finer levels. Therefore, the algorithm recursively creates the next level, which does not need to refine subsequent levels, since they don't yet exist. But when that level has finished being refined, it will in turn create further levels as necessary.

Thus, either the former or the latter recursive refinement may occur, but not both, since the former occurs only if there are finer grids to cover, and the latter occurs only during initialization, when there are no finer grids to cover, and when the values on the finer grids are obtained from the initial conditions, rather than injected from (possibly far-removed) coarser levels. It might be possible to dispense with the condition governing the latter recursion, but that might inject very coarse values to very fine grids, and also could result in $\mathbf{O}(2^{l_{\max}+1})$ refinements.

4.4.6 Selection of Cells to be Refined

Selection of cells to be refined is problem-specific: depending on the nature of the application to which AMR techniques are applied, a variety of selection criteria can be applied.

The simplest selection criterion is comparing a solution value to a *threshold*; those cells whose value exceeds the threshold are refined. This criterion has the advantage of being simple and quick, but it is not particularly rigorous, and it rarely represents a physical phenomenon of interest.

Another fairly simple selection criterion is comparing the *gradient* of a solution value — that is, its local rate of change — to a threshold. While this criterion is not as simple as a direct comparison of values, it is still quite simple and quick.

Berger recommends a more rigorous selection criterion, Richardson truncation error estimation:

- Let $\mathrm{u}(x, t)$ be the solution vector at position x and time t.
- Let $\mathbf{Q}_h$ be the integration operator with mesh spacing h.
- Let q be the order of accuracy of the integration method $\mathbf{Q}$.

Thus,

$$\mathrm{u}(x, t + \Delta t) = \mathbf{Q}_h \ \mathrm{u}(x, t)$$

Then the truncation error τ is estimated by

$$\frac{\mathbf{Q}_h^2 \mathrm{u}(x,t) - \mathbf{Q}_{2h} \mathrm{u}(x,t)}{2^{q+1} - 2} = \tau + \mathbf{O}(h^{q+2})$$

In other words, the truncation error estimate is obtained by advancing the solution one timestep of interval $2\Delta t$ on a grid of mesh spacing $2h$, advancing the solution two timesteps of interval Δt on a grid of mesh spacing h (Figure 4.20), and comparing the results. The method is not only an accurate error estimator, but also an intuitively appealing refinement criterion. In a sense, it asks the question: how different would the results be if the solution were evolved with the current resolution, as compared to using a different resolution? If the answer for some cell is significantly different, then the cell is refined.

In principle, this error estimator is ideal for AMR, since it uses the same solver as is used for evolving the solution. In practice, however, it can be somewhat unwieldy.

The difficulty is in the calculation of $\mathbf{Q}_{2h}\mathrm{u}$. Computing this solution is trivial: it is simply another call to the solver. In practice, however, setting the grid with mesh spacing $2h$ can be considerably more complicated.

In principle, a simple approach is a striding copy, copying every other locus to a grid of half the size, the computing a solution. If the striding copy is performed twice for each dimension, for a total of 2^d calls on grids of $1/2^d$ cells, and the partial results copied back

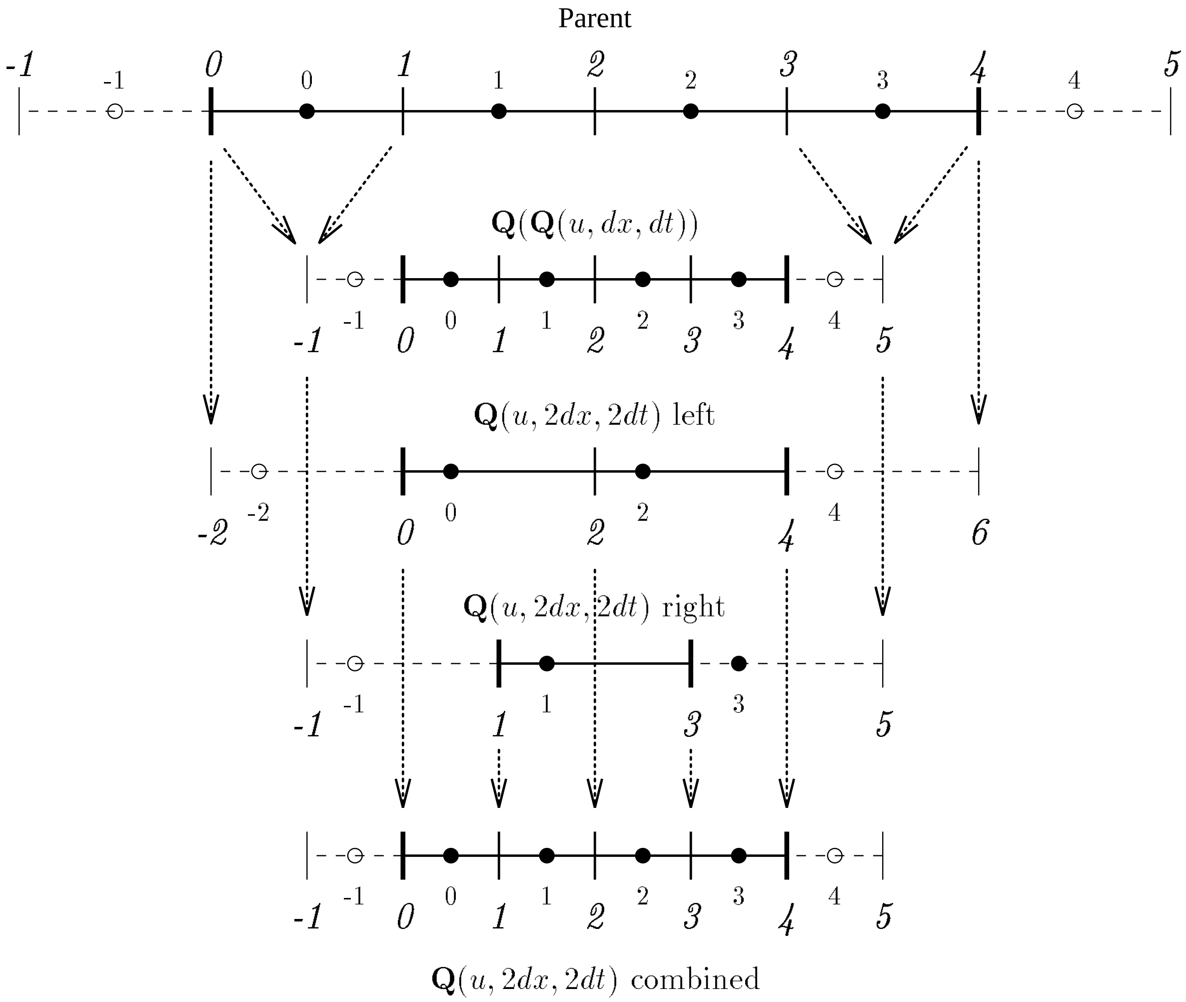


Figure 4.20: Computing the Richardson truncation error estimate

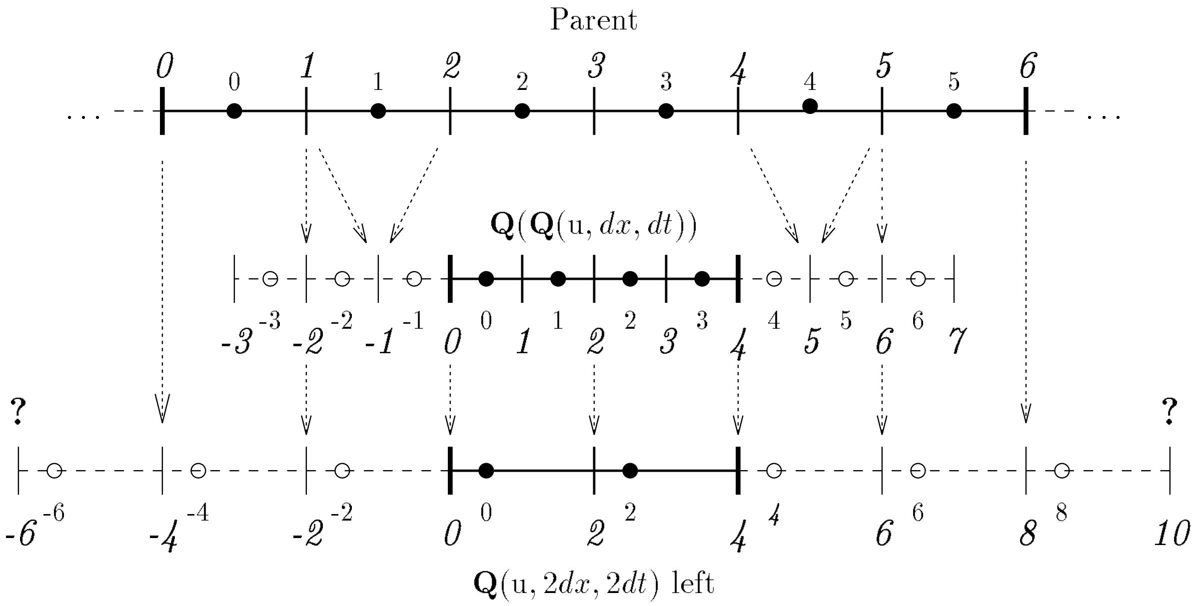


Figure 4.21: Richardson truncation error estimation boundaries

into appropriate holding fields, then the $\mathbf{Q}_{2h}$u solution is obtained. While perhaps somewhat cumbersome to code, this set of striding copies is not particularly complicated.

However, the boundary zones must also be obtained at mesh spacings of $2h$ (Figure 4.21), and this constraint can be problematic. In order to obtain boundary values, either the computational interior of the grids must be expanded, or the boundary values must be obtained from the boundaries of the parent grids. The former approach is more expensive, the latter is less accurate.

Not coincidentally, some AMR implementations — including Berger's original AMR code — make significant adjustments to the body of the solver to simplify computation of $\mathbf{Q}_{2h}$u. To some extent, however, such alterations defeat one of the primary advantages of Berger's AMR strategy: that it can be quickly and easily applied to new applications, by incorporating their solvers and other relevant routines with a minimum of recoding.

4.4.7 Clustering Algorithm

The clustering algorithm of Berger and Rigoutsos [BR90] (Figure 4.22) is based on a method used in computer vision and pattern recognition. The algorithm is as follows: first, a minimal bounding box is placed around the cells that have been flagged to be refined. Next, a *signature* list is computed along each grid axis. A signature Σ_i is the number of flagged entries along the i^{th} grid surface; e.g., a grid line in 2D, a grid plane in 3D, and so on:

$$\Sigma_i = \sum_j \sum_k f_{ijk}, \quad \text{where} \quad f_{ijk} = \begin{cases} 1 & \text{if cell } ijk \text{ is flagged} \\ 0 & \text{otherwise} \end{cases}$$

Then, any zero entry in the signature list of any axis is a grid surface perpendicular to that axis that contains no flagged cells, and thus this zero entry is a potential cutting index. The best of these zero entries — the most central along some axis — is used as the cutting index to subdivide the domain.

In many cases, the signatures are all nonzero, yet these signatures are often obtained from arrangements of flagged cells that clearly can be further decomposed. In this case, the Laplacian second derivative of the signatures

$$\Delta_i = \Sigma_{i+1} - 2\Sigma_i + \Sigma_{i-1}$$

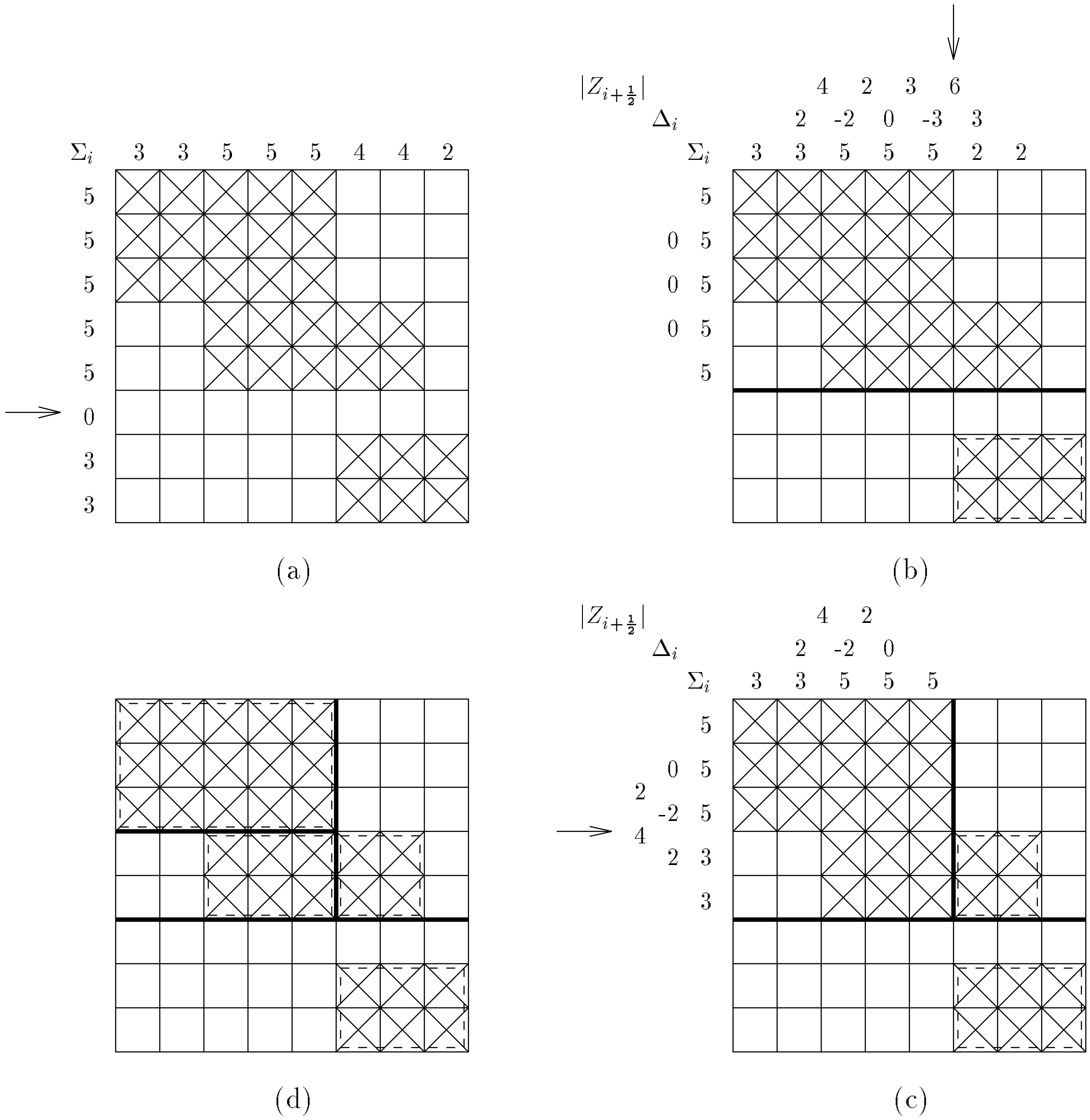


Figure 4.22: Example of clustering by signatures (clockwise from upper left)

is used. A *zero crossing* is an index where Δ changes sign, and any zero crossing is a potential cutting index. The best zero crossing is the one whose magnitude

$$Z_{i+\frac{1}{2}} = |\Delta_{i+1} - \Delta_i|$$

is largest, and this index is chosen as the cutting index.

The clustering algorithm recurses on each subregion; thus, the algorithm traverses a k-d tree [Ben75] with specially defined cutting criteria. Recursion continues until some halting criteria are satisfied. Typical halting criteria for a cluster are:

- the cluster is at least some minimum efficiency threshold c, in the sense that the ratio of the number of its cells that are flagged for refinement to the total number of cells in the cluster is at least c;

- further partitioning of the cluster would produce subclusters too small to be optimized.

In some cases, recursion continues even if the regions already produced meet the halting criteria; for example, the algorithm may wish to produce clusters of no greater than a specified size, so any cluster larger than that is bisected along the longest axis, in much the same manner as the cutting criterion for a standard k-d tree.

An important point here is that the clustering algorithm operates entirely in computational space. Thus, it is not only more efficient, since it operates in $\mathbf{O}(n \log n)$ time on the number of cells, it is also easily adaptable to curvilinear grids.

4.4.8 Regridding

Regridding creates new grids at a finer level to cover the selected area of refinement at the immediately coarser level. In principle, the operation is very simple:

```
Regrid(level)
begin
  if (newclusters > 0) then begin
    CreateNewGrids(level)
    if (level > 0) then
      InjectInteriorsFromParents(level, level-1)
    if (oldclusters > 0) then
      CopyInteriorOverlaps(level)
  end
  if (oldclusters > 0) then DeleteOldGrids(level)
end
```

Thus, the values on the new grids' computational interiors are at worst one level of resolution less well resolved than the old grids that they replace, and typically most of the values are actually copied directly from the old grids, so very little information is lost. In fact, if the size of the buffer regions is carefully chosen, the new grids will be sufficiently resolved everywhere (Figure 4.23). This condition arises because the cells of the old grids that do not overlap the new grids are, by definition, the regions where the solution no longer needs to be so finely resolved. In contrast, the cells of the new grids that are not overlapped by the old grids, and that are therefore outside the old grids' buffer regions, are — for carefully chosen buffer sizes — outside the region of interest of the new grids, since the purpose of the buffer regions is to anticipate the movement of the phenomenon of interest and to continue to cover it until the next regridding.

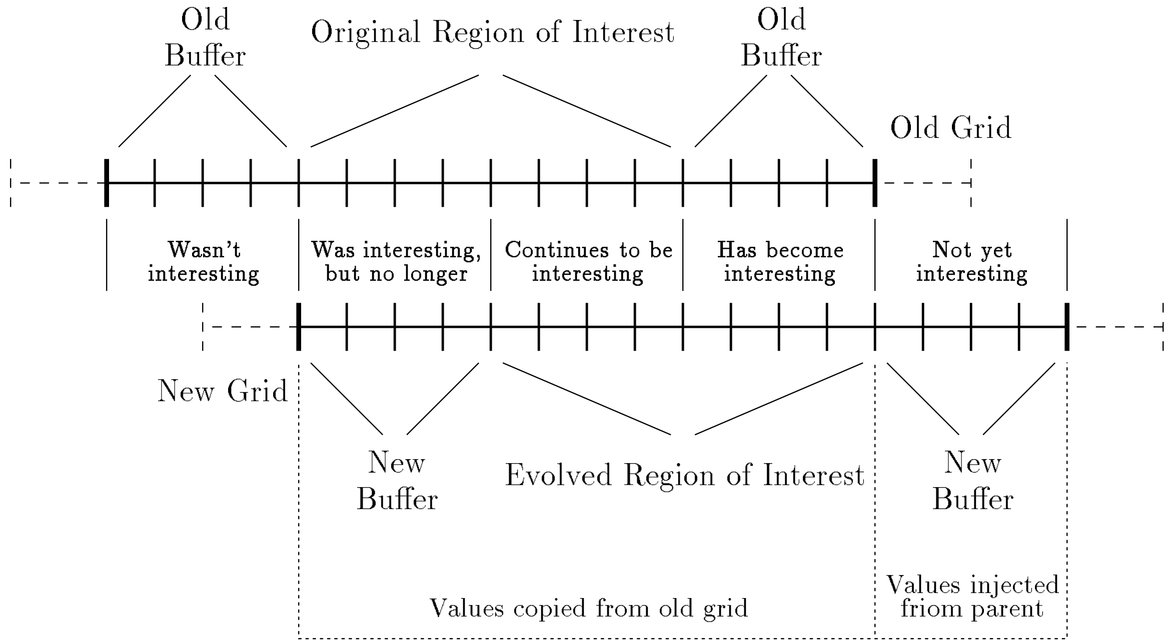


Figure 4.23: Copying from old to new during regridding

4.5 Evolution of Berger's AMR Strategy

Berger's AMR strategy has changed in several significant ways over the last decade. Some aspects of the system have been improved, some altered and some completely eliminated.

When Berger first implemented her strategy [Ber83], it had several properties that no longer hold:

- all grids were isotropic;
- grids at the same level could overlap;
- grids could be rotated with respect to the coordinate axes;
- the clustering algorithm was inefficient;

- variables were located at the nodes.

4.5.1 Allowed Mesh Types

The first concern, that grids could be only isotropic, was easy to dispense with, by adding program mechanisms which had the capability of recognizing and addressing rectilinear and curvilinear grids [BJ85b].

4.5.2 Overlapping Grids

In Berger's original strategy, grids at the same level of resolution were allowed to overlap one another, creating a situation that can give rise to two significant problems: waste and flux overcorrection.

First, overlapping grids are wasteful. This issue is not merely a matter of a few extra cells at the level of the overlap. Rather, overlapping has a subtler and more significant impact: at finer levels, entire complicated structures can be duplicated [Bry96a] (Figure 4.24), consuming not only considerable additional memory, but also a great deal of computation time, because the majority of computation is on the finer levels.

Second, overlapping grids complicate flux correction. During the correction step, each grid contributes its flux values to the flux correction of its parent(s). When two grids overlap, the flux correction is performed twice around the region of overlap (Figure 4.25), unless great care is taken to ensure that this overcorrection does not occur.

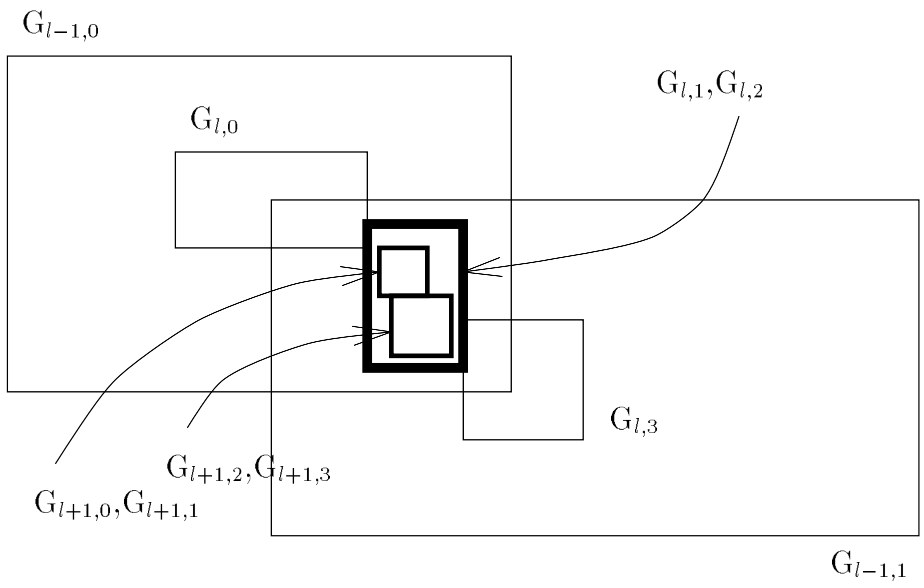


Figure 4.24: Duplicated fine structure in an overlap region

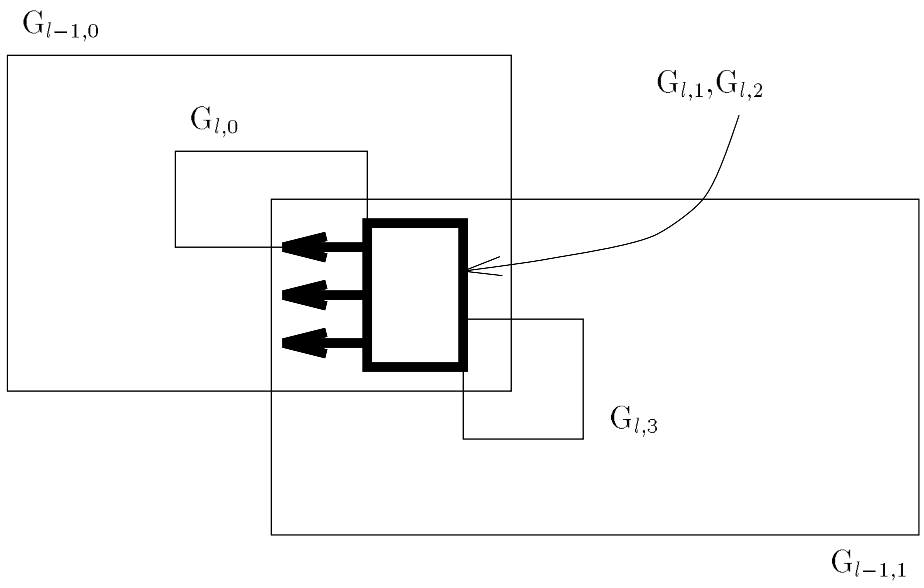


Figure 4.25: Inappropriate flux correction in an overlap region

4.5.3 Rotated Grids

Rotated grids, long a staple of Berger's strategy, were ultimately dispensed with. Berger notes that, although they covered the refinement regions more efficiently, in the sense that fewer nodes were unnecessarily refined, their efficiency contribution was small, only about 15% [Ber91]. In addition, flux correction for conservation at interfaces between coarse and fine grids, which is simple with fine grids that line up along coarse gridlines, is considerably complicated by rotated grids [Ber85]. Also of significance is the added computational overhead required: when grids line up, interpolation between coarser and finer grids is quite simple, and degenerates to simple copies between grids of the same resolution. With rotated grids, however, all interpolations involve point location, which on rectilinear grids is $O(\log n^{\frac{1}{3}})$ on the number of cells and on curvilinear grids is $O(\log^2 n)$ [PT92] (although some heuristics typically produce better results in many cases [Wil92], [Nee90]). Furthermore, every interpolation between any pair of grids must be based on positions in physical space, because distances between positions can be expressed only as floating point values (Figure 4.26). Thus, the computational overhead can be excessive, as can be the coding overhead.

Another problem that arises with rotated grids is that grids at the same level can overlap. In addition to the problems associated with non-rotated overlapping, each rotated overlap region has multiple, slightly different solution values for the same physical position, which renders the concept of a "solution" ill defined.

Perhaps most important, though, is that in order to use rotated grids, the solver must

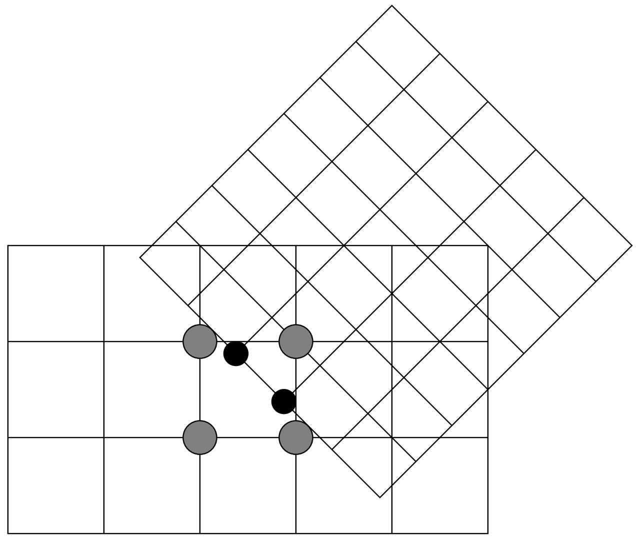


Figure 4.26: Interpolating between rotated grids

have mechanisms to determine the rotation and apply it to the numerical physics. This requirement can be unnecessarily burdensome to the application scientist.

4.5.4 Clustering

Berger originally used a modified minimal cost spanning tree algorithm for clustering, with the distance between nodes being the cost. The algorithm was as follows: first, the nodes that required refinement were split into clusters, based on proximity. Then grids were fitted around the clusters. Finally, the grids were tested to determine whether merging any pair of grids produced a reasonably efficient single grid, thus reducing the boundary space of the grids. This approach to clustering was not only highly heuristic, it was very time–consuming, because constructing a minimal cost spanning tree is $O(n^2)$ on the number of nodes requiring refinement, and merging is $O(g^2)$ on the number of grids. Over time, Berger pursued a number of different clustering strategies, but the one she ultimately settled on, based on signatures, has several advantages: it is fast, performing in $O(n \log n)$ time on the number of cells; it produces a good clustering in most cases and an acceptable clustering in all cases; it operates in computational rather than physical space; and it produces no overlaps.

4.5.5 Location of Variables

Berger's original system permitted variables to be located at the nodes only. However, with the introduction of flux correction to ensure conservation, and with the use of the new, signature–based clustering algorithm, it became necessary to locate the variables at cell centers. This decision proved useful in a number of ways, particularly because many of the phenomena to which Berger's AMR strategy was applied had cell–centered solvers. However, choosing cell centers over nodes, while perhaps more popular, is nonetheless still quite restrictive, because many multivariate applications require staggered grids.

4.6 Related Research Using Berger's AMR

There are several examples in the literature of experiments conducted using Berger's AMR strategy. These examples cover a wide variety of application areas, including computational fluid dynamics, meteorological simulations, materials science and general relativity.

Berger, with various collaborators, has produced more than a dozen publications on various aspects of her AMR strategy. Among the issues she has studied have been the AMR strategy itself [BO84], [BC89]; AMR data structures [Ber83], [Ber86]; conservation issues [Ber87]; the signature clustering algorithm [BR91]; implementation, distribution and load balancing issues on MPPs [BB87] and on SIMD architectures [BS94]; modeling embedded surfaces in adaptive hierarchies [AMB95]. She has also collaborated on studies of the application of her AMR stragegy to specific research topics, for example two-dimensional Euler equations for transonic flows [BJ85a], [BJ85b], and interaction of shocks and bubbles [BBSW94]. Generally, Berger's work has addressed applications in computational fluid dynamics.

Skamarock has used Berger's AMR strategy to study numerical weather simulation and prediction, beginning with his work in 1987 with Oliger and Street [SOS89]. Skamarock's implementation is based on Berger's original strategy [BO84], and thus he uses rotated, overlapping grids. Among the topics his research has examined are severe convective storms [SKW91], nonhydrostatic atmospheric flow in two and three dimensions [SK93], and long-lived squall lines [SWK94]. In addition, he has examined a variety of truncation error estimates in order to optimize the selection of regions to be refined in the class of problems that he studies [Ska89].

Some of Berger's collaborators, including Colella, Welcome and Bell, have published AMR studies separately from Berger as well, often collaborating with one another. For example, the three of them, with Pember and Crutchfield, have studied methods of embedding irregular regions within adaptive hierarchies [PBC+95]. Pember, Crutchfield, Bell and Colella joined Greenough and Beckner in studying interface-capturing in multifluid flows [GBP+95] using Berger's AMR. And Crutchfield, Bell and Colella joined Steinthorsson and Modiano in studying unsteady viscous compressible flows [SMC+95].

Another group of researchers using Berger's AMR strategy belong to the Binary Black Hole (BBH) Grand Challenge consortium. For example, Massó, Seidel and Walker describe a numerical relativity AMR implementation for evolving Schwarzschild spacetime in one dimension [MSW95]. In addition, Choptiuk has developed a Fortran 77 AMR implementation for studying numerical relativity in multiple dimensions [Cho94]. Also, Haupt describes a Fortran 90 implementation of the AMR strategy [Hau95] that may prove useful for developing new AMR implementations.

Another member of the BBH consortium, Parashar, has developed perhaps the most promising approach to distributing AMR grid hierarchies. Parashar's AMR implementation, called the *Distributed Adaptive Grid Hierarchy* (DAGH) system, achieves load balancing and minimizes communication by distributing the grid according to a self-similar space filling curve [PB]. In addition, Parashar's C++ implementation provides some object-oriented abstraction of many mesh and AMR concepts. Thus, implementing AMR versions of existing applications involves only a moderate amount of C++ coding.

A few other researchers are using Berger's AMR strategy. For example, Meakin discusses the relationship of Berger's strategy to overset grids, in which grids of different geometries overlap one another [Mea95]. Finally, Kohn and Baden have implemented AMR support using LPARX, a runtime parallel support system implemented as a C++ class library [KB95].

4.7 Popularity of Berger's AMR Strategy

Berger's AMR strategy shows incredible potential as a means of expanding the tractability of a wide variety of numerical experiments. For example, one of Berger's recent three-dimensional experiments exhibited an improvement over conventional techniques by a factor of 55 [BBSW94]. And yet, relatively few researchers are using this strategy, despite the fact that it has been available for over a decade. By contrast, dozens if not hundreds of researchers use AMR techniques on unstructured grids.

Most of those who use Berger's strategy are concentrated in a few small interlocking research groups. For example, Berger's work at RIACS led to collaborations with Pember, Bell, Crutchfield and Welcome of Lawrence Livermore National Laboratory and Saltzman of Los Alamos National Laboratory, and most of these researchers have also collaborated with Colella of the University of California, Berkeley. Skamarock, now at the National Center for Atmospheric Research, had Oliger as his dissertation advisor, as did Berger. Similarly, the Binary Black Hole Grand Challenge group has several researchers who have experimented with Berger's AMR, including Parashar, Haupt, Choptiuk and Chrisochoides, who has written a parallel AMR support library. Outside of these two groups are a few other researchers using Berger's AMR, including Kohn and Quinlan, but most of the scientists who use this strategy are connected either to the Berger-Oliger-Colella group or to the Binary

Black Hole group.

The primary reason for the lack of popularity of Berger's strategy is that it is extremely cumbersome to code and to maintain. Of the few researchers who have developed software for Berger's AMR, many have made a variety of application-specific simplifying assumptions in order to streamline the coding. For example, many AMR codes allow only one or two staggerings, typically either cell-centered, node-based or both. Also, some codes implement Richardson truncation error estimation by making significant adjustments to the solver — as in fact did Berger's original code — which is unrealistic for many legacy codes, and for other complicated codes. For example, Brandt describes a relativity solver he uses that has a loop body of approximately 500 lines, which he notes would be cumbersome to recode, although recoding the formal argument list and the loop bounds would be acceptable [Bra96].

Berger's AMR is difficult to code for two primary reasons. First, this strategy presents a very complicated data management problem, not only because of the dynamic nature of memory usage but also because of the relationships and interactions between the various parts. Second, the AMR algorithm itself is not only elaborate, but also very sensitive to even the smallest incongruities. At every step in the development process, new problems are discovered that must be addressed in a manner consistent with the rest of the AMR implementation. The temptation to treat Berger's AMR as a relatively minor modification of existing techniques — for example, traditional multigrid strategies — must be avoided, because Berger's strategy is significantly more complicated in both its data management and its algorithm.

To make Berger's AMR accessible to the community of researchers using structured finite difference schemes, a new approach must be developed. Requiring application scientists to code their own AMR schemes directly is unrealistic; rather, the computational science community requires an AMR system that manages its own data, includes all the appropriate algorithms and presents them in an intuitively clear manner, and imposes minimal reinstrumentation requirements on existing codes.

Chapter 5

Autonomous Data Management for Grid Hierarchies

Berger's adaptive mesh refinement strategy presents a significant data management challenge. AMR software architectures are unlike traditional numerical approaches, which need to manage only the solution vectors and perhaps some additional work space, typically for a small, fixed number of static grids whose relationships are known at compile time. Instead, AMR systems must manage not only the solution vectors for a large, dynamically changing collection of grids, but also the relationships between the grids, the relationships between the solution vectors and various other data items, and the relationships of the various data items to the methods that operate on them. In addition, maximal flexibility is achieved if the manner of describing the data items and relationships allows the data structure to be *autonomous*: that is, to manage itself, rather than relying on hard-coded management functions.

This aspect of Berger's AMR strategy has been one of the primary stumbling blocks to the increase of its popularity among computational scientists. The burdensome coding requirements of grid hierarchy data structures make development of general-purpose AMR systems in scientific languages like Fortran unrealistic. Even if a system is designed in a language that supports sophisticated data constructs, such as C, the enormous variety of structured simulation paradigms leads to a prohibitively high amount of design and implementation labor.

However, if an AMR system's data management infrastructure has at its foundation a solid body of theory, then design and implementation become not only less cumbersome but also more intuitive. Thus, a fundamental aspect of the research for this dissertation is such a collection of principles, which describe the nature and properties of the data structures and their management. Over the course of the development of the software architecture that demonstrates the research contribution of this dissertation, the underlying theoretical basis has influenced, and been influenced by, practical design requirements. Therefore, a description of the data management principles which give rise to generality and autonomy is in order.

Autonomous data management consists of several aspects:

- a data structure that encapsulates data items that apply to a particular stratum of the grid hierarchy;

- attributes that describe each data item and its relationships to other data items;
- a specification that serves as a lookup table for queries about the data items and their attributes;
- a declaration, supplied by the user, that describes the application, its data, and their relationships;
- a set of modules, which encapsulate the data items associated with various operations and properties;
- a software infrastructure for data management.

From these aspects, a relatively straightforward theoretical framework has been developed.

5.1 A Data Structure for Grid Hierarchies

Because Berger's AMR strategy is a hierarchy of levels of grids, an appropriate starting point for this discussion is to examine such a data structure (Figure 5.1). In this arrangement, an operation on the entire hierarchy — for example, a control algorithm — has access to each level, and an operation on a level — for example, integration — has access to each grid of the appropriate resolution. Encapsulating each of these *strata* — hierarchy, level and grid — in individual data structures, with more broadly applicable strata encapsulating more narrow ones, simplifies both data management and algorithms, since the contents of any of the strata are accessible via a single argument.

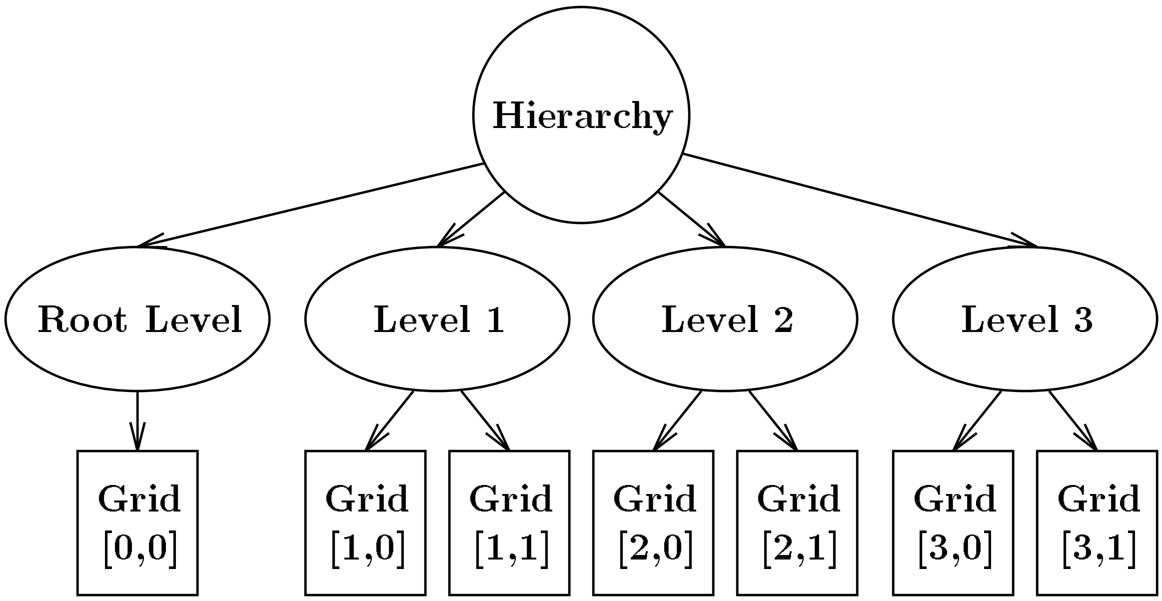


Figure 5.1: Basic grid hierarchy data structure

However, managing this arrangement poses a significant problem. For example, consider the physical domain of a simulation, represented as a pair of diagonally opposite endpoints $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. These endpoints apply to every level; that is, every level *inherits* the endpoints from the hierarchy. Yet, this data structure would have to store the endpoints redundantly with every level, because of the lack of a direct mechanism for the levels to access information on the hierarchy. Alternatively, the data structure could have pointers not only from hierarchy to level, but also from level to hierarchy:

```
struct hierarchy { int rank; struct level *l; }
struct level { float time; struct hierarchy *h; }
struct grid { float *solution; struct level *l; }
```

However, this arrangement is not ideal, because it tends to obscure the top down nature of the data structure.

Regarding the issue of redundancy, in this context it is not a memory consumption issue; rather, it is an issue of data *consistency* throughout the grid hierarchy. In the physical domain example, the endpoints perhaps do not change over the duration of the simulation. Instead, consider the physical time, which can be different at each level; for example, if level l is at timestep i_t, and level $l+1$ is at timestep $ri_t + r/2$, then its time is halfway between the times of the old solution and the new solution at level l. However, the time value is the same for each grid on a particular level, and it changes with each integration at each level. Thus, updating the time value after an integration is considerably simplified, and is intrinsically self-consistent, if the time value is associated with the level and is accessible by each grid, rather than being stored redundantly on each grid.

Therefore, instead of using the arrangement presented in Figure 5.1, consider a new arrangement that offers a *mirror image* data structure, shown in Figure 5.2. In this new arrangement, data that applies to the entire hierarchy can be declared in the structure labeled "Hierarchy Data," data that applies to an entire level can be declared in the structure labeled "Level Data," and data that applies only to a particular grid can be declared in the structure labeled "Grid Data." (In some sense, the "Grid Data" structure is superfluous, since the only structure that can access it is the associated grid. However, this arrangement simplifies both the underlying formalisms and the coded constructs, and the cost of implementation is one pointer per grid.) Thus, the structure declaration that is presented above can be replaced by:

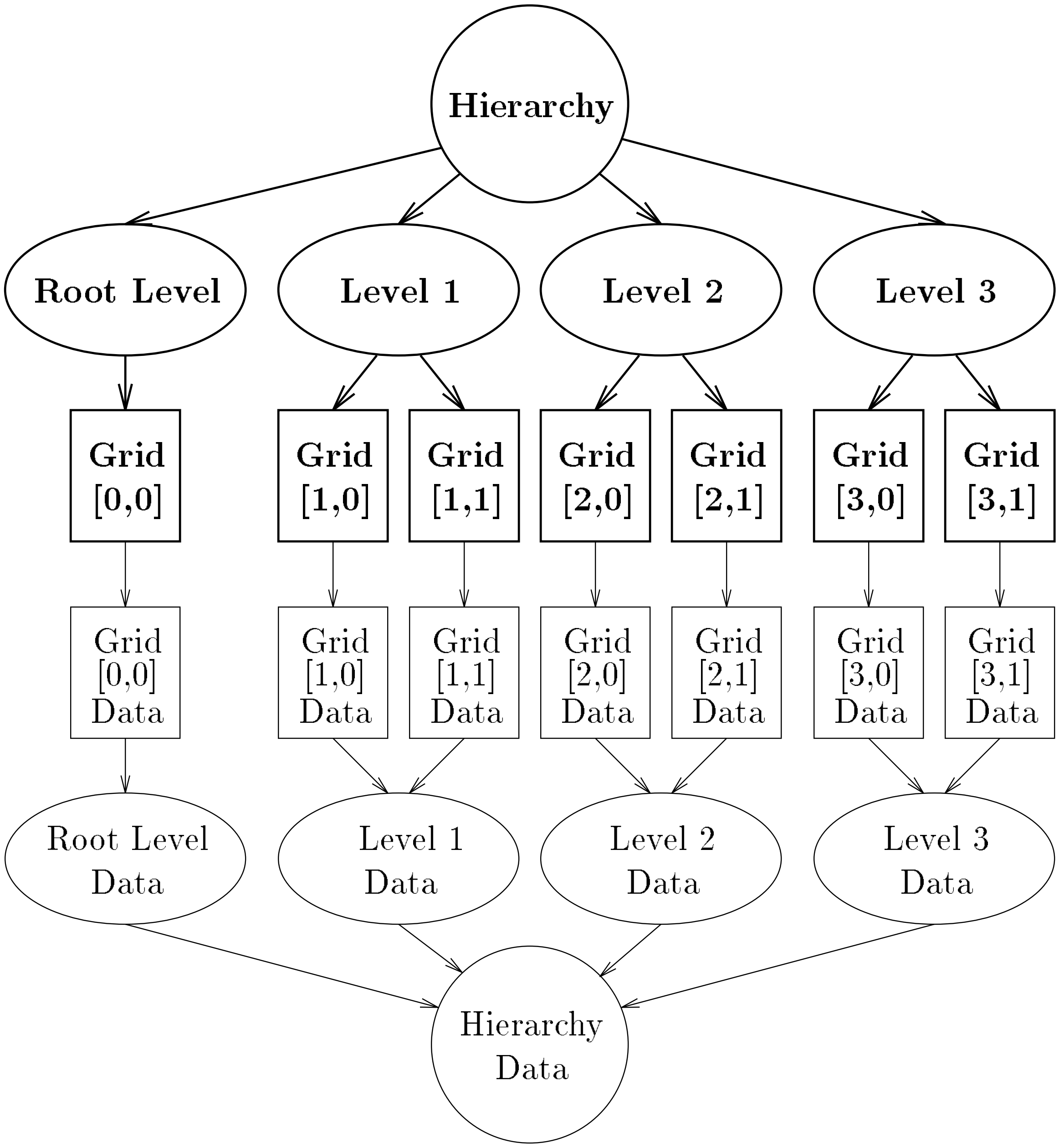


Figure 5.2: Mirror image data structure

```
struct hierarchy_data { int rank; ... }
struct level_data { float time; ... struct hierarchy_data *hd; }
struct grid_data { float *solution; ... struct level_data *ld; }
struct grid { struct grid_data *gd; }
struct level { int grids; struct grid **g; }
struct hierarchy { int levels; struct level **l; }
```

This arrangement of the strata and their data clarifies the location of various data items in the data structure. For example, the region of physical space delimiting the domain can be contained in the hierarchy data, the time value in the level data and the solution vectors in the grid data. More formally,

- a data item that applies to all levels should be contained in the hierarchy data;
- a data item that applies to all grids at a particular level but that can vary from level to level should be contained in the level data;
- a data item that applies exclusively to a specific grid should be contained in the grid data.

Actually, the depiction of the data structure in Figure 5.2 is incomplete in two ways. The first missing aspect is that, while each grid can access all of the relevant data, a level that as yet has no grids cannot access the associated level data, nor can a hierarchy access the hierarchy data if it as yet has no levels. Therefore, it is necessary for the hierarchy to have a direct reference to the hierarchy data, and for each level to have a direct reference to the corresponding level data (Figure 5.3):

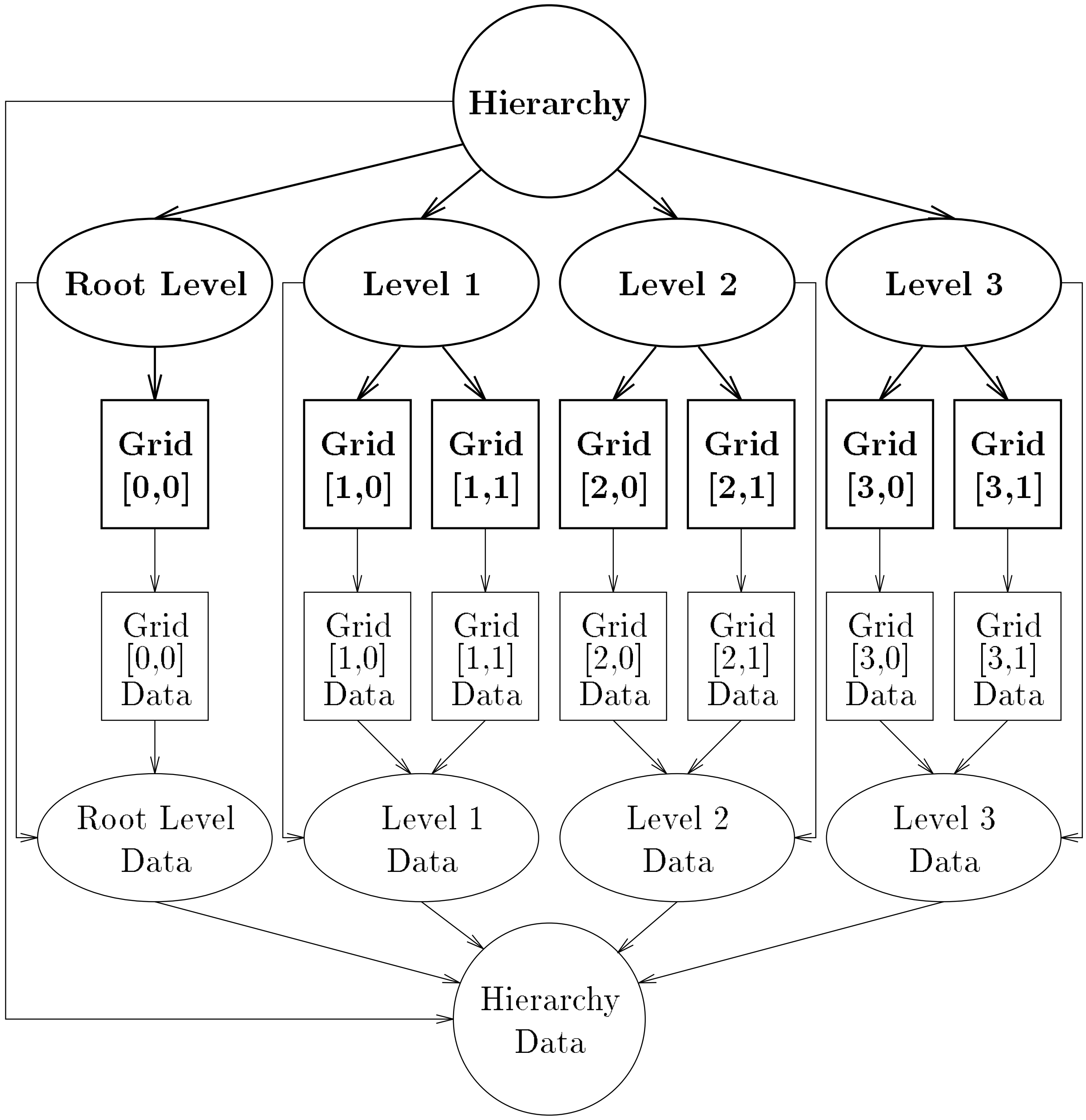


Figure 5.3: Mirror image data structure with additional pointers

```
struct hierarchy_data { int rank; ... }
struct level_data { float time; ... struct hierarchy_data *hd; }
struct grid_data { float *solution; ... struct level_data *ld; }
struct grid { struct grid_data *gd; }
struct level {
  int grids; struct grid **g; struct level_data *ld; }
struct hierarchy {
  int levels; struct level **l; struct hierarchy_data *hd; }
```

Second, in the current depiction, no distinction is made between data items whose values are determined on the fly, in a manner analogous to variables, and data items whose values are fixed at compile time, in a manner analogous to declared constants. An additional stratum, called the *fixed* stratum, encapsulates those data items whose values are constant (Figure 5.4). One disadvantage of this approach is that it disrupts the mirror image nature of the data structure, since the natural point of entry for operations on the entire data structure is the hierarchy, rather than the fixed structure. However, this disadvantage is outweighed by the abstraction provided in decoupling constant values from dynamically varying values, a common feature of programming languages.

5.1.1 Scope and Extent of Data Items

Two important concepts governing this data structure are borrowed from the literature on programming languages: scope and extent.

In this context, the *scope* of a data item is the portion of the data structure to which it is applicable, and by which it can therefore be accessed. Specifically, fixed data items apply to everything, data items on the hierarchy apply to all levels and to all grids on all levels, data

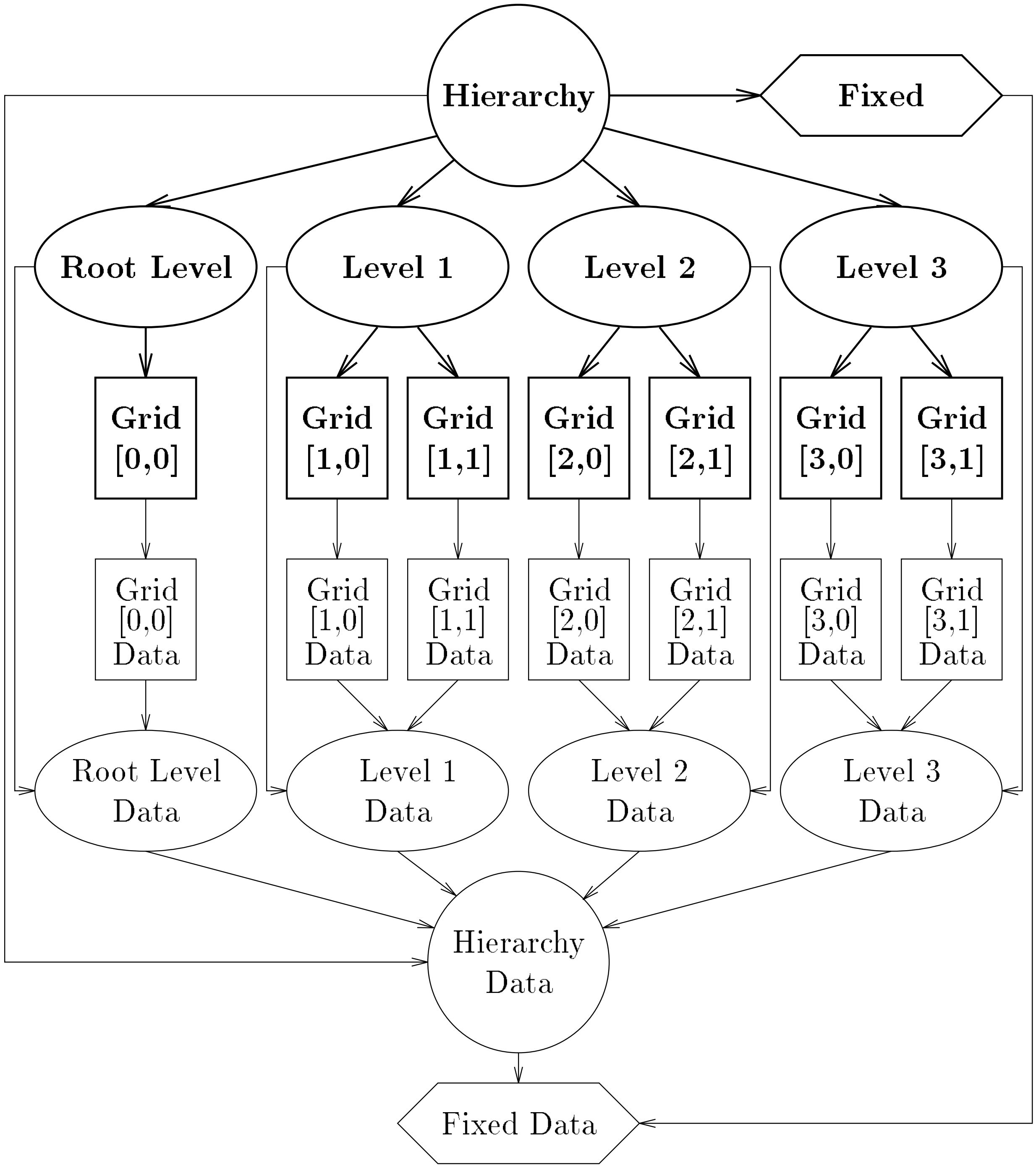


Figure 5.4: Data structure with fixed stratum

items on a level apply to all grids on that level, and data items on a grid apply exclusively to that grid. Thus, fixed data items have the *broadest* scope of the four strata, while grid data items have the *narrowest* scope. Scoping in this context is *static*: fixed data items are frozen, but similarly other data items do not migrate between the hierarchy, the level and the grid, and therefore scope does not change.

The *extent* of a data item is the period of runtime during which it exists and is accessible. For example, the extent of a hierarchy scalar is the entire existence of the hierarchy (though conceivably its value might be undefined at times), whereas the extent of a work vector used by a solver is (ideally) the duration of the call to the solver itself; that is, the work vector should be allocated immediately before calling the solver, and deallocated immediately after returning from the solver.

Extents fall into four categories: permanent, enduring, automatic and temporary. A *permanent* data item is one that is created immediately when the structure on which it is located is created, and is retained throughout the existence of the encapsulating structure. For example, a solution vector is required not only throughout all integrations, but also when the encapsulating grid is first allocated, because it is the destination of values from the old grids it replaces, and when the grid is being replaced, because it is the source of values on the new grids that are replacing it. An *enduring* data item is one whose extent is the computational lifetime of the data structure that encapsulates it, but that is deallocated before the encapsulating structure is replaced. Specifically, the values of an enduring data item are not retained from instance to instance of the encapsulating data structure, nor are they required for the process of retaining the values of other data items between instances. For example, the flux correction vectors are used during all of the integrations, but are not needed for regridding, because their values do not need to be transferred to the new grids and do not assist in transferring other values. (In the static hierarchy and single grid cases, and in the case of structured of fixed, hierarchy and level data, which are not replaced during a run, the distinction between permanent and enduring is meaningless.) An *automatic* data item is one that is automatically allocated immediately before a particular operation and deallocated immediately after, as in the work vector example above. (This definition conforms to the standard definition in the context of programming languages [KR88].) Finally, a *temporary* data item is one that must be explicitly allocated and deallocated by the operation that uses it.

5.1.2 Types

Each data item is of a specific type; however, these types can be fairly elaborate. Thus, distinguishing between the type of the data item's elements and the overall structure of the data item is crucial. An *element type* is a primitive data type, such as a boolean, integer, floating point, and so on, or a nearly primitive data type, such as a string, a point in physical or computational space, a region defined by a pair of diagonally opposite endpoints, and so on. A *parameter type* is the gross structure of a data type. Examples include scalars, contiguous lists, arrays, linked structures, discretized fields, and so on. Thus, the overall *data type* of a data item is the combination of its element type and its parameter type.

More specific than its data type is a data item's *stratiform type*, which comprises not only its element and parameter types but also the stratum that encapsulates it. For example, the physical time values of a solution are not merely a list of floating point values — one for each time level of the difference scheme — but more specifically a list of floating point values on a level.

A related issue is the distinction between a data item and an *instance* of the data item. For example, the list of physical time values is a data item that is encapsulated in the level data structure, but the list of physical time values for a specific level is an instance of the data item. And in the case of a dynamically evolving grid hierarchy, a particular solution vector encapsulated in the grid data structure is the data item, but the instance of the data item is that solution vector on a particular grid, which may well be replaced as the grid hierarchy evolves.

A *method*[1] is a procedure that applies to a particular structure or data item. In the data management system under discussion, methods are treated in essentially the same manner as are data items; specifically, methods are another parameter type. The advantage of this approach is that it simplifies both the conceptualization and the implementation of the system.

By definition, all methods are implicitly permanent, although the actual procedures to which they refer — that is, their values — may change during a run. Also, the element type

[1]This term is borrowed from the object oriented language literature; its use here is not meant to imply a strictly object oriented implementation.

associated with a method is the scalar type that the method returns, or a void for a method that does not return a value.

5.2 Attributes

Most data items have *attributes*, which describe aspects that can vary from instance to instance of their type. For example, a list of floating point values has a length attribute associated with it, a solution vector has a staggering, and so on.

Data management is simplified if each piece of data has its attributes conveniently available. There are two means of achieving this goal: either the attributes can be explicitly included by encapsulating them with the data item, or they can be implicit, either by being hard-coded — for example, as macros or declared constants — or by pointer references. This last approach has the benefit of generality; that is, the fewer pieces of information that are hard-coded, the less recompiling is necessary in order to change them, and therefore the more quickly and conveniently they can be altered.

The list example appears fairly trivial. But consider the case in which there are two arrays of the same size, where the length of the rows of the arrays can vary (Figure 5.5). In this case, changing the length of the arrays is significantly more complicated if it must be explicitly changed for both instances, whereas changing just the one instance and giving both arrays access to the length list simplifies encapsulation. Thus, new instances of these arrays can now be created simply by calling a constructor with the type and identifier of

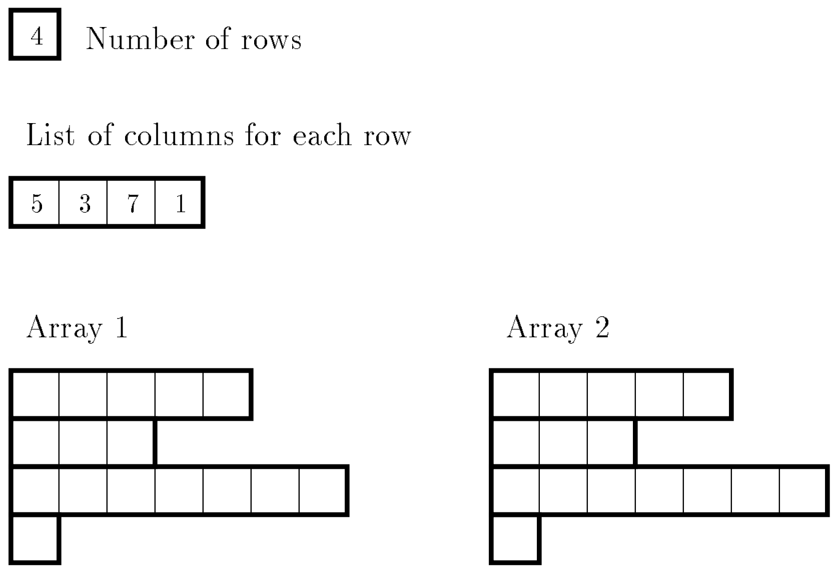


Figure 5.5: Arrays and their shared attributes

each; the constructor can then access the attributes of the data item to be created, with complete confidence that these values will be the same as for other instances of this item.

5.2.1 Attribute Categories

Grid hierarchy attributes come in two pairs of complementary categories: structural and functional, and referential and reciprocal.

Structural attributes describe the computational shape of a data item. In the above example, an array whose rows may have varying lengths has two structural attributes: the number of rows, and the list of the number of columns in each row. A solution vector, on the other hand, has many structural attributes, including rank, staggering, stencil, number of time levels, and number of loci along each axis. In fact, this last attribute will depend on other attributes for its value; for example, the solution vector for a node-centered variable will have one more node along each axis than the number of cells in the grid, and will also have extra nodes for the ghost boundary cells specified by its stencil. Informally, it may be helpful to consider structural attributes to be those attributes whose values must be known for the data item to be allocated. (In practice, there are structural attributes that do not exhibit this property, but it may be helpful to think of them this way for the moment.)

Functional attributes describe the non-structural roles that various other data items play for the attributing data item. Both data and methods can be functional attributes; that is, there are *functional data* attributes and *functional method* attributes. For example, a solution vector may have associated with it another data item in which its flux values are stored, in anticipation of the flux correction step. Similarly, it may have also associated with it an initialization method, so that when it is created it can immediately be set to the proper value.

Referential attributes are those that connect an instance of a data item to instances of other data items with which it has a specific relationship. The advantage that referential attributes provide is that they associate data items in a manner that promotes generality. In the correction vector example, the reference from the solution vector to the correction vector makes possible a general-purpose correction routine, because the identity of the solution vector that is to be corrected implicitly identifies the appropriate correction vector as well.

Reciprocal attributes are returns of referential attributes. That is, if data item i refers to data item j for attribute a, then data item j reciprocates to data item i for attribute $\bar{a}$.

Every referential attribute has a reciprocal, and thus each referent knows which data items refer to it, and for which attributes.

A crucial distinction between reciprocal and referential attributes is that reciprocal attributes do not subsume references; that is, no memory address is associated with a reciprocal attribute. Instead, reciprocal attributes are encoded information that indicate to a data item what role it plays with respect to other data items. For example, if a solution vector refers to a stencil item as its stencil, then the stencil reciprocates in its knowledge that instances of the solution vector refer to it for stencil information. Thus, the information about the relationship between the two data items is bidirectional, but the information about the relationship between *instances* of the two data items is unidirectional.

Reciprocal attributes provide an important advantage in promoting autonomy, because they allow decisions to be predicated on maximal information with minimal research. Consider the time information about a solution vector. There are many facets to time information, including the number of time levels, the time level relative to the time interval — for example, a centered time difference scheme has three time levels whose relative values are -1, 0 and 1, corresponding to u^{n-1}, u^n and u^{n+1} — the physical time for each time level, the timestep index for each time level — that is, the iteration number at that level — and so on, as well as indices indicating which time level is the "old" timestep and which is the "new" timestep. Rather than requiring a separate data item — and a separate attribute — for each of these categories of time information, a simpler solution is to require only one of them to be explicitly declared as a data item, and to attach the others as attributes. However, because the number of time levels can vary from difference scheme to difference scheme — for example, forward time differencing would have relative time levels 0 and 1 — the other time attributes must be explicitly allocated, according to the number of time levels (and thus elements) of the relative time level list. Thus, each relative time level list has associated with it (as attributes) several other lists, some of them floating point (for example, physical time) and some of them fixed point (for example, timestep index), of the same length as the relative time level list. However, allocating such lists for all lists of the appropriate type would be extremely unwise, since another list might instead be, for example, a data item for a set of particles, and might therefore be thousands or even millions of elements long, and so the memory waste would be outrageous. Therefore, an important prerequisite, before allocating such attribute lists, is the determination that they are necessary, in the sense that the data item is actually used as time information by some other data item. A reciprocal attribute provides precisely that information. That is, if the set of data items that consider the list to be time information is nonempty, then the list is time information by definition, and therefore its time attribute lists should be allocated; conversely, if the list's time information reciprocal attribute set is empty, then by definition it is not time information, and therefore time attribute lists should not be allocated, since they would be completely wasted.

A few final points about the two pairs of attribute categories will clarify these concepts. First, some structural attributes are also referential. In the array example, the list of the number of columns in each row can itself be a data item; thus, its structural role is implemented by its reference. In principle, the list can have its own intrinsic meaning that is separate from its role in the computational shape of the array. In contrast, *all* functional attributes are referential. Also, no attribute is simultaneously structural and functional.

5.2.2 Rules for Referential Attributes

Referential attributes are subject to a few simple rules. The first rule addresses the issue of scoping.

> *Rule of Monotonic Scoping: a data item can reference other data items in strata of equivalent or broader scope only.*

Thus, a fixed data item can reference only other fixed data items, a hierarchy data item can reference fixed and hierarchy data items, but it cannot reference level or grid data items; a level data item can reference fixed and hierarchy data items as well as level data items; a grid data item can reference any data items. (This principle is illustrated in Figure 5.4, wherein all pointers are directed downward. Some pointers, such as pointers between siblings, are lateral, but no pointers are directed upward.) This rule recognizes an important practical aspect of the data structure: that the fixed data must be allocated before the hierarchy data, the hierarchy data before the level data, and the level data before the grid data, in order to give succeeding allocations something to reference. Here a clarification is in order: the scope of a data item includes other structures of the same stratum; for example, if the scope of a data item is the grid, then it can reference data items on other grids, although most references will be local to the grid that contains the data item. More importantly, a data item whose scope includes the level can reference data items on other levels as well as on its own, a situation that is more common than the grid case.

By contrast, there is no analogous rule about the extents of referential attributes. Thus, for example, a permanent data item like a solution vector can reference an enduring data item like a correction vector, and a temporary array can reference a permanent list as its row length vector.

The second rule governing referential attributes is a rule applying to those attributes that are simultaneously referential and structural.

> *Rule of Monotonic Complexity: an attribute that is both referential and structural can refer only to a data item whose data type is less complex than that of the referring data item.*

In the array example, a scalar is less complex than a list, and therefore the list of row lengths can refer to a scalar for its length; similarly a list is less complex than an array, and so the array can refer to the list as its row length vector, and by extension to the scalar for the length of its row length vector.

With data types defined over a space, for example solution vectors and correction interface vectors, the notion of complexity is less obvious. To clarify, the complexity of a spatial data type is defined as the minimum overall size it can take on relative to the region of space on which it is defined. Thus, a solution vector covers the entire grid on which it is defined, while a correction interface vector covers only the surface of the grid — for example, in

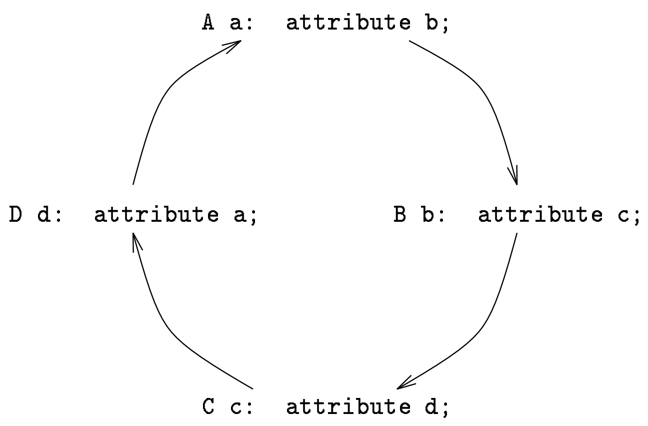


Figure 5.6: Cycle of structural attribute references

three dimensions an interface comprises the six faces of the rectangular prism on which the grid is defined, and each face of the interface vector has a thickness of a single locus. In addition, a spatiotemporal type is more complex than the corresponding time-independent type, because the spatiotemporal type covers multiple time levels, while the spatial type covers only one; that is, the spatiotemporal type covers more of spacetime than the time-independent type. Also, a spatial parameter type is by definition more complex than a structured type; therefore, for example, a solution vector can reference a stencil as part of determining its size and shape.

The primary motivation for the latter rule is purely practical: the order of data item creation must be fixed and well-defined, because otherwise *reference cycles* can form (Figure 5.6). Reference cycles are unresolvable; that is, no data item in the cycle can be created until its structural attributes have been resolved, and its structural attributes cannot be resolved if their referents, or the data items their referents depend on, cannot be created.

5.2.3 Attribute Appendices

Each data item has associated with it an attribute *appendix*, which is an attached list of some of the attributes of the data item. The choice of which attributes to store explicitly in the appendix, and which to access implicitly or to derive from other attributes, is purely an implementation decision, and is driven by the need to balance memory consumption against processing time and coding convenience. Any attribute that is explicitly stored in the appendix consumes additional memory space; if the application is memory-bound, or if the fixed size of certain data structures exceeds the anticipated sizes of the grids, then it may be more appropriate to put fewer attributes in the appendix, and to decode the rest on the fly. On the other hand, any attribute query or derivation requires additional processing time, which may be inappropriate if the application is CPU-bound. However, the amount of time associated with obtaining attribute values during the run typically is low compared to more significant operations, so the waste is likely to be miniscule.

Another concern in making these decisions is the construction of *wrappers* around existing, legacy-style code fragments. The more attributes that are precomputed and stored in appendices, the less extra instrumentation is required within a wrapper. Thus, a data management system that is motivated by the desire to minimize the amount of additional coding required by the application scientist may be better served by a larger, more complete attribute appendix. On the other hand, this concern could be minimized by providing a means of automatically generating the wrappers.

5.3 The Specification

The last missing piece to the grid hierarchy data structure is the *specification* of the attributes of the various data items (Figure 5.7). This structure contains all the information about the various attributes as they apply to each data item, which can then be imposed on each instance of each data item. For example, the specification has entries asserting which list describes the rows of a particular array, which stencil applies to a particular solution vector, and so on.

In essence, the specification acts like an enormous, five-dimensional lookup table, with the five dimensions being the stratum, the element type, the parameter type, the parameter identifier of the given stratiform type, and the attribute. (Obviously, it would be unwise to implement the specification this way, because expanding the specification by even a single new attribute could potentially add thousands of new table entries.) The lookup table is sparse, because

- not every combination of stratum/element type/parameter type is a valid stratiform type,
- not every valid stratiform type will be used by an application, and
- each valid stratiform type requires only a subset of the set of possible attributes.

For example, a real solution vector on a grid may require rank, staggering, stencil and so on, but will not require a length vector.

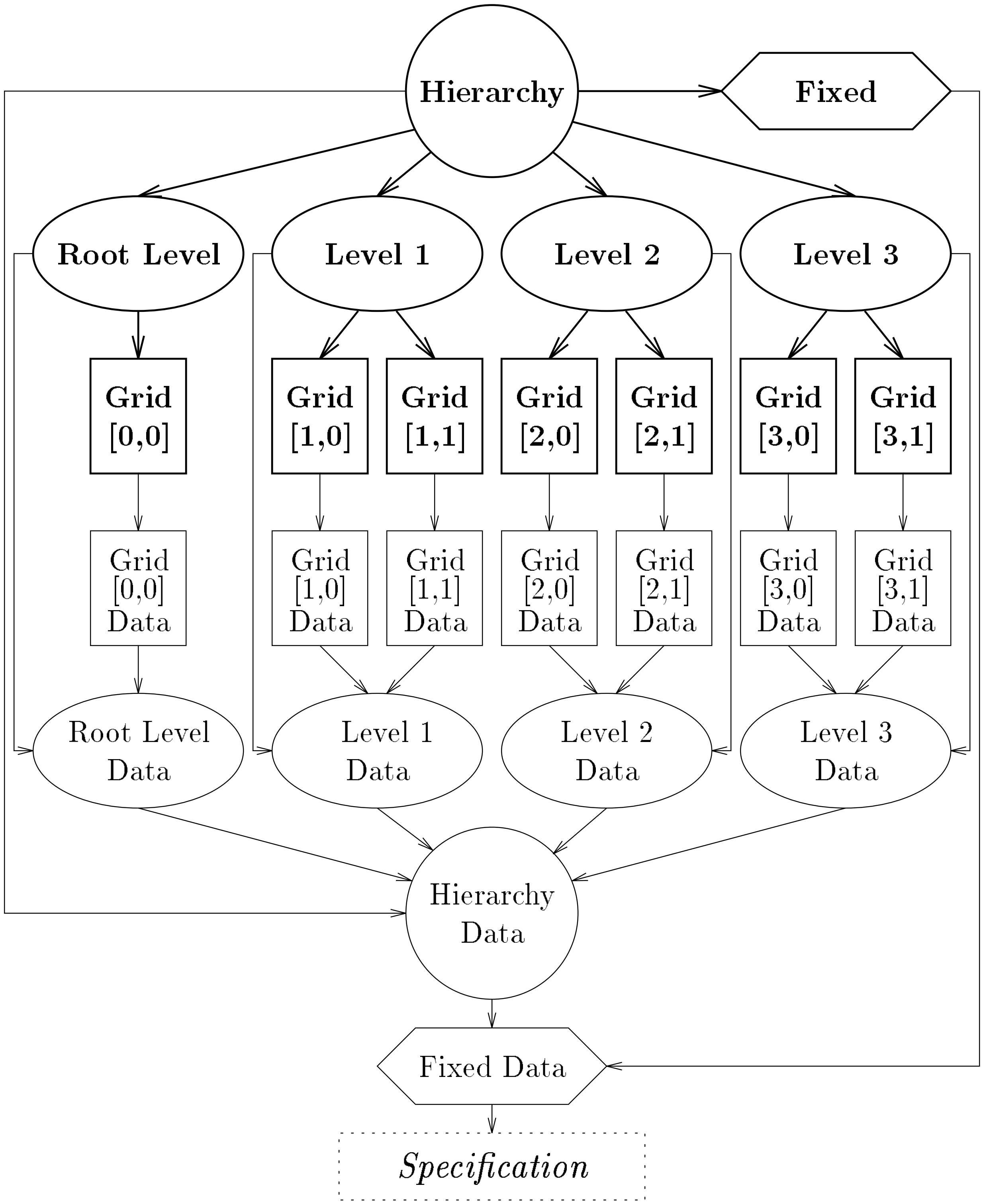


Figure 5.7: Grid hierarchy data structure with specification

For each attribute that cannot be derived trivially from other attributes, the specification contains either the value or an encoded *generic* reference; specifically, it contains an encoded description of the stratum, element type, parameter type and parameter identifier, which the data management system can translate into an actual memory address for each instance of the data item. The data management system can query the specification at any time, in order to create, manipulate or delete a data item or structure, a capability that provides significant flexibility not only to data management activities but also to the AMR algorithms.

For purposes of scoping, the specification can be considered simply as another stratum of the system; in fact, it has the broadest scope, being accessible by all other strata but having access to none of them. Naturally, its extent is permanent; in fact, it must be created first and deleted last, because all of the data items on all of the other structures depend on it for determining their attributes. In fact, by encapsulating the description of the data and methods, the autonomic nature of the data management is decoupled from the compiler, so that different collections of data for different applications need not require complete recompilation of the data management software.

However, it is not appropriate to consider queries to the specification as constituting references, because the queries do not require knowledge of particular memory addresses on the specification in order to obtain appropriate results. Thus, while some attributes are both structural and referential, for example the row length vector of an array, there are also some attributes that are exclusively structural, for example the rank of a solution vector. Those that are exclusively referential — for example, the correction vector associated with a solution vector — are, of course, functional attributes.

5.4 The Declaration

If the specification is a description of the data items and their attributes in a form that the data management system can understand, then the *declaration* is the same description in a form that the user can understand. Specifically, to create a new application for the data management system, the user must *declare* the application's data and methods, and the relationships between them.

Thus, the data management system requires a declaration language, and a parser to convert the declaration into an encoding that is appropriate for a specification. The syntax of the declaration language is arbitrary; that is, any number of approaches will produce valid declarations. But the kinds of information that the declaration language must be able to express is more well defined.

Obviously, the first thing that the declaration language needs is a way to declare a data item. In C- or Pascal-like syntax, this might look like:

```
RealList time_level;
```

However, this declaration is insufficient, because it doesn't indicate

- the stratum on which the data item is located;
- the extent;

- the structural attributes.

Therefore, a more accurate depiction would look like:

```
RealList time_level:
  Level, Permanent,
  Elements number_of_time_levels;
Integer number_of_time_levels:
  Hierarchy.
```

A data item requires one more piece of information before it can be considered useful: an initial value. Thus:

```
RealList time_level:
  Level, Permanent,
  Elements number_of_time_levels,
  Value 0 1;
Integer number_of_time_levels:
  Hierarchy,
  Value 2.
```

So in this case, `number_of_time_levels` is an integer scalar whose value is 2, and `time_level` is a list of real (floating point) values whose length is expressed by `number_of_time_levels` and whose values are 0 and 1; for example, `time_level` might refer to a forward time finite difference scheme of the form $\mathrm{u}^{n+1} = \mathbf{Q}(\mathrm{u}^n)$.

In addition to structural attributes, a declaration may need to include functional attributes as well. For example, in flux correction a solution vector refers to the appropriate correction vector:

```
SolutionVector density_vector:
  Grid, Permanent, ... ,
  Correction density_correction_vector;
Surface density_correction_vector:
  Grid, Enduring, ....
```

In this example, the relationship between the two vectors is explicitly declared via the keyword `Correction`.

Similarly, a data item may have associated with it functional methods:

```
SolutionVector density_vector:
  ...
  Initialize shock_tube_initialization_method;
VoidMethod shock_tube_initialization_method:
  Hierarchy,
  Value Grid_shock_tube_initialization;
Void Grid_shock_tube_initialization(): Grid.
```

In this example, the method `shock_tube_initialization_method` has as its value a pointer to the function `Grid_shock_tube_initialization`, and is referred to by `density_vector` for the purpose of initialization. Thus, at runtime, when `density_vector` is to be initialized, it accesses `shock_tube_initialization_method` and executes the function that its contents point to, `Grid_shock_tube_initialization` (Figure 5.8).

In the tradeoff between memory consumption and computation time, the full decoding process adds nothing to the memory cost of the attribute, but takes more time to interpret; decoding all attribute references when a data item is created and storing the memory addresses in the attribute appendix saves time, but costs the space of the extra pointers and values.

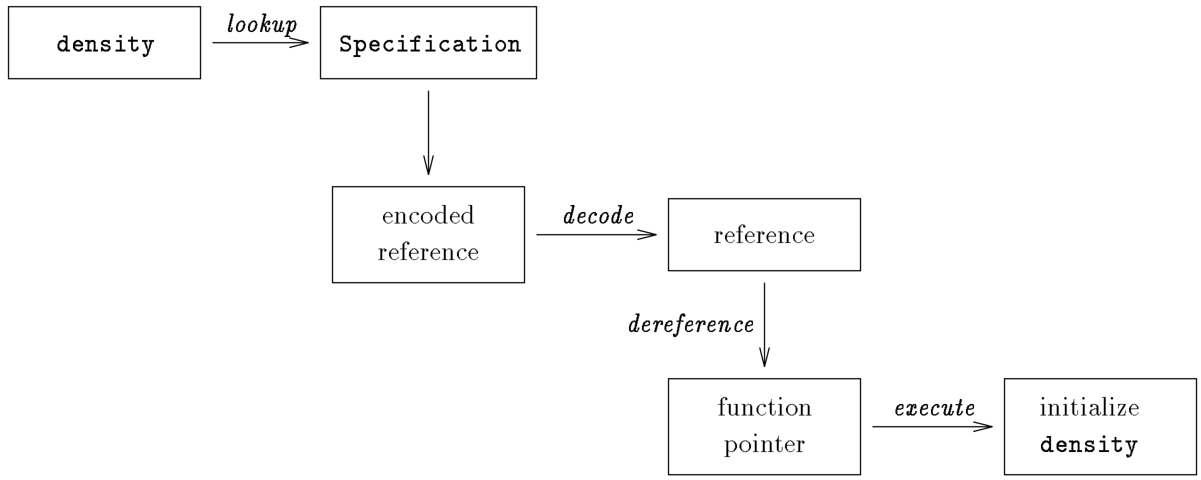


Figure 5.8: Initialization via attribute lookup

5.5 Modules

A *module* is a logically affiliated collection of data items and a (possibly empty) operation with which they are associated. For example, the data appropriate for a solver — the stencil, the time levels, the temporary work vectors and so on — have a logical relationship to the solver and perhaps to each other. The data items of a module can be located on multiple strata of the grid hierarchy, rather than being confined to the stratum to which the operation applies.

To illustrate, consider an example solver (Figure 5.9). Here, the stencil and time levels for the solver are implicitly attached to the solver itself, as is the work vector. Furthermore, the work vector is declared `Automatic`. In this context, the meaning of an automatic extent is more clear: an automatic data item is created immediately before calling the operation of the item's module, and it is deleted immediately after returning from

```
Solver Grid_piecewise_parabolic_method();

Stencil ppm_stencil:
  Hierarchy;
List ppm_time_level:
  Level;
Integer ppm_time_levels:
  Hierarchy;
SolutionVector ppm_workspace:
  Grid, Automatic.
```

Figure 5.9: Example solver module

```
Void Grid_generic_solve(grid) {
Grid_automatic_allocate(
  grid, PIECEWISE_PARABOLIC_METHOD_MODULE);
Grid_piecewise_parabolic_method(grid);
Grid_automatic_deallocate(
  grid, PIECEWISE_PARABOLIC_METHOD_MODULE); }
```

Figure 5.10: Stub for allocating and deallocating automatic data items

that operation. Thus, for a call to `Grid_piecewise_parabolic_method` to advance a particular grid, `ppm_workspace` is allocated immediately before the actual call, and as soon as `Grid_piecewise_parabolic_method` terminates, `ppm_workspace` is deallocated (Figure 5.10).

Modules contribute two important data management capabilities, activation and variation.

Activation refers to the ability of a module to be *active* or *inactive*; that is, the module can be declared "on" or "off," and in the latter case none of its data items will be created (except, of course, scalars and methods, which are not allocated as such). This capability is helpful in the case of large, module-specific data items that are encapsulated in modules that are not currently in use. For example, a solver module might include a large, enduring work area; if the solver were not going to be used in a particular experiment, that storage space would be wasted. However, if the module is inactive, the work area is guaranteed never allocated.

Variation is the property that certain attributes change according to which of the declaration's modules are in use. For example, consider an application that has two solvers, one with a five point stencil and the other with a seven point stencil, and suppose that the application scientist wishes access to both solvers at runtime, perhaps deciding to change solvers at a particular level during a regridding. In this case, the notion of the stencil of a solution vector is not fixed: the solution vectors that are on the levels that use the lower order solver have fewer ghost boundary zones, compared to those on the levels that use the higher order solver. Thus, a *variform* attribute is one that can take on any of several different values depending on which of a set of modules is currently in use in the context most directly associated with the attribute's data item. For example, consider a variform referential attribute whose referent is in a module that applies to a level. The attribute refers to the instance of the referent data item on the same level as the attribute's data item, unless explicitly declared otherwise.

Variation and activation combine to produce additional utility, because an inactive module will not contribute its data items as potential attribute referents of a variform attribute.

Returning to the multiple solver example, if both the lower order and higher order solvers are active, then the data management system must assume the worst case when refining, that the levels finer than the level about to be regridded might employ the higher order solver, regardless of which solver is used by the level to be regridded, and that therefore the new grids must be expanded sufficiently to cover the maximal possible stencil. This effect can produce significant waste in some cases. For example, in a case of refinement by a factor of two, an $8 \times 8 \times 8$ grid with a buffer region of one cell in each direction must be expanded by two zones to cover a seven point stencil, but by only one zone to cover a five point stencil. The expanded grid in the former case, including its own buffer and ghost zones and covering finer ghost boundary zones, is $18 \times 18 \times 18$, for 5832 cells total, and in the latter case is $16 \times 16 \times 16$, for only 4096 cells total, so forcing all grids to assume that finer levels can use a seven point stencil imposes a significant waste, over 40% in this example. In fact, even in the case of a $64 \times 64 \times 64$ grid, the waste would be over 8%. Thus, combining activation and variation provides sufficient flexibility to reduce overhead and improve performance in many cases.

While some modules apply only in certain circumstances, others always apply; these are called *generic* modules, and typically they have no operation associated with them. For example, all clustering algorithms must create a list of cluster regions and have such clustering arguments as minimum efficiency and maximum cluster size. Therefore, a useful module is one that declares these data items, since then they need not be declared repeatedly if there are multiple clustering algorithms spread out among various clustering modules.

5.6 Data Management

The declaration and the data structure lead naturally to a very simple data management paradigm, which consists of three basic operations: creation, deletion and execution. All three of these operations are split into suboperations, but among them they constitute the entire set of tasks required of data management. These same operations are performed on individual data items, on the data structures associated with the strata, and on the grid hierarchy as a whole — though to some extent the last case is simply the result of the second case.

5.6.1 Management of Data Items

Creation and deletion operate at several levels of complexity. The simplest is the creation or deletion of a single instance of a single data item, for example a particular solution vector. To create such an instance, the data management system

- creates all attributes;
- allocates the instance of the data item according to its structural attributes;
- initializes the instance of the data item.

For example, consider the creation of an instance of a solution vector on a grid. First, its attributes — staggering, stencil, number of loci along each axis, number of time levels and so on — are initialized. Next, based on this information, the solution vector is allocated.

Finally, the solution vector is initialized, most likely by calling the method referred to by its initialization attribute.

Deletion of a data item operates similarly, although it is a bit less elaborate, because the attributes have already been initialized at creation. Deletion includes an operation similar to initialization, called *finalization*, which is executed immediately before the data item is deallocated, typically to transfer some or all of the data item's values to a more permanent location. For example, consider flux vectors. The solver for a flux-conservative difference scheme must produce flux values for the interfaces between a grid and its parents and children. It is generally not convenient to use the correction vectors directly for flux storage, since the complexity of the extra instrumentation inside the solver may be off-putting for the application scientist, and also because some of the flux correction values, namely those on the interface of the grid being advanced, are immediately added to the running sum for the interface of the grid. Therefore, the flux vectors may be automatic with respect to the solver, and before they are deleted, their values must be stored in the appropriate correction vectors. The finalization method automatically performs the appropriate data transfers, without having to code the transfers explicitly into either the solver, the wrapper around the solver, or the integrator; that is, the data transfer can be performed by a specialized subroutine which is declared to be the finalization routine for the flux vectors. Thus, the flux vectors can be automatic with respect to the solver, and so they need not be explicitly created and deleted, since the data management system will automatically create and delete them before and after calling the solver, respectively, and will automatically call the flux transfer method immediately before deletion, to finalize the flux vectors. In addition, this approach promotes generality and code reuse: attributes of the flux vectors implicitly identify the associated solution vector that the fluxes will correct, and the solution vector's correction data attribute identifies its correction vectors, so the flux vectors can easily trace the correction vectors into which their fluxes are to be transferred, without needing to know any application-specific information about any of these data items.

Finally, execution with respect to a data item specifically denotes execution of its functional method attributes, including initialization, injection, projection, input, output and so on. The interchangeability of these operations promotes flexibility and generality within the data management framework.

5.6.2 Management of Strata

The management of strata is as straightforward as the management of data items, and again is simply a matter of creation, deletion and execution. Creation of an instance of a stratum is a four-step process. First, the "stratum data" structure is allocated. Next, the framework for storing data items and their attribute appendices is allocated and cleared; for example, the framework for floating point solution vectors is a list of pointers to solution vectors, all of which are initially null. Third, the "stratum" structure is allocated, and appropriate pointers are set to link it to the "stratum data" structure. Lastly, the permanent data items on the stratum are created. As for deletion, the process is the same in reverse order, except that all data items are deleted before the stratum framework can be eliminated. Finally, execution on a stratum denotes the execution of a module method that applies to that stratum; for example, executing the solver on a grid.

5.6.3 Management of the Grid Hierarchy

The management of the grid hierarchy as a whole is a combination of the management of the various strata. To create a grid hierarchy, the strata are created in order from broadest to narrowest scope. Thus, the specification is created first, then the fixed structure, then the hierarchy, then all of the levels and finally one or more grids. In the general case, only the grids at the root level are created, either directly from a list of root level subdomains, or by deriving the domain of a single root level grid from the overall domain. (In fact, an ideal implementation checks for a list of root level subdomains, and if it exists, creates appropriate grids; if there is no such list, it creates the default, full domain grid.) Deletion is performed in the same manner as creation, but in reverse order. There is no execution of the grid hierarchy as a whole; rather, operations that apply to the entire grid hierarchy are executed on the hierarchy stratum.

5.7 Summary

The design and implementation of a general-purpose AMR system requires a solid theoretical basis, not only to formalize the constructs that constitute the system, but also to simplify the description and development of its components, not only the data structures but also the means by which they are managed. Thus, the conceptual constructs laid out in this chapter are both the result of and an aid to designing, using and understanding the implementation of Berger's AMR strategy that constitutes the balance of the research presented in this dissertation.

These concepts were not wholly derived prior to implementation. Rather, they arose naturally as the design evolved. Originally, the design required that the application scientist code aspects of the data structure by hand, in the language of the implementation. However, it rapidly became clear that such an approach had many disadvantages, and no obvious advantages beyond simplifying the implementation of the system — the opposite of the purpose of the system, which is to simplify the implementation of applications. Thus, the need for an autonomous, easily described, extremely flexible data structure and management framework became increasingly urgent. The theoretical framework, and the principles it embodies, were to some extent implicit in the design decisions, both affecting how the data management was implemented and being affected by the development decisions that necessity imposed. Thus, the theory underlying this design evolved progressively, in concert with the implementation.

Over the course of these investigations, the advantages of this autonomous approach to data management have become increasingly clear. The ability to declare an almost limitless variety of data, methods, and — most importantly — relationships between them, has provided a flexibility and an ease of application design that are otherwise unknown in the AMR literature. Thus, the ideas laid out in this chapter provide an excellent foundation for the implementation of Berger's AMR strategy that encompasses the practical benefit of this dissertation.

Chapter 6

HAMR: A Software Framework for Hierarchical Adaptive Mesh Refinement

The research for this dissertation includes an implementation of the concepts examined, which is called the *Hierarchical Adaptive Mesh Refinement* (HAMR) system. HAMR is a software architecture for building adaptive mesh refinement applications on grid hierarchies. It is general-purpose and flexible, combining the autonomous data management concepts described in Chapter 5 with Berger's AMR stragegy, described in Chapter 4. Unlike other implementations of AMR, HAMR does not require extensive programming in a high-level language, burdensome modifications to existing solvers, or detailed knowledge of the computational platform on which the simulation executes. Rather, HAMR largely decouples the simulation kernel from the AMR strategy, the data management, and a variety of implementation details.

HAMR consists of four components (Figure 6.1): a set of type definitions, a low-level function library, an autonomous data structure, and a set of commonly-used algorithms for implementing Berger's adaptive mesh refinement strategy. Each of these components includes a set of predefined standard entries, and also leaves room for application-specific entries that can be supplied by the user. In addition to these components, each application requires a set of algorithms that define and implement it.

HAMR is implemented in the C language[1] and easily allows incorporation of Fortran subroutines such as solvers, initializations and so on.

Underlying the design philosophy of HAMR is a single, clear goal: to simplify the process by which an application scientist can convert a traditional, nonadaptive simulation to an adaptive version that employs Berger's strategy. Each of the components of HAMR addresses this goal in its own way. The set of data types provides the flexibility that allows a wide variety of applications, which may have diverse data requirements, to be incorporated with minimal user coding. The function library provides the operational underpinnings from which sophisticated management and AMR algorithms can be constructed, while encapsulating those operations in a manner which requires little or no adjustment to address specific

[1]A C++ implementation would also have been appropriate; the advantage of C is that it comes bundled with virtually every platform, which simplifies portability. Some C++ implementations of Berger's AMR have experienced difficulties because appropriate C++ compilers are unavailable on some platforms [Bal96] or because of insufficient optimization power in existing C++ compilers [Bry96b].

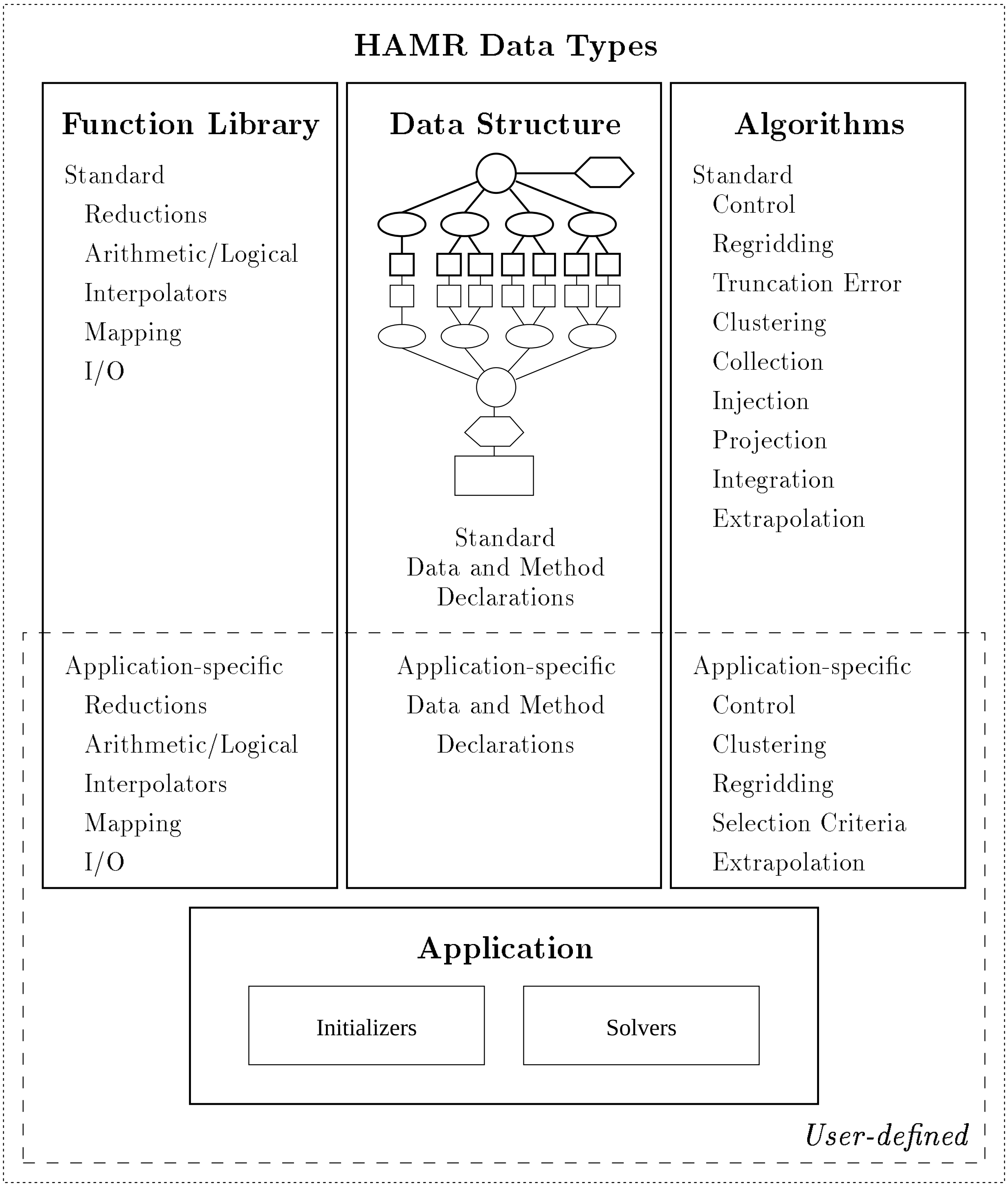


Figure 6.1: Components of the HAMR architecture

situations. The data structure and its management decouple the declaration of the data and methods from their implementation, so that all data management operations can be fully distinct from the adaptive techniques. Finally, the AMR algorithms provide the functionality required for converting an application to adaptivity, without requiring the researcher to learn — let alone to implement — the subtle complexities of Berger's strategy.

6.1 Data Types

HAMR provides a variety of element and parameter types, to promote maximal flexibility for applications. While a typical application will use only a small fraction of the total collection of available type combinations, at various stages HAMR utilizes almost every element type and almost every parameter type.

6.1.1 Element Types

HAMR has several element types (Figure 6.1). The `Void` type is empty; its purpose is to indicate a method that returns no value, in a manner analogous to a Fortran subroutine or a Pascal procedure. As for the primitive data types — `Boolean`, `Integer`, `Index`, `Character`, `String` and `Real`[2] — their purposes are intuitively clear, and all but `Boolean` and `Index` correspond directly to C/C++ predefined types. The other element types fall into two

[2]Although Table 6.1 lists `Real` as corresponding to the C type `float`, a `Real` can be `double` instead, or `long double` on platforms supporting that type. Similarly, `Integer` and `Index` types can in principle be `char`, `short int`, `int`, `long int`, `long long int` and so on.

Element Type	Description	C type	Fortran type
`Void`	no value	`void`	
`Boolean`	true/false	`char`	`integer*1`
`Integer`		`int`	`integer`
`Index`	nonnegative	`int`	`integer`
`Real`		`float`	`real`
`Character`		`char`	`character`
`String`	C-style (null terminated)	`char *`	
`IntegerVector`	`IntegerList`	`int *`	
`IndexPoint`	`IndexList`	`int *`	
`RealVector`	`RealList`	`float *`	
`RealPoint`	`RealList`	`float *`	
`IntegerStencil`	`IntegerArray`	`int **`	
`IndexRegion`	`IndexArray`	`int **`	
`RealRegion`	`RealArray`	`float **`	

Table 6.1: HAMR element types

categories, dimensional and extreme.

Dimensional element types — `IntegerVector`, `IndexPoint`, `RealVector` and `RealPoint` — are short lists, typically of length d in a d-dimensional domain, that take on the roles indicated by their names. Thus, `IntegerVector` and `RealVector` correspond to the mathematical definition of a vector, and `IndexPoint` and `RealPoint` correspond to the mathematical definition of a point. Typically, the real types correspond to physical space, while the integer types correspond to computational space. For example, the dimensions of a region in physical space are indicated by a `RealVector`, while a single index in a three-dimensional computational domain is indicated by an `IndexPoint`.

The *extreme* element types — `IntegerStencil`, `IndexRegion` and `RealRegion` — are small arrays, one of whose indices denotes an axis and the other of which denotes an extreme, specifically *minimum* or *maximum*. Data items whose element type is `IntegerStencil` are indexed as `stencil[axis][extreme]`. Specifically, an `IntegerStencil` is a set of d 2-vectors, each of which describes the increments around a center point along a particular axis; for example, the `IntegerStencil` for a five-point centered space difference scheme would have values -2 and 2 along each axis. A `Region`, on the other hand, is indexed as `region[extreme][axis]`; that is, a `Region` denotes a pair of diagonally opposite end-points in either physical or computational space. For example, if a grid has endpoints $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$, then `region[MINIMUM]` will contain $(x_{\min}, y_{\min}, z_{\min})$ and `region[MAXIMUM]` will contain $(x_{\max}, y_{\max}, z_{\max})$.

6.1.2 Parameter Types

HAMR's parameter types are divided into four categories: structured, dimensional, spatial and method.

A *structured* parameter type — for example, a scalar, a list, an array — is a type whose computational shape must be explicitly expressed by its attributes (Table 6.2, depicted in Figure 6.3). Specifically, the levels of complexity of structured types are `Single`, `List`, `Array`, `ArrayList`, `ArrayArray` and `Queue`. `Arrays`, `ArrayLists` and `ArrayArrays` come in two varieties, those having varying dimensions and those having fixed dimensions; the latter are called `Boxes`. The appropriate element types for structured parameter types span all of HAMR's element types except `Void`.

As can be seen in Figure 6.3, structured types are implemented contiguously; that is,

Parameter Type	Description	Example
`Single`	scalar or single instance of an element type	dimension; filename
`List`	one-dimensional contiguous list	relative time value
`Array`	two-dimensional array with varying columns per row	boundary region per grid for determining parents
`ArrayBox`	two-dimensional array with fixed columns per row	flags for which boundaries reflect
`ArrayList`	three-dimensional array (list of two-dimensional arrays)	
`ArrayListBox`	three-dimensional array with fixed dimensions	grid's parent regions
`ArrayArray`	four-dimensional array (array of two-dimensional arrays)	
`ArrayArrayBox`	four-dimensional array with fixed dimensions	
`Queue`	FIFO linked list	list of initial root level grids

Table 6.2: Structured parameter types

```
arrayarraybox[arraylistnum][arraynum][listnum][eltnum]
```

(a) sequential dereference

```
***arrayarraybox[((arraylistnum * arrays_per_arraylist +
                   arraynum)    * lists_per_array      +
                   listnum)     * elements_per_list    + eltnum]
```

(b) contiguous index

Figure 6.2: Indexing in structured parameter types

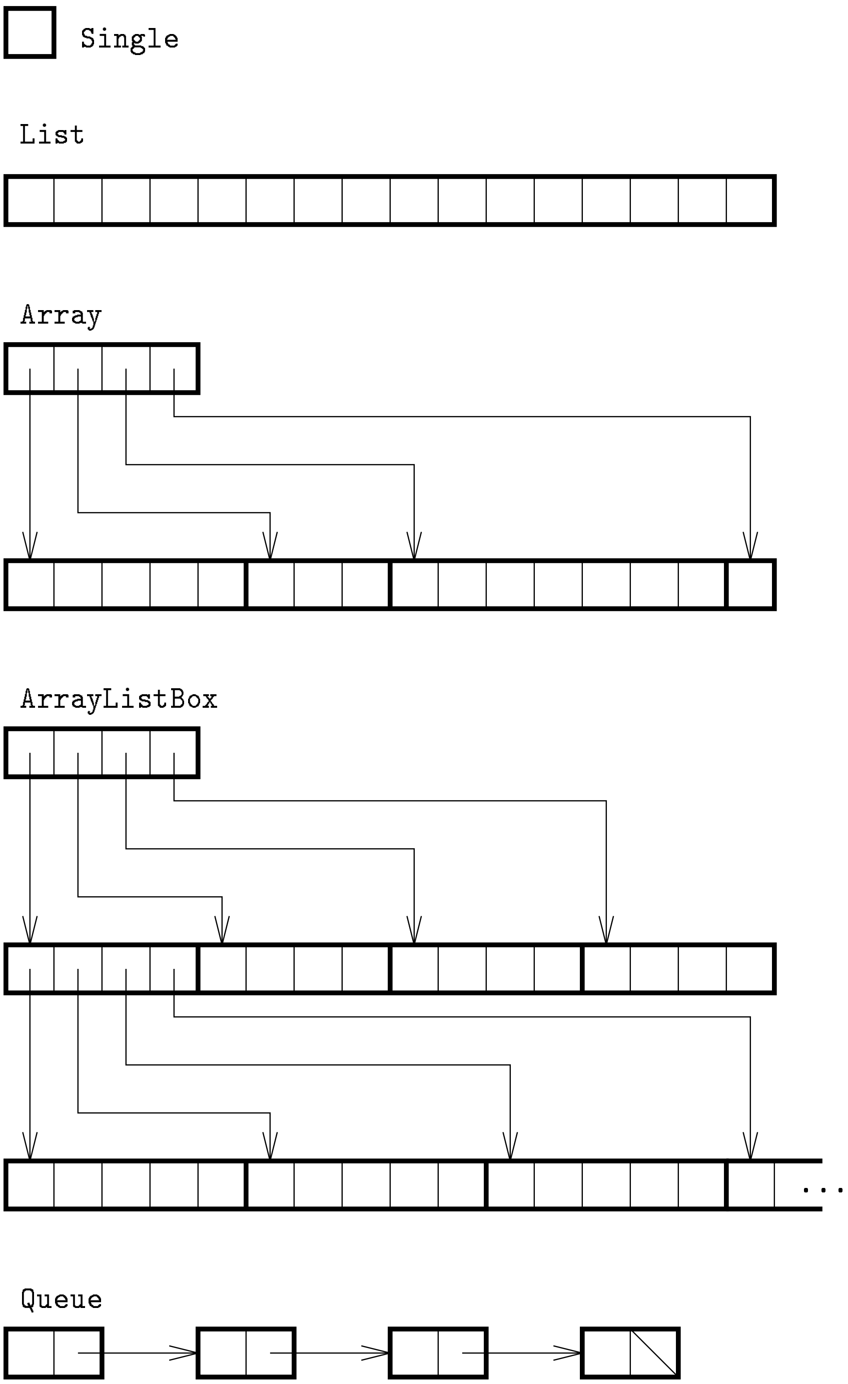
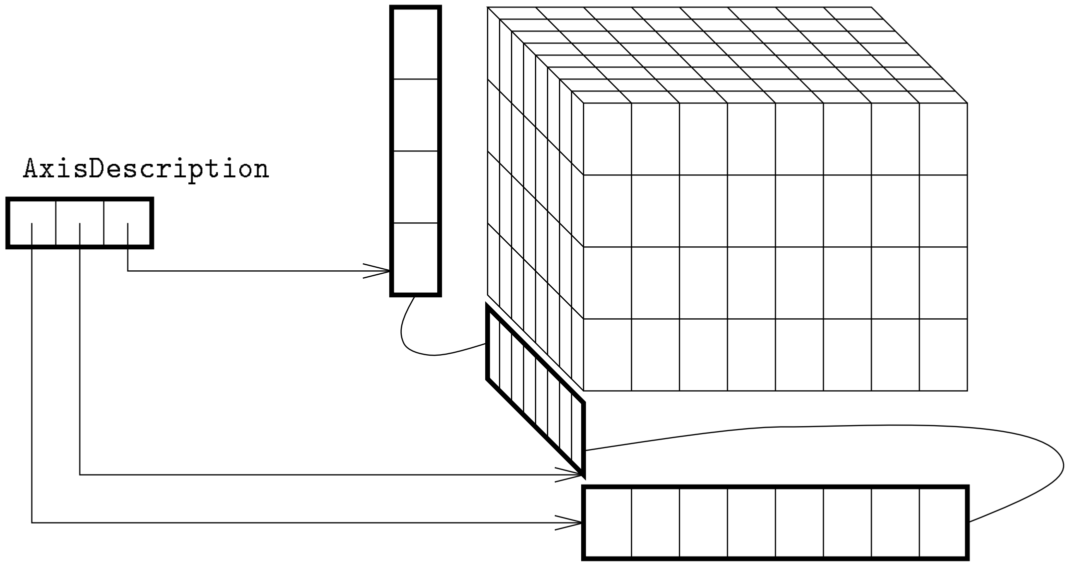


Figure 6.3: Structured parameter types

a `List` is contiguous, an `Array` is a contiguous list that is delimited by a contiguous list of pointers to `List`, an `ArrayList` is a contiguous list that is delimited by a contiguous list of pointers to `List` that in turn is delimited by a contiguous list of pointers to `Array`, and so on. (The exception is the `Queue`, which is a linked first in/first out structure. Also, in the case of `String` structures, the `Strings` themselves are not contiguous, though the structure of pointers is.) Thus, each element of a structured type can be accessed in two ways, either by a series of pointer dereferences, or by dereferences on the outer pointer lists and an index on the innermost, contiguous list (Figure 6.2). This seemingly redundant approach provides both the semantic power associated with Fortran-like indexing — for example, `arrayarraybox(eltnum,listnum,arraynum,arraylistnum)` — and the kinds of data locality optimizations associated with contiguous memory blocks. Specifically, a lone reference is most conveniently expressed as a sequential dereference, while a loop over a large portion of the array can be most effectively optimized if expressed as an index into the innermost, contiguous block.

A *dimensional* parameter type is one that has a set of values along each axis of the domain. Currently, HAMR offers two different dimensional parameter types: `AxisDescription` and `AxisContribution`. An `AxisDescription` (Figure 6.4) is simply a list of values along each axis; for example, the positions of the cell centers. An `AxisDescription` is contiguous, and in fact is implemented as an `Array`: the number of lists is the dimension of the domain, and the elements for each list are the number of cells or nodes along each axis, depending on the axis staggering. The appropriate element types for `AxisDescriptions` are `Boolean`,



Note: the axis lists are implemented as a contiguous list.

Figure 6.4: `AxisDescription`

`Integer`, `Index` and `Real`.

An `AxisContribution` (Figure 6.5) is a bit more complicated. It is, in fact, a programming convenience for certain kinds of interpolators, specifically interpolators of the form

$$\mathrm{u}^l(\mathrm{u}^{l-1}, x_1, ..., x_d) \ = \ \mathbf{F}_{\mathrm{interp}}(\mathbf{F}_{x_1}(x_1), ..., \mathbf{F}_{x_d}(x_d), \mathbf{F}_1(\mathrm{u}^{l-1}), ...\mathbf{F}_k(\mathrm{u}^{l-1}))$$

that operate in orthogonal coordinate systems. In this case, the functions $\mathbf{F}_{x_i}$ are time-invariant, and are defined over the entire computational domain on each level. Therefore, these functions can be precomputed and stored when the level is created, thus saving computing time but occupying minimal storage space, specifically $\mathbf{O}(n)$ for an $\mathbf{O}(n^d)$ computational domain. The `AxisContribution` is the storage space for these precomputed values. Because

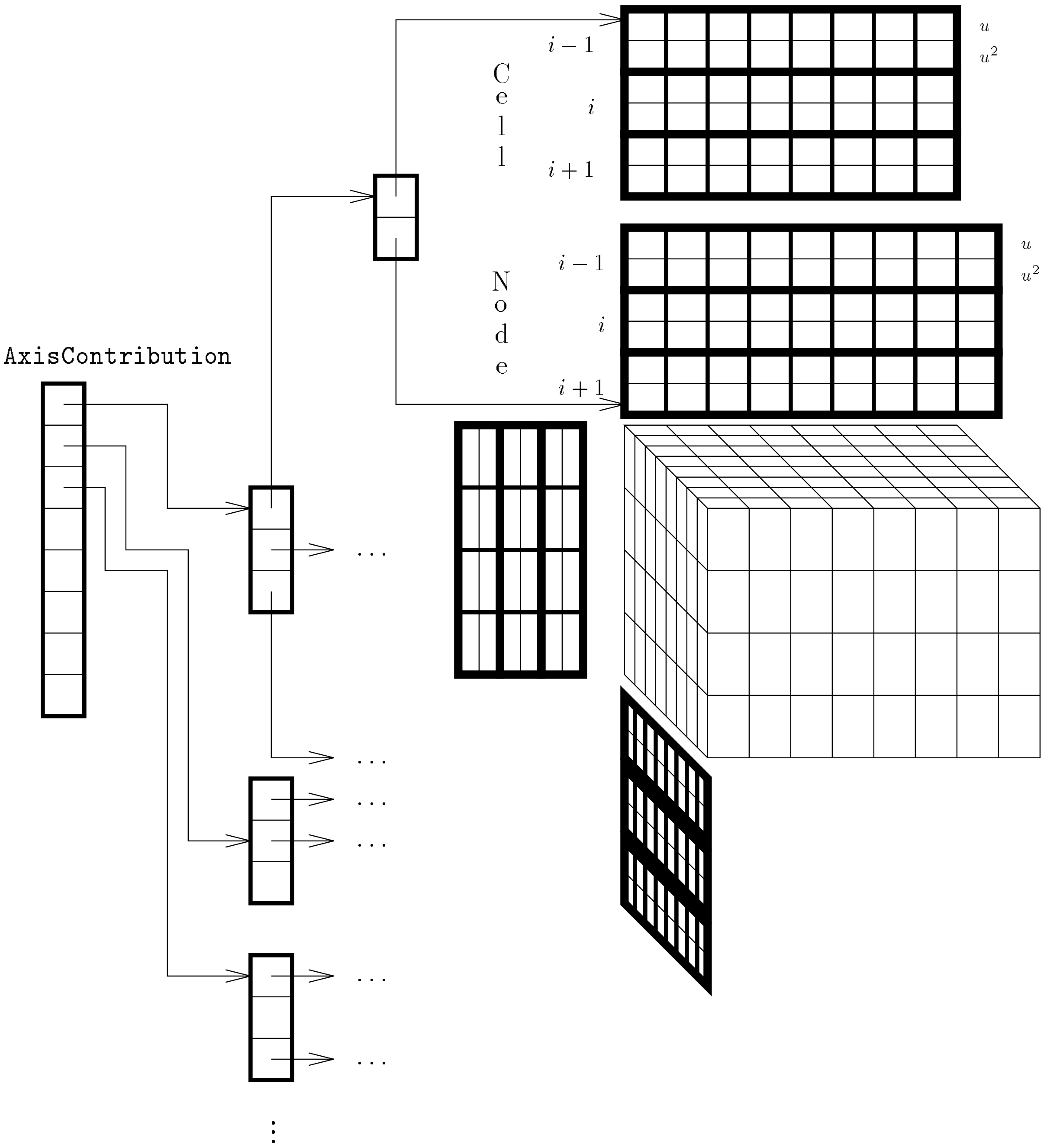


Figure 6.5: `AxisContribution`

Parameter Type	Description	Example
`SpaceParameterSet`	set of time-independent spatial vectors	selection flags
`SurfaceParameterSet`	set of time-independent spatial vector surfaces	correction vectors
`MaximalParameterSet`	set of time-independent $(d-k)$-dimensional spatial vectors of maximal size, $1 \leq k \leq d$	work space for direction sweep methods
`SpacetimeVariableSet`	set of time-dependent spatial vectors	solution
`SurfaceVariableSet`	set of time-dependent spatial vector surfaces	parental boundary vectors
`MaximalVariableSet`	set of time-dependent $(d-k)$-dimensional spatial vectors of maximal size	

Table 6.3: Spatial parameter types

`AxisContributions` are used in interpolation, the only appropriate element type is `Real`.

A *spatial* parameter type is a structure that is defined over the computational domain or over a subdomain (Table 6.6, shown in Figure 6.3). The appropriate element types are `Boolean`, `Character`, `Integer`, `Index` and `Real`.

An important property of spatial parameter types is that, rather than having individual solution vectors as a spatial parameter type, such vectors come in *sets*, where all members of the set have the same attributes.[3] Thus, for example, an application might have density, energy and velocity components as one set of solution vectors, and the magnetic field components as another, perhaps defined on different staggerings. Sets of vectors provide

[3]Thanks to M. Norman for proposing this idea.

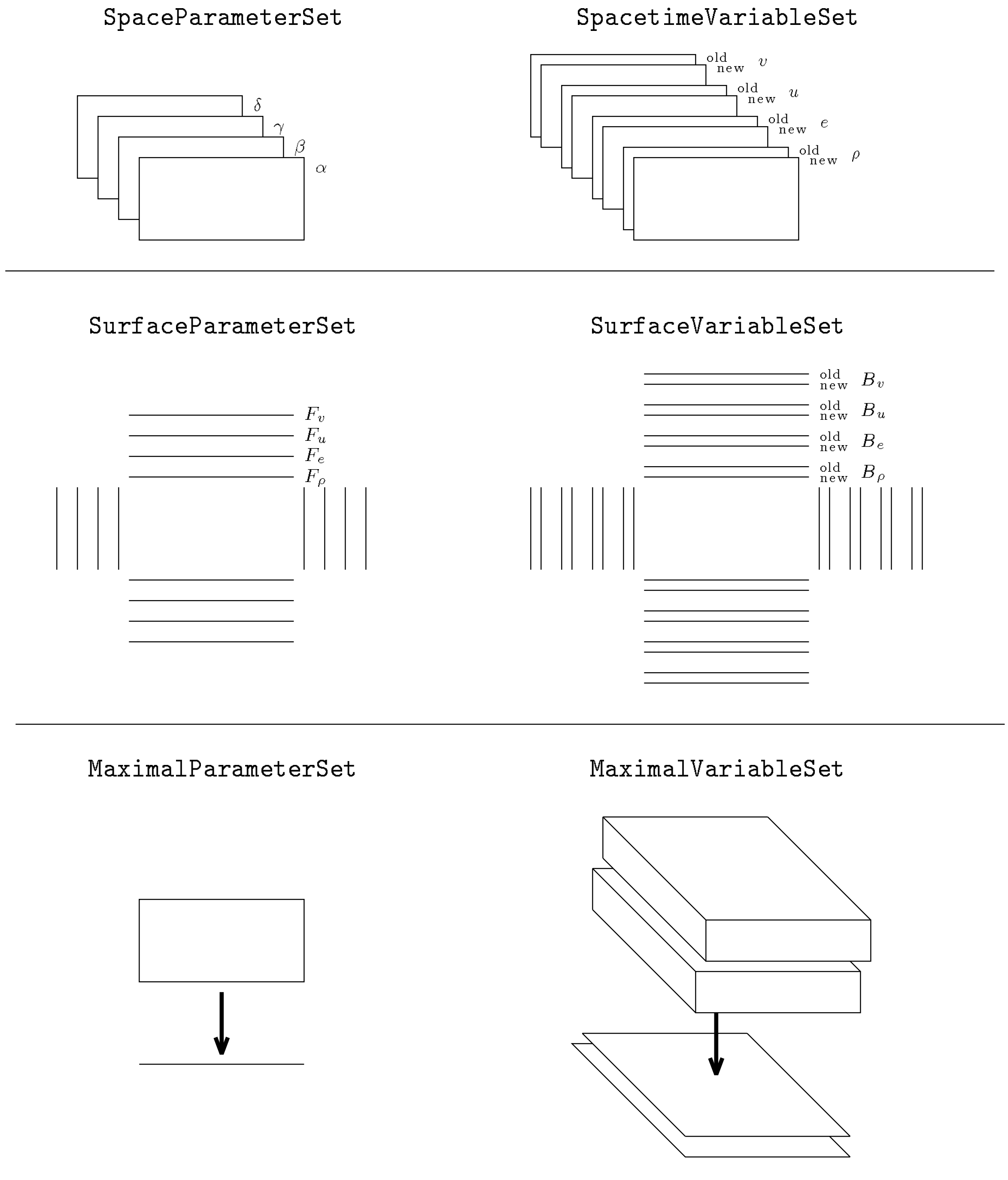


Figure 6.6: Layout of spatial parameter types

an additional advantage: potentially, they can reduce the number of entries in the formal argument list of an application subroutine, which may be an issue if the subroutine requires a great many arguments and if the compiler has a hard-coded limit on the maximum length of the argument list.

HAMR provides six spatial parameter types, grouped into pairs, with each pair having a time-dependent version of the type called a `Variable`, and a time-independent version called a `Parameter`. The three pairs are:

- `SpacetimeVariableSet` and `SpaceParameterSet`;
- `SurfaceVariableSet` and `SurfaceParameterSet`;
- `MaximalVariableSet` and `MaximalParameterSet`.

The first pair is the set of vectors that completely cover the subdomain. The most obvious example of a `SpacetimeVariableSet` is the set of solution vectors; a good example of a `SpaceParameterSet` is the set of selection flags corresponding to each solution vector.

The `Surface` sets, on the other hand, cover only the $(d-1)$-dimensional interface surfaces of the subdomain; that is, the two endpoints in 1D, the four sides in 2D, the six faces in 3D, and so on. An example of a `Surface` set is the set of correction vectors for a set of solution vectors.

Finally, `Maximal` sets are sets of a particular rank whose size is the largest possible surface of that rank. For example, for a grid of $30 \times 40 \times 20$ cells, the maximal set of rank 1 has 41 nodes, and the maximal set of rank 2 has 31×41 nodes. (The rank d

maximal set is simply the collection of all nodes for the grid.) `Maximal` sets are provided as a programming convenience, as they can be used as temporaries in certain finite difference schemes.[4] For example, in directional sweep strategies, some components of the difference scheme are applied to each *slice plane* of the grid in succession, marching along a particular axis, and then similarly along the next axis, and so on. Often, the requisite temporary work space matches the size of the slice plane. However, rather than allocating space sufficient for each of the d possible planes, a single maximal surface can be allocated, to be used by each sweep in turn.

Spatial set types are implemented as contiguous blocks, in precisely the same manner as structured parameter types; in fact, they are built on top of the type definition for `Lists`. A `SpacetimeVariableSet` has three indices: the variable within the set, the time level, and the index within the mesh. In this case, the mesh index is a composite index, rather than a reference for each dimension. As with structured parameter types, an element can be accessed either by dereferences or by a global index (Figure 6.7).

An exception to this contiguity property of spatial parameter types is the `Surface` types, which are not completely contiguous, because the sizes of their surface components vary according to surface plane they represent. For example, a `SurfaceVariableSet` on a $30 \times 40 \times 20$ grid has X-Y surfaces of 30×40 cells, X-Z surfaces of 30×20 cells and Y-Z surfaces of 40×20 cells. Thus, the surface sets themselves are implemented discontiguously, with contiguity only within a specific face (Figure 6.8).

[4]Thanks to G. Bryan for proposing this idea.

```
spacetimevariableset[varnum][time_level][((k * nj + j) * ni + i]
```

(a) sequential dereference

```
**spacetimevariableset
    [(((varnum      * number_of_time_levels +
        time_level) * total_loci + k) * nj + j) * ni + i]
```

(b) contiguous index

```
subroutine subrtn (stvarset, ni, nj, nk, ntl, nvar)
real stvarset(ni,nj,nk,ntl,nvar)
integer ni, nj, nk, ntl, nvar
```

(c) Fortran formal argument declaration

```
subrtn_(**spacetimevariableset, ni, nj, nk,
  number_of_time_levels, number_of_members);
```

(d) actual arguments for calling to Fortran

Figure 6.7: Indexing in a `SpacetimeVariableSet`

```
surfacevariableset
    [axis][extreme][varnum][time_level][((k * nj + j) * ni + i]
```

(a) sequential dereference

```
(**(surfacevariableset[axis][extreme]))
    [(((varnum      * number_of_time_levels +
        time_level) * total_loci + k) * nj + j) * ni + i]
```

(b) contiguous index

```
 subroutine subrtn (srfvarsetxmin, srfvarsetxmax,
+                   srfvarsetymin, srfvarsetymax,
+                   srfvarsetzmin, srfvarsetzmax,
+                   ni, nj, nk, ntl, nvar)
 real srfvarsetxmin(nj,nk,ntl,nvar), srfvarsetxmax(nj,nk,ntl,nvar),
+     srfvarsetymin(ni,nk,ntl,nvar), srfvarsetymax(ni,nk,ntl,nvar),
+     srfvarsetzmin(ni,nj,ntl,nvar), srfvarsetzmax(ni,nj,ntl,nvar)
 integer ni, nj, nk, ntl, nvar
```

(c) Fortran formal argument declaration

```
subrtn_(
  **(surfacevariableset[X][MINIMUM]), **(surfacevariableset[X][MAXIMUM]),
  **(surfacevariableset[Y][MINIMUM]), **(surfacevariableset[Y][MAXIMUM]),
  **(surfacevariableset[Z][MINIMUM]), **(surfacevariableset[Z][MAXIMUM]),
  ni, nj, nk, number_of_time_levels, number_of_members);
```

(d) actual arguments for calling to Fortran

Figure 6.8: Indexing in a `SurfaceVariableSet`

Finally, `Maximal` sets are implemented and indexed in precisely the same manner as their `Space/Spacetime` counterparts, but are of lower dimension.

The final category of parameter types, methods, are types whose values are pointers to functions. Specifically, the two method parameter types implemented in HAMR are `Method` and `SetMethod`; the latter is a set of methods corresponding to a spatial set, with one method value for each member (or some subset of the members) of the archetypal spatial set. Method types can have as element types `Boolean`, `Character`, `Integer`, `Index`, `Real` and `Void`.

6.1.3 Type Attributes

The data types that HAMR defines employ a variety of both structural and functional attributes. These vary according to the shapes of the data types and the roles that the types play.

All stratiform types have a `parameters` attribute, which indicates how many data items of that stratiform type have been declared, and each data item has a `name` attribute. In addition to these universal attributes, each data type has its own set of attributes, which describe its computational shape and its relationships to other data types.

6.1.3.1 Structural Attributes

Among element types, `Voids`, scalars and `Strings` contribute no attributes to their overall data types. Dimensional and extreme element types contribute a single attribute, the number of `axes` that they span. Superficially, this attribute may appear redundant. However, some

Parameter Type	Structural Attributes
`Single`	none
`List`	`elements` (`Integer`)
`Array`	`lists` (`Integer`), `elements` (`IntegerList`)
`ArrayBox`	`lists` (`Integer`), `elements per list` (`Integer`)
`ArrayList`	`arrays` (`Integer`), `lists` (`IntegerList`), `elements` (`IntegerArray`)
`ArrayListBox`	`arrays` (`Integer`), `lists per array` (`Integer`), `elements per list` (`Integer`)
`ArrayArray`	`arraylists` (`Integer`), `arrays` (`IntegerList`), `lists` (`IntegerArray`), `elements` (`IntegerArrayList`)
`ArrayArrayBox`	`arraylists` (`Integer`), `arrays per arraylist` (`Integer`), `lists per array` (`Integer`), `elements per list` (`Integer`)
`Queue`	none

Table 6.4: Attributes of structured parameter types

dimensional and extreme data items have a different rank than the computational domain. The most obvious example is the refinement factor, an `IntegerVector` that has entries not only for each dimension but also for time, thus requiring $d + 1$ elements.

The attributes of data items of structured parameter types explicitly describe the items' computational shapes (Table 6.4). All of these structural attributes are referential; that is, each is a pointer to a data item of the appropriate type. In addition, all structured types except scalars and linked types have a nonreferential `permanence` attribute, which indicates their extent category. (Linked structures have no `permanence` because they are necessarily created on the fly.)

An `Integer` can have not only its intrinsic role, but also roles with respect to other data items. For example, it may be the number of `elements` or `elements_per_list` of any of the structured types (except `Single`). Similarly, an `IntegerList` can take on the role of `elements` for an `Array`, `lists` for an `ArrayList` or `arrays` for an `ArrayArray`; an `IntegerArray` can take on the roles of `elements` for an `ArrayList` or `lists` for an `ArrayArray`; an `IntegerArrayList` can take on the role of `elements` for an `ArrayArray`.

Similarly, an `IntegerStencil` can play other roles; specifically, it can be the `stencil` attribute of a dimensional or spatial data item, or it can be the `increment` attribute of an `AxisContribution` (see below).

A `Level RealList` may play one other role: it may contain the relative time values of a spatiotemporal data item. Whether that is the case for a specific data item depends on whether it has any reciprocal time attributes, which is indicated by nonzero `time_module_dependants`. If so, then it has additional attributes that are derived from the values of the `List` and from other time information. The additional temporal attributes (Figure 6.9) are the number of `time_levels`, which is actually a reference to the list's `elements` attribute, and the time level indices among those stored that represent the old and new solutions, `old_time_level` and `new_time_level`. For each stored time level, the data item has these attributes: the time interval between the given timestep and the timestep previous to it, `time_interval_from_previous`; the absolute physical time, `absolute_time`; the overall timestep for the level, `absolute_level_timestep`; and the contribution of each parent time level to the injected value of the specific time level of the data item, `time_contribution`. Thus, a data item that contains relative time values has four `RealList` attributes attached to it, each of the same length as the data item, which describe the listed time information.

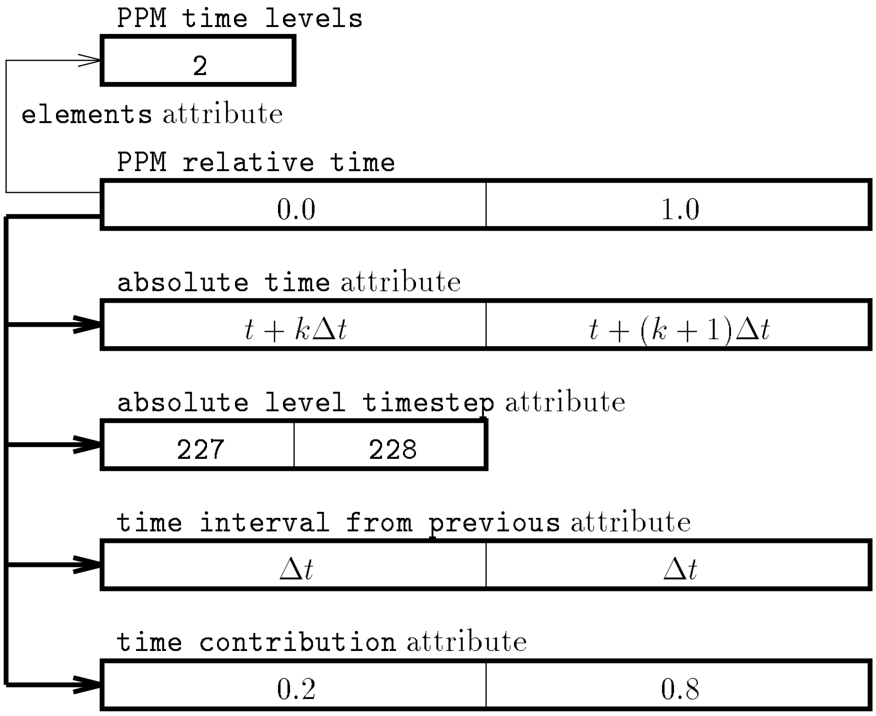


Figure 6.9: Time attributes of a `RealList`

Dimensional parameter types have fewer attributes. The structural attributes of an `AxisDescription` are `axis_loci`, which is the number of loci along each axis, and which is an `IntegerVector`; `axis_staggering`, the staggering along each axis (that is, cell-centered or on the nodes); a `stencil`, which is a variform reference to an `IntegerStencil`; and a `Boolean permanence`. An `AxisContribution` has several structural attributes: the interpolation `increment`, a variform `stencil` so that its values can cover not only the computational domain but also any exterior regions to which it might be applied; the number of `axis_loci`, and the `relationship` it applies to (that is, injection or projection). `AxisContribution` data items have no `permanence` attribute, because they are implicitly permanent.

The parameter types with the most structural attributes are the spatial parameter types, which have the following: the relative `spatial_resolution` (that is, the data item can have the resolution of the level it is on, the immediately coarser level, or the immediately finer level) and the associated `refinement_factor`; a variform reference `stencil_pointer` and a local copy of the the `stencil` itself, to ensure consistency in the event that the stencil referent changes after the instance of the data item is created; the `axis_set` (for example, in three dimensions, all axes, an XY-plane, the XZ-plane, the X-axis, and so on); the `staggering`; and the number of loci along each axis, `axis_loci`. In addition, all spatiotemporal parameter types have variform time attributes, one of which points to a `Level RealList` data item that contains the relative time values, and the rest of which point to that data item's time attributes.

In addition to these attributes, `SurfaceParameterSet` and `SurfaceVariableSet` data items have a `thickness` attribute. If the `thickness` attribute is empty, for example in the case of a set of correction vectors, then the thickness of the surfaces is one locus; that is, the surface is a set of computational planes corresponding to the interface of the grid. Otherwise, each extreme along each axis can have its own thickness, according to the instance of the data item that is declared as the thickness attribute. For an example of a `SurfaceVariableSet` with non-unitary thickness, consider a case in which a surface is used to store the spatially interpolated values of the boundary region of the grid, to save computation time during iterations on the grid's level. (This case might arise for an interpolation scheme that is very computationally expensive relative to the cost of advancing the solution, for example a higher order conservative interpolation.) Here, the thickness of the boundary surface would be the ghost boundary stencil of the associated solution vector. From these spatially interpolated values, the spatiotemporally interpolated boundary values can be obtained with minimal computational overhead, such as would be incurred with linear time-weighted interpolation, which is a very common time interpolation scheme.

`MaximalParameterSet` and `MaximalVariableSet` data items have a `rank` attribute rather than a `thickness` attribute, which indicates the number of dimensions of the maximal surface; that is, a `rank` of 1 indicates the longest line, a `rank` of 2 indicates the plane of greatest area, and so on. If the `rank` is d, then the maximal surface is the space covered by the grid's nodes.

Spatiotemporal parameter types also have attributes expressing their time characteristics. Among these are the `temporal_resolution` (that is, whether its time information is drawn locally, from the parent level or from the child level), and references to the `relative_time` list and its associated time attributes.

Resolutions are an important property of spatial sets. The spatial resolution of a set indicates whether the set has the resolution of the level that encapsulates it, the resolution of the immediately coarser level, or the resolution of the immediately finer level. The advantage of this approach is that it allows every set that derives its shape from a particular grid to be a data item for that grid. For example, the correction vectors that are ultimately applied to a parent grid have shapes based on its children (Figure 4.14). Thus, the correction vectors are data items of the child grids, thereby requiring minimal extra information to determine their shapes and locations in the domain. But the set of fluxes taken from the parent grid are stored in a child-shaped surface with the parent's resolution, while the aggregate flux sums taken from the child grid are also stored in a child-shaped surface, but with the child's resolution. The operations that combine these fluxes can be performed locally on the child, then transferred to the parent when the result is obtained (Figure 6.10).

Finally, spatial data types also have attributes that describe the characterics of their set properties. First, every set has a number of `members`, each of which has a `member_name`. Next, every set has a (possibly empty) `archetype` attribute. The archetype of the set s_1 is another set s_2 such that the members of the set s_1 are a subset of the members of s_2. Finally, every set has a list of member indices that directly map each member of the set to a member of its archetype. The advantage of this approach is that some sets are used for purposes that do not necessarily apply to all the members of the set's archetype. As an example,

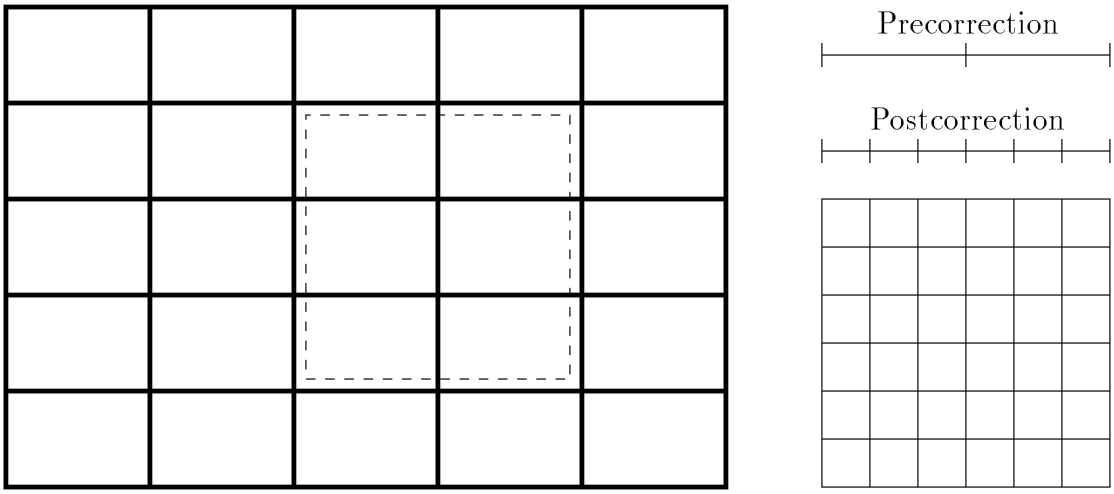


Figure 6.10: Spatial data items of different resolutions

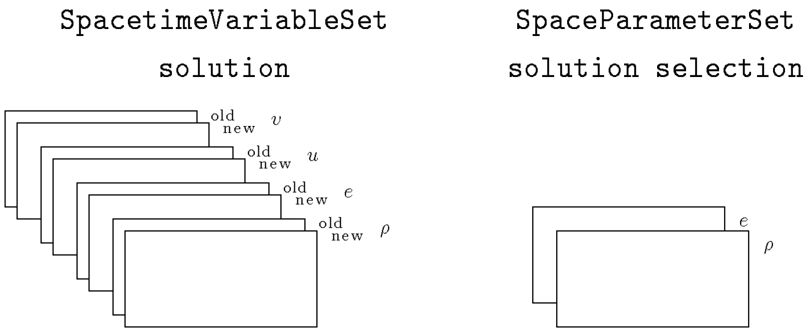
consider a selection criterion that concerns itself only with a subset of the variables. In this case, the selection `SpaceParameterSet` data item should have vectors only corresponding to the variables from which grid points are selected, to minimize memory consumption (Figure 6.11).

As for method parameter types, they are very limited in their structural attributes; in fact, `Methods` have none. `SetMethods` have an one structural attribute, the set archetype, which maps the members of the `SetMethod` to the members of its spatial set archetype, in precisely the same manner as for set archetype relationships between spatial sets.

6.1.3.2 Functional Attributes

In addition to structural attributes, all data types have functional attributes as well.

Structured types have an initialization method attribute `initialize`, a finalization




```
SpacetimeVariableSet solution: ...
  Member density, Member energy,
  Member u-velocity, Member v-velocity, Member w-velocity;
SpaceParameterSet solution selection: ...
  Archetype solution, Member density, Member energy;
```

Figure 6.11: Set archetype

method attribute **finalize**, and a list of initial values. (Typically, a structured data item will have either an initialization method or a set of initial values, but in principle nothing prohibits it from having both.) When an instance of a structured data item is created, its initialization sequence first determines whether it has any initial values, and if so assigns them, and then checks whether it has an initialization method, and if so executes it. If there are initial values, then a cascade of choices determines which values are assigned to which elements:

- If there are at least as many initial values as components of the type being initialized, then the initial values are simply copied into the components, up to the number of components.

- If there is only one initial value, it is assigned to all the components.

- If the element type is an extreme type, and there are two initial values, then they are assigned respectively to every pair of extrema in every element.
- If the element type is a dimensional type, and there are no more initial values than axes, then each initial value is assigned to the component corresponding to the associated axis, with the last initial value being propagated to further axes if necessary.
- If the element type is an extreme type, and there are no more initial values than twice the number of axes, then each initial value is assigned to the component corresponding to the associated axis, with the last initial value being propagated to further components if necessary.
- Otherwise, the initial values are assigned to the first set of elements, and then the last few initial values are assigned to the rest of the components, according to the other rules.

Of course, the latter rules can be circumvented by providing sufficient values to cover all components, and thus employing only the first rule.

Like structured parameter types, dimensional parameter types also have attributes for initialization and finalization methods, and these attributes apply in precisely the same way as for structured parameter types.

Spatial types have several different functional attributes, referring to data as well as methods. In addition to methods for initialization and finalization, the methods of a spatial data item are `inject`, `project`, `extrapolate`, `correct`, `select`, `input`, and `output`. (To promote disambiguity, method functional attributes take on the verb form of the operation name.) These methods are typically called from within a module, and they operate on the particular data item. For example, the `inject` attribute of a `SpacetimeVariableSet` applies a specific injection method from a parent's instance of the data item to a child's instance, over a specified computational region on each.

Spatial types also have several functional data references:

- `collection`, which contains the ghost boundary values collected from the parent, and can either be the raw parental values at the parents' resolution, or the spatially interpolated values at the local resolution;
- `extrapolation`, which contains the values on the exterior of the computational domain, for example in a case where those values are time-independent but expensive to calculate;
- `precorrection`, which contains the flux values obtained by a grid's parents, at the parents' resolution, which will later be compared to the aggregate flux values of the grid itself;
- `postcorrection`, the aggregate flux values of the grid;
- `selection`, the flags for refinement on each variable from which refinement regions are selected.

(To promote disambiguity, data functional attributes take the noun form of the name of the operation.)

As for method parameter types, their attributes describe the set of potential function values that they can take on: the number of potential_values; a list containing a function pointer corresponding to each potential_value; and the current value index. In the case of a SetMethod, the number of potential_values is an IntegerList, with one entry per member, and the structure containing every potential_value entry is an array of function pointers. In the case of a Method, the respective attributes are an Integer and a list of function pointers. In addition, Method data items have a data_method_value_index and SetMethod data items have a set_method_value_index; the former is a single index indicating which of the potential function pointer values has been assigned to the data item, and the latter is a list of such indices, one for each member of the set.

6.2 The HAMR Function Library

The HAMR function library comprises the functionality required for the many data types used by grid hierarchies in Berger's adaptive mesh refinement strategy. The library has a set of subroutines that operate on each category of parameter type. Each library function is designed in a manner that promotes generality, by decoupling the specifics of the size and shape of the data item(s) on which it operates from the operation itself; typically, this design goal is achieved by providing support for as many cases as possible — including, for example, support for each possible rank of a spatial type, and separate cases for operand aliasing.

6.2.1 Structured Library Functions

The HAMR function library provides several categories of operations on structured types:

- memory management;
- assignments;
- reductions;
- comparisons;
- unary operations;
- binary operations.

6.2.1.1 Memory Management

Associated with every data type, are memory management functions that perform allocation and deallocation of instances of the data type. Specifically, for each element type, there are corresponding memory management operations over all the structured parameter types. If the element type is a scalar, then the most primitive operation is the allocation of a list of that element type. However, if the type is non-scalar, then the most primitive operation is the allocation of a single instance of the type, and there is also associated an operation that allocates a *framework* for a list of the element type. In the case of `Strings`, for example, there is a function `String_allocate` that allocates space for a single `String`, and a function `StringList_framework_allocate` that allocates space for the list of pointers

```
ElementTypeList ElementTypeList_framework_allocate (Integer elements)
{ ElementTypeList elementtypelist; Integer e;
  elementtypelist = memory_allocate(sizeof(ElementType) * elements);
  for (e = 0; e < elements; e++) elementtypelist[e] = NULL;
  return elementtypelist; }
```

Figure 6.12: Framework allocation

to `Strings`. The other non-scalar element types are designed similarly; for example, an `IntegerStencilList` is actually a list of pointers to `IntegerStencils`.

Thus, every allocation — except of a list of scalars or of a single instance of a non-scalar element type — is actually an allocation of a framework. After each framework is allocated, its entries are set to the null pointer as a matter of course (Figure 6.12), because in some cases only the framework itself is required, with the entries to be filled in later.

However, in most cases a framework is allocated because it is required as part of a full data item. In such a case, the allocation function allocates an appropriate framework, allocates the next most complex structured type, then assigns to the elements of the former pointers to the elements of the latter (Figure 6.13). The advantage of this approach is that it maximizes code reuse, since each level of complexity adds only a small amount of coding, which is built on top of the functions for less complex types.

`String` structured types are an exception to this pattern. In this case, the allocated structures do not contain any actual `String` values; rather, since the length of an instance of a `String` is rarely known beforehand, the structure is simply a framework of the appropriate complexity, and the instances are inserted on the fly.

```
ElementTypeArrayBox ElementTypeArrayBox_allocate (
         Integer lists, Integer elements_per_list, Integer axes)
{ ElementTypeArrayBox elementtypearraybox; Integer l;
  elementtypearraybox = ElementTypeArray_framework_allocate(lists);
  *elementtypearraybox = ElementList_allocate(lists * elements_per_list);
  if (elements > 1)
    for (l = 1; l < elements; l++)
      elementtypearraybox[l] =
        &(elementtypearraybox[0][l * elements_per_list]);
  return elementtypearraybox; }
```

Figure 6.13: `ArrayBox` allocation

```
ElementTypeArrayBox ElementTypeArrayBox_free (
         ElementTypeArrayBox elementtypearraybox,
         Integer lists, Integer elements_per_list, Integer axes)
{ ElementTypeList_free(*elementtypearraybox, lists * elements_per_list);
  ElementTypeArrayBox_framework_free(elementtypearraybox, lists);
  return NULL; }
```

Figure 6.14: Deallocation

Deallocation is even simpler than allocation, because the values of the elements and pointers are not of concern. Thus, deallocation requires only a recursive deallocation of the next most complex structured type, and then deallocation of the most complex framework (Figure 6.14).

Every allocation and deallocation is logged, both by the memory management routines of the type category, and by the generic memory management routines on which they are based. As a result, memory use statistics can be easily collated and reported.

6.2.1.2 Assignments

The simplest operations on a structured type are assignments, which operate on a single instance of the type.

Among the assignment operations is the assignment of a constant to all elements of a structure, or to some subset of the elements. This constant can be a scalar, or in the case of dimensional and extreme element types, a single instance of the element type. In fact, with such element types, several different assignments are provided: assigning a scalar to every component of every element, assigning a scalar to a specific individual component of every element, and, as mentioned, assigning an instance of the element type to every element. Because of the contiguous nature of the structured types, many operations on more complex types are simply calls to the same operations on less complex types, with appropriate dereferencing (Figure 6.15).

In addition to generic assignments, HAMR also provides assignments of specific, commonly used values, including zero, one, an undefined value (for example, to clear a structure of indices, which indicates that certain indices have not yet been assigned), true and false values for boolean structures, and so on. These functions are provided as a programming convenience, and are implemented by calls to more generic assignments with the appropriate constant argument.

6.2.1.3 Reductions

Reductions are operations that derive a single value from multiple elements. Among the

```
Void ElementTypeArrayListBox_set_to_constant (
       ElementTypeArrayListBox elementtypearraylistbox,
       Integer arrays, Integer lists_per_array, Integer elements_per_list,
       ElementType constant)
{ ElementTypeArrayBox_set_to_constant(*elementtypearraylistbox,
    arrays * lists_per_array, elements_per_list); }

Void ElementTypeArrayBox_set_to_constant (
       ElementTypeArrayBox elementtypearraybox,
       Integer lists, Integer elements_per_list, ElementType constant)
{ ElementTypeList_set_to_constant(*elementtypearraybox,
    lists * elements_per_list); }

Void ElementTypeList_set_to_constant (ElementTypeList elementtypelist,
       Integer elements, ElementType constant)
{ Integer e;
  for (e = 0; e < elements; e++) elementtypelist[e] = constant; }
```

Figure 6.15: Assignment to a complex structured type

reduction operations HAMR provides are aggregates, instance examinations and extrema examinations.

An *aggregate* reduction is one that performs an arithmetic operation on the elements of a structure, in order to produce a single value. HAMR provides sum, product and mean aggregates (Figure 6.16); the first two are direct calculations, while the last is accomplished by performing a sum and then dividing the result by the number of elements. The sum operation also plays an important role with respect to operations on non-`Box` `Arrays`, `ArrayLists` and `ArrayArrays`: such operations require the sum of the number of items in the next-to-outermost index, in order to determine the length of the outermost dimension of the next most complex structured type, which is required by the recursive call to the associated

```
ElementType ElementTypeList_sum (ElementTypeList elementtypelist,
                Integer elements)
{ ElementType sum; Integer e;
  for (e = 0; e < elements; e++) sum += elementtypelist[e];
  return sum; }

ElementType ElementTypeList_product (ElementTypeList elementtypelist,
                Integer elements)
{ ElementType product; Integer e;
  for (e = 0; e < elements; e++) product *= elementtypelist[e];
  return product; }

ElementType ElementTypeList_mean (ElementTypeList elementtypelist,
                Integer elements)
{ return ElementTypeList_sum(elementtypelist, elements) / elements; }
```

Figure 6.16: Aggregate operations

function on that type (Figure 6.17).

Next, an *instance examination* is an operation that searches for instances of a value and performs some operation with respect to the instances. The instance operations HAMR provides are

- identification of the first instance of the value in a structure;
- identification of the last instance of the value in a structure;
- counting the number of instances of the value;
- determining whether the structure contains any instances of the value.

The last of these operations is implemented by determining the first instance of the value: if the result of that operation is undefined, then the structure does not contain the value. (In

```
ElementType ElementTypeArray_operation (
         ElementTypeArray elementtypearray,
         Integer lists, IntegerList elements)
{
  ElementTypeList_operation(*elementtypearray,
    IntegerList_sum(elements, lists));
}

ElementType ElementTypeArrayList_operation (
         ElementTypeArrayList elementtypearraylist,
         Integer arrays, IntegerList lists, IntegerArray elements)
{
  ElementTypeArray_operation(*elementtypearraylist,
    IntegerList_sum(lists, arrays), *elements);
}

ElementType ElementTypeArrayArray_operation (
         ElementTypeArrayArray elementtypearrayarray,
         Integer arraylists, IntegerList arrays,
         IntegerArray lists, IntegerArrayList elements)
{
  ElementTypeArrayList_operation(*elementtypearrayarray,
    IntegerList_sum(arrays, arraylists), *lists, *elements);
}
```

Figure 6.17: Operations on non-Box array structures

principle, this operation could also be implemented by counting the number of instances of the value, and then determining whether that count is nonzero, and typically the count of the instances optimizes significantly better than the search for the first instance. However, the counting operation does not short circuit when an instance is found, and the advantage of optimization is unlikely to outweigh the advantage of short circuiting when examining a structure with a large number of elements.)

Finally, an *extreme examination* is like an instance examination, except that the value it examines is an extreme — that is, either the minimum or the maximum value of the structure. The extreme operations in HAMR, each of which is available for both the minimum and the maximum, are:

- determination of the value of the extreme;
- identification of the first instance of the extreme in a structure;
- identification of the last instance of the extreme in a structure;
- counting the number of instances of the extreme.

6.2.1.4 Comparisons

HAMR provides the standard comparisons — equal, not equal, less, less or equal, greater, greater or equal — both between the elements of two structures of the same shape, and between a structure's elements and a constant. Every comparison operation has a boolean

```
Void ElementTypeList_equal (
         BooleanList dstbooleanlist,
         ElementTypeList srcelementtypelist1,
         ElementTypeList srcelementtypelist2, Integer elements)
{ Integer e;
  if (srcelementtypelist1 == srcelementtypelist2)
    BooleanList_set(dstbooleanlist, elements);
  else
    for (e = 0; e < elements; e++)
      dstbooleanlist[e] =
        srcelementtypelist1[e] == srcelementtypelist2[e];
}
```

Figure 6.18: Comparison operand overlap cases

field as its destination operand; that is, the values on the source operands are mapped to flags on the destination operand.

Comparison operations (and other operations that have multiple structured operands) employ multiple versions of the operation, with each version covering a different case of operand overlap. This approach maximizes the potential for optimization, because it allows the compiler to assume legitimately that, within each loop, each individual operand refers to a distinct area of memory. In the case of comparisons between two structured operands, the two cases are that the operands are actually the same structure and that they are different structures (Figure 6.18), except in the case of boolean comparisons, in which case it may be the case that one or both of the source operands overlap the destination operand. (In practice, this rule only applies to operations on `Lists`, because operations on more complex structured types are implemented by recursively calling operations on less structured types.)

```
Void ElementTypeList_negate (
         ElementTypeList dstelementtypelist,
         ElementTypeList srcelementtypelist, Integer elements)
{ Integer e;
  if (dstelementtypelist == srcelementtypelist)
    for (e = 0; e < elements; e++)
      dstelementtypelist[e] = -dstelementtypelist[e];
  else
    for (e = 0; e < elements; e++)
      dstelementtypelist[e] = -srcelementtypelist[e]; }
```

Figure 6.19: Unary negation

6.2.1.5 Unary Operations

Unary operations require one source operand and one destination operand. HAMR's unary operations include copying, absolute value, negation, square root, exponential, and the boolean *not* operation. A typical unary operation has two cases, one in which the operands are the same structure, and one in which they are different (Figure 6.19). Technically, the source operand of a unary operation could be a constant, instead of a structured operand. However, such operations are classified as assignments, and are treated separately.

6.2.1.6 Binary Operations

Binary operations require two source operands and one destination operand. HAMR's binary operations are the standard arithmetic operations — addition, subtraction, multiplication, division and remainder — as well as the boolean operations *and*, (inclusive) *or* and *exclusive or*. A typical binary operation has five cases that address the various combinations of

```
Void ElementTypeList_add (ElementTypeList dstelementtypelist,
         ElementTypeList srcelementtypelist1,
         ElementTypeList srcelementtypelist2, Integer elements)
{ Integer e;
  if (srcelementtypelist1 == srcelementtypelist2)
    if (dstelementtypelist == srcelementtypelist1)
         for (e = 0; e < elements; e++) dstelementtypelist[e] *= 2;
    else for (e = 0; e < elements; e++)
           dstelementtypelist[e] = srcelementtypelist1[e] * 2;
  else if (dstelementtypelist == srcelementtypelist1)
       for (e = 0; e < elements; e++)
         dstelementtypelist[e] += srcelementtypelist2[e];
  else if (dstelementtypelist == srcelementtypelist2)
       for (e = 0; e < elements; e++)
         dstelementtypelist[e] += srcelementtypelist1[e];
  else for (e = 0; e < elements; e++)
         dstelementtypelist[e] =
           srcelementtypelist1[e] + srcelementtypelist2[e]; }
```

Figure 6.20: Typical binary operation

overlapping operands (Figure 6.20).[5]

In addition to binary operations on two structured source operands, HAMR also includes such operations for cases in which one operand is a constant (Figure 6.21), with a single function for each commutative operation — i.e., addition, multiplication, and the boolean binary operations *and*, *inclusive or* and *exclusive or* — and two functions for each non-commutative operation — i.e., subtraction, division and remainder — one for each ordering of the operands. For example, the two subtraction operations are `subtract_constant` and

[5] More formally, for k structured operands including the destination, there are $2^k - k$ cases, because $\sum_{i=1}^{k} \binom{k}{i} \equiv 2^k$ [Tuc84], and the concept of a single operand overlapping itself is meaningless, which eliminates $\binom{k}{1} \equiv k$ cases.

```
Void ElementTypeList_add_constant (ElementTypeList dstelementtypelist,
         ElementTypeList srcelementtypelist, Integer elements,
         ElementType constant)
{ Integer e;
  if (constant == 0)
    ElementTypeList_copy(dstelementtypelist,srcelementtypelist,elements);
  else if (dstelementtypelist == srcelementtypelist)
       for (e = 0; e < elements; e++) dstelementtypelist[e] += constant;
  else for (e = 0; e < elements; e++)
         dstelementtypelist[e] = srcelementtypelist[e] + constant; }
```

Figure 6.21: Typical binary operation with constant operand

`subtract_from_constant`, with a structured minuend for the former and a structured subtrahend for the latter.

Most binary operations with a constant operand also treat special cases based on the value of the constant. For example, the function for adding a constant to a structured data item includes special treatment of the case in which the constant is zero; specifically, the addition reduces to a copy, as does subtraction of zero, multiplication by one and division by one. Subtraction *from* zero reduces to negation, and remainder by one reduces to assignment of zero.

6.2.2 Dimensional Library Functions

For each dimensional parameter type, the HAMR function library provides appropriate functionality. For `AxisDescriptions`, HAMR provides the same functions as for structured types: the `AxisDescriptions` are implemented as `Arrays` of the same element type, and

```
Void ElementTypeAxisDescription_set_to_constant (
         ElementTypeAxisDescription elementtypeaxisdescription,
         AxisStaggering axisstaggering, IntegerVector axisloci,
         Integer axes, ElementType constant)
{ ElementTypeArray_set_to_constant(elementtypeaxisdescription,
    axes, axisloci, constant); }
```

Figure 6.22: Typical `AxisDescription` operation

so the `AxisDescription` functions are implemented as calls to the corresponding `Array` functions (Figure 6.22).

As for `AxisContributions`, they require very different operations, because of their unique purpose. Aside from allocation, deallocation and copying, the other functions on `AxisContributions` are initializers for various combinations of mesh type and coordinate system.

6.2.3 Method Library Functions

HAMR provides limited functionality for method types, because most of the operations available for other types are inappropriate for methods. Specifically, HAMR provides allocation, deallocation, copying and assigning a constant (i.e., a function pointer value). In fact, when a `MethodList` or `SetMethodList` is allocated, its entries are automatically initialized to an empty function that returns a neutral value of the appropriate element type (or no value). For example, the entries of a newly allocated `VoidMethodList` are all assigned a pointer to the function `Void_do_nothing`, which takes no arguments and has an empty function body.

This preassignment of the neutral function guarantees that executing the function pointer will not abort the run.

6.2.4 Spatial Library Functions

The HAMR function library provides the same categories of operations for spatial types as for structured types, with one additional category, interpolation. Spatial functions are intended to be maximally self-contained, in the sense that each function addresses as many cases as possible. For example, each function treats one-, two- and three-dimensional cases with equal ease. More generally, a spatial function:

- determines whether the spatial operands are linearly independent, and if not — for example, if the lengths of all the spatial operands are one locus along some axis or axes — then reduces the formal arguments to the true rank of the operands and calls the function recursively;
- determines whether any special case — that is, a case whose calculation is simpler than the general case — is applicable, and if so applies that case, often by calling another, simpler library function;
- otherwise, performs the operation, choosing the case that corresponds to the combination of spatial operands that overlap, if any.

Spatial functions come in several categories of operational complexity:

	Contiguous	Offset	Striding	Marginal	Incremental	Injection	Projection
Assignment	$\surd$	$\surd$	$\surd$				
Reduction	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$		$\surd$
Comparison	$\surd$	$\surd$	$\surd$			$\surd$	
Unary	$\surd$	$\surd$	$\surd$			$\surd$	
Binary	$\surd$	$\surd$	$\surd$			$\surd$	
Interpolation		$\surd$	$\surd$			$\surd$	$\surd$

Table 6.5: Function complexities of operation categories

- contiguous;
- offset;
- striding;
- marginal;
- incremental;
- injection;
- projection.

Most of the operation categories are relevant to only a subset of the complexity categories (Table 6.5).

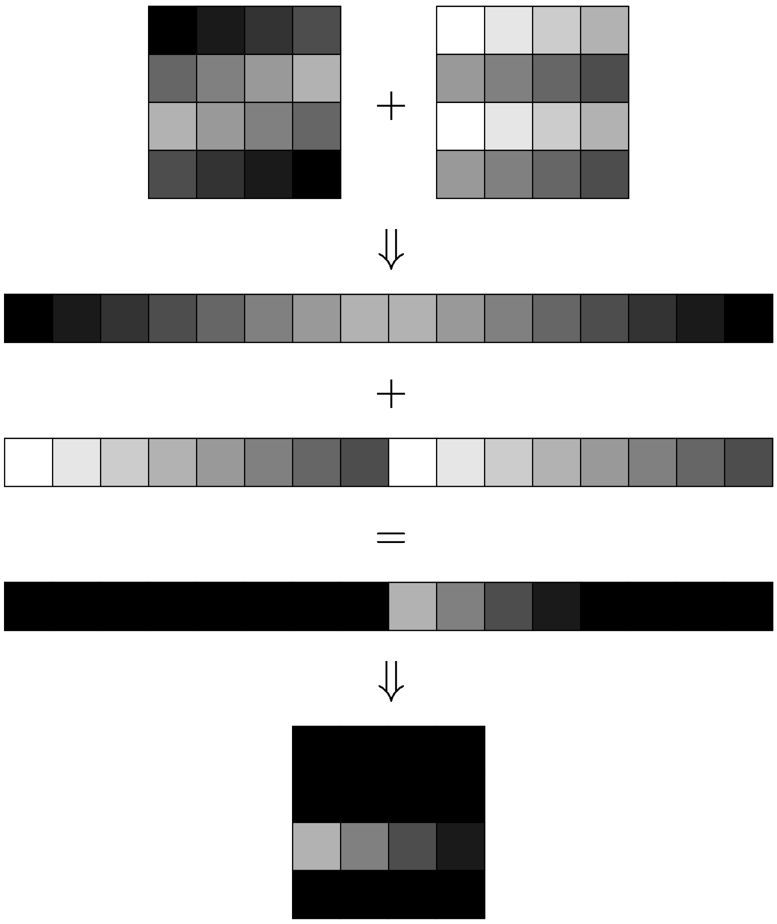
6.2.4.1 Contiguous Operations

Contiguous operations are those that treat an entire spatial operand, or treat entire multiple operands of identical shape; they are implemented by calling the corresponding function for a `List` of that element type. For example, if two operands defined over the computational interior of a grid are to be added, with the sum placed in a third operand of identical shape, then the operation can be performed contiguously (Figure 6.23).

6.2.4.2 Offset Operations

Offset operations are those that operate over a subregion of a spatial operand, or more commonly over different subregions of multiple operands (Figure 6.24). A common use of an offset operation is to perform the operation over a bounded operand, but only on the computational interior; for example, to divide the interior of one operand by the interior of another, as would be the case in obtaining velocity from momentum and density. In such a case, it can be crucial to avoid dividing the boundary as well, since the denominator operand may be known to have nonzero interior values, but there may be no such certainty about the boundary, for example if the boundary values have not been collected recently. Similarly, cases often arise in which the operation should be performed on the boundaries but not the interior, or on the boundary of one grid and the interior of another — as in the case of copying boundary values from a sibling (Figure 6.25). Indeed, offset operations are probably the most commonly used in HAMR, because they provide maximum generality for operations on a single level of resolution.

Offset operations take three structural arguments for each operand: staggering, number of loci along each axis, and starting index. In addition, such operations take a single length vector, which describes the number of loci along each axis on which to perform the operation, for all of the operand regions. This arrangement guarantees that the regions are of identical



```
Void ElementTypeField_contiguous_add (
         ElementTypeField dstelementtypefield,
         ElementTypeField srcelementtypefield1,
         ElementTypeField srcelementtypefield2,
         AxisSet axisset, Staggering staggering,
         IntegerVector axisloci, Integer axes)
{ ElementTypeList_add(dstelementtypefield,
    srcelementtypefield1, srcelementtypefield2,
    ElementTypeField_number_of_loci(axisset, staggering, axisloci, axes)); }
```

Figure 6.23: Contiguous addition

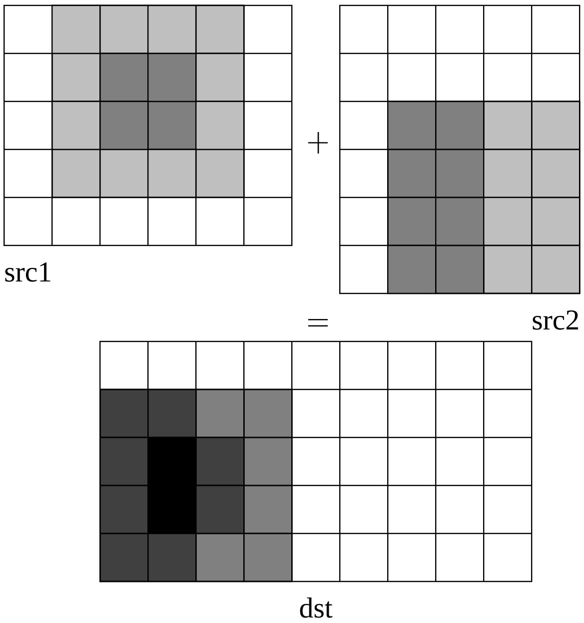


Figure 6.24: Offset addition

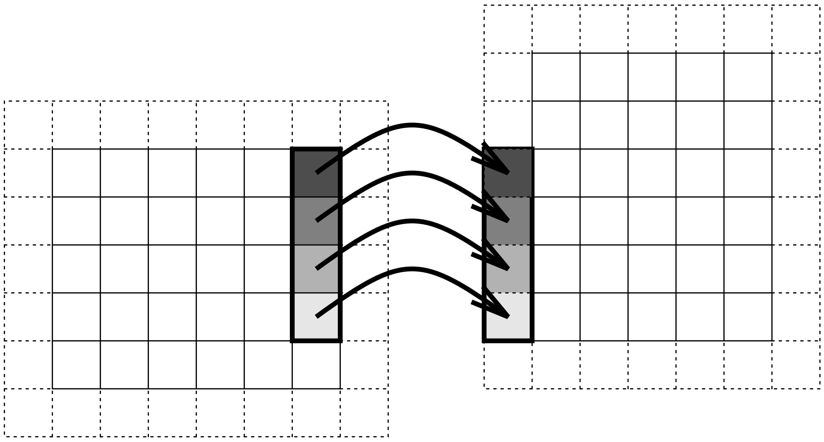


Figure 6.25: Offset copy from sibling to boundary

shape, regardless of the shapes of the operands themselves.

All offset operations except interpolation must treat two special cases. First, if the offsets cover the entire corresponding spatial operands, and the operands are of identical shape, then quite naturally the contiguous version of the operation is called. Second, if the rank of the operands is 1, then the contiguous operation can be called on the subregions indicated by the offset arguments.

If neither of the special cases apply, then the operation is performed directly. Thus, each offset function has, for each rank up to some maximum rank[6] that is defined at compile time, a set of nested loops, one for each axis (Figure 6.26).

6.2.4.3 Striding Operations

Striding operations perform over subregions of the input operands, in the same manner as offset operations, but perform the operation on only some of the loci within the subregions. Specifically, each operand has not only offset arguments but also a stride argument, an `IntegerVector` that dictates which loci will be operated on; those loci are a fixed distance apart along each axis, with the distance indicated by the corresponding stride vector component.

A common example of a striding operation is the use of a striding copy to obtain exterior boundary conditions on a reflecting boundary. Specifically, the cells closest to the exterior boundary are copied into the ghost region using a stride of -1 along the axis corresponding

[6]Currently, the maximum rank can be has high as six.

```
Void RealField_offset_set_to_constant (RealField realfield,
    AxisSet axisset, Staggering staggering,
    IntegerVector axis_loci, IntegerVector size, IndexPoint start,
    Integer axes, Real constant)
{ ...
  if (axes == 1) {
    RealField_contiguous_set_to_constant(&realfield[*start],
      axisset, Staggering_cell_center(axes), size, axes, constant);
    return; }
  switch (axes) {
    case 2:
      start1 = start[0]; size1 = size[0]; al1 = axis_loci[0];
      start2 = start[1]; size2 = size[1];
      for (a2 = 0; a2 < size2; a2++)
        for (a1 = 0; a1 < size1; a1++)
          realfield[a1+start1 + al1*(a2+start2)] = constant;
      break;
    case 3:
      start1 = start[0]; size1 = size[0]; al1 = axis_loci[0];
      start2 = start[1]; size2 = size[1]; al2 = axis_loci[1];
      start3 = start[2]; size3 = size[2];
      for (a3 = 0; a3 < size3; a3++)
        for (a2 = 0; a2 < size2; a2++)
          for (a1 = 0; a1 < size1; a1++)
            realfield[a1+start1 + al1*(a2+start2) + al1*al2*(a3+start3)] =
              constant;
      break;
  }
}
```

Figure 6.26: Nested loops in an offset operation

to that boundary, with a stride of 1 along all other axes (Figure 6.27).

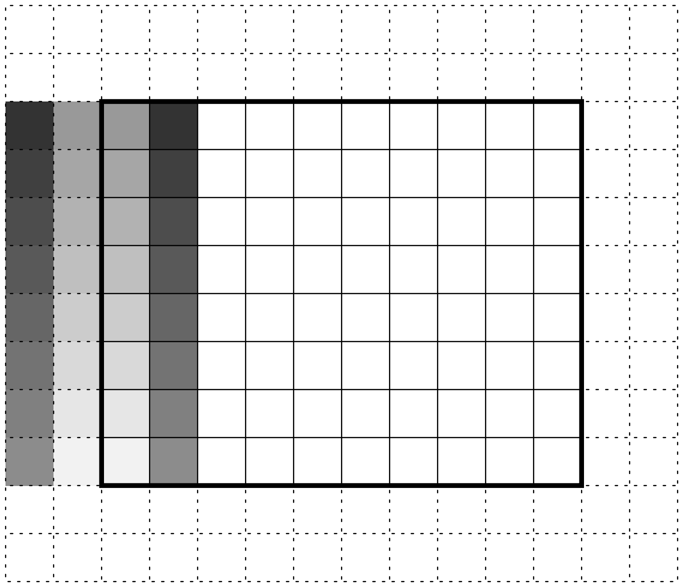
Striding operations must treat one special case, the case in which all stride vectors have value 1 for all components. In this case, the operation reduces to the associated offset operation. Otherwise, the appropriate nested loop is employed.

An important point regarding special cases is that they can be pseudo-recursively defined. In the case described above, in which all strides are 1, the call to the associated offset operation may determine that the operation is to be performed over the entirety of operands of identical size and shape. In this case, the offset operation would itself call a special case, namely the contiguous operation, which would in turn call the associated operation on a `List`.

6.2.4.4 Marginal Operations

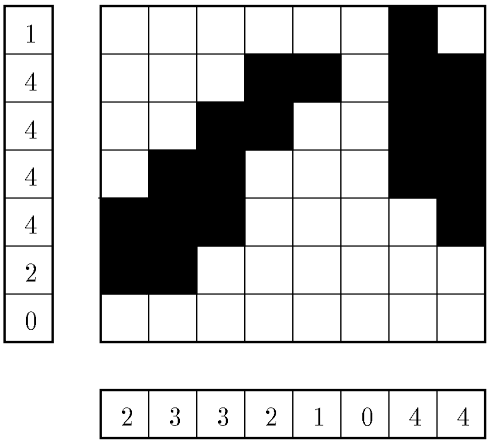
Marginal operations reduce a full d-dimensional spatial operand to a $(d-k)$-dimensional operand, based on the axis set of the destination operand. Necessarily, therefore, all marginal operations are reductions. A common example of a marginal operation is obtaining the count of the instances of selection flags that are set along a particular slice plane (Figure 6.28), an operation that plays a critical role in clustering, since this operation determines the signatures.

Marginal operations take one set of offset arguments, rather than one for each operand, but take an axis set for each operand, since the axis set describes the polytope (e.g., line, plane) that the operand represents. Specifically, the destination axis set must be a (not



```
Void RealField_striding_copy (
         RealField dstrealfield, RealField srcrealfield,
         AxisSet axisset, Staggering dststaggering, Staggering srcstaggering,
         IntegerVector dstaxisloci, IntegerVector srcaxisloci,
         IntegerVector size, IndexPoint dststart, IndexPoint srcstart,
         IntegerVector dststride, IntegerVector srcstride, Integer axes)
{ ...
  switch (axes) { ...
    case 2:
      dstart1 = dststart[0]; sstart1 = srcstart[0]; size1 = size[0];
      dstride1 = dststride[0]; sstride1 = srcstride[0];
      dal1 = dstaxisloci[0]; sal1 = srcaxisloci[0];
      dstart2 = dststart[1]; sstart2 = srcstart[1]; size2 = size[1];
      dstride2 = dststride[1]; sstride2 = srcstride[1];
      for (a2 = 0; a2 < size2; a2++)
        for (a1 = 0; a1 < size1; a1++)
          dstrealfield
              [a1*dstride1+dstart1 + dal1*(a2*dstride2+dstart2)] =
            srcrealfield
                [a1*sstride1+sstart1 + sal1*(a2*sstride2+sstart2)];
      break; ...
  } }
```

Figure 6.27: Striding copy




```
Void BooleanField_marginal_instances_of_true (
       IntegerField dstintegerfield, BooleanField srcbooleanfield,
       AxisSet dstaxisset, AxisSet srcaxisset, Staggering staggering,
       IntegerVector axisloci, IntegerVector size, IndexPoint start,
       Integer axes)
{ ...
  switch (axes) {
    case 2:
      start1 = start[0]; size1 = size[0]; al1 = axisloci[0];
      start2 = start[1]; size2 = size[1];
      switch (dstaxisset) {
        case 01b:
          for (a2 = 0; a2 < size2; a2++)
            for (a1 = 0; a1 < size1; a1++)
              if (srcbooleanfield[a1+start1 + al1*(a2+start2)])
                dstintegerfield[a1+start1]++;
          break;
        case 10b:
          for (a2 = 0; a2 < size2; a2++)
            for (a1 = 0; a1 < size1; a1++)
              if (srcbooleanfield[a1+start1 + al1*(a2+start2)])
                dstintegerfield[a2+start2]++;
          break;
      }
      break; ...
  } }
```

Figure 6.28: Marginal instances of true

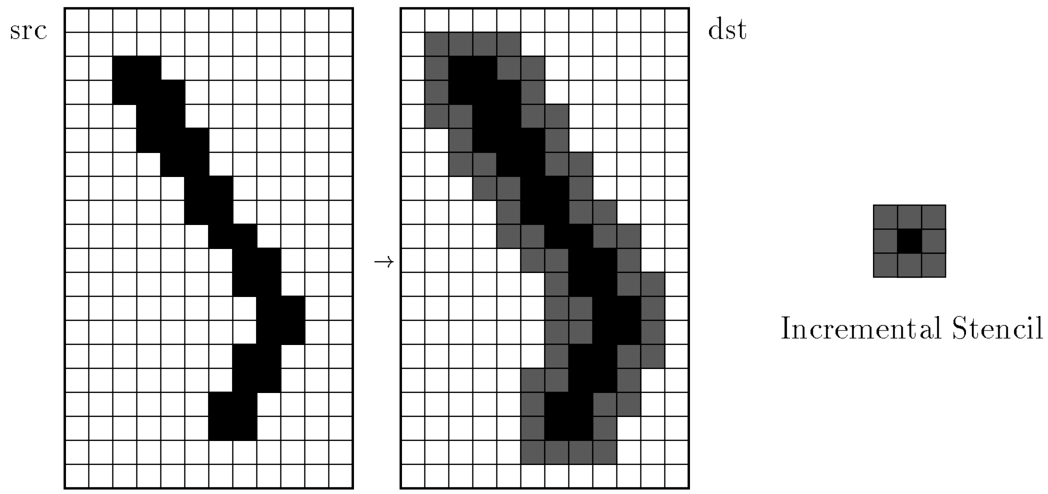
necessarily proper) subset of the source axis set. Thus, the destination operand has the shape of an appropriate subsurface of the source operand.

The special cases that marginal operations address separately from the general case are the cases of an empty destination axis set, which indicates a full reduction of the region of interest to a single scalar, and the case of identical source and destination axis sets, which indicates that the operation does not reduce, but rather simply examines the value at each locus. For example, in the case of the marginal count of the number of instances of boolean flags that are set, the empty axis set case would count the total number of flags that are set, and the complete axis set case would assign a 1 for every flag that is set and a zero for every flag that is clear.

6.2.4.5 Incremental Operations

Like their marginal counterparts, *incremental* operations are also all reductions; specifically, they reduce the loci surrounding a locus to a value on the locus. A common example of an incremental operation is an examination of a field of boolean flags to determine whether any of the surrounding flags (including the flag that is surrounded) are set (Figure 6.29). This operation is employed during selection, to expand the selected region by the size of the buffer region and any child boundaries needed.

Like marginal operations, incremental operations take one set of offset arguments, rather than one for each operand, so all operands have identical shape. They also take an increment argument, which describes the neighborhood surrounding each locus.




```
Void BooleanField_incremental_any_true (
         BooleanField dstbooleanfield, BooleanField srcbooleanfield,
         AxisSet axisset, Staggering staggering,
         IntegerVector axisloci, IntegerVector size, IndexPoint start,
         IntegerStencil increment, Integer axes)
{ ...
  switch (axes) {
    ...
    case 2:
      incmin1 = increment[0][0]; incmax1 = increment[0][1];
      al1 = axisloci[0]; size1 = size[0]; start1 = start[0];
      incmin2 = increment[1][0]; incmax2 = increment[1][1];
      al2 = axisloci[1]; size2 = size[1]; start2 = start[1];
      for (i2 = incmin2; i2 <= incmax2; i2++)
        for (i1 = incmin1; i1 <= incmax1; i1++)
          BooleanField_offset_or(
            dstbooleanfield, dstbooleanfield, srcbooleanfield,
            axisset, staggering, staggering, staggering,
            axisloci, axisloci, axisloci,
            incsize, incdststart, incdststart, incstart, axes);
     break; ...
  }
}
```

Figure 6.29: Incremental check of whether any surrounding flags are true

```
for (i1 = incmin1,
       incdststart[0] = (((start1+incmin1) < 0) ? 0 : start1+incmin1),
       incstart[0] =
         (((start1+incmin1) < 0) ? start1-incmin1 : start1),
       incsize[0] =
         (((start1+incmin1) < 0) ? size1+start1+incmin1 :
          ((size1+start1+incmin1) > al1) ?
             size1-(start1+incmin1) : size1);
     i1 <= incmax1; i1++,
       incdststart[0] = (((start1+i1)  < 0) ? 0 : (start1+i1)),
       incstart[0] = (((start1+i1)  < 0) ? (start1-i1) : start1),
       incsize[0] =
         (((start1+i1) < 0) ? (size1+start1+i1) :
          ((size1+start1+i1) > al1) ? size1-(start1+i1) : size1))
```

Figure 6.30: Incremental loop with coverage guarantees

The special case that incremental operations address separately from the general case is the case of an increment whose entries are all zero. In this case, the operation reduces to the same simple operation as the identical axis sets case of the corresponding marginal operation.

Regarding the incremental loops: in fact, they are considerably more complicated than depicted in Figure 6.29. The complication arises because the increment may actually leak off the edge of the operand, in which case the operation would draw its source values from random memory addresses. To combat this problem, checks and adjustments are incorporated into the loop declaration (Figure 6.30). Essentially, these checks guarantee that the incremental region begins and ends inside the operand, and that the incremental size is appropriately adjusted if the incremental region must be truncated on either side.

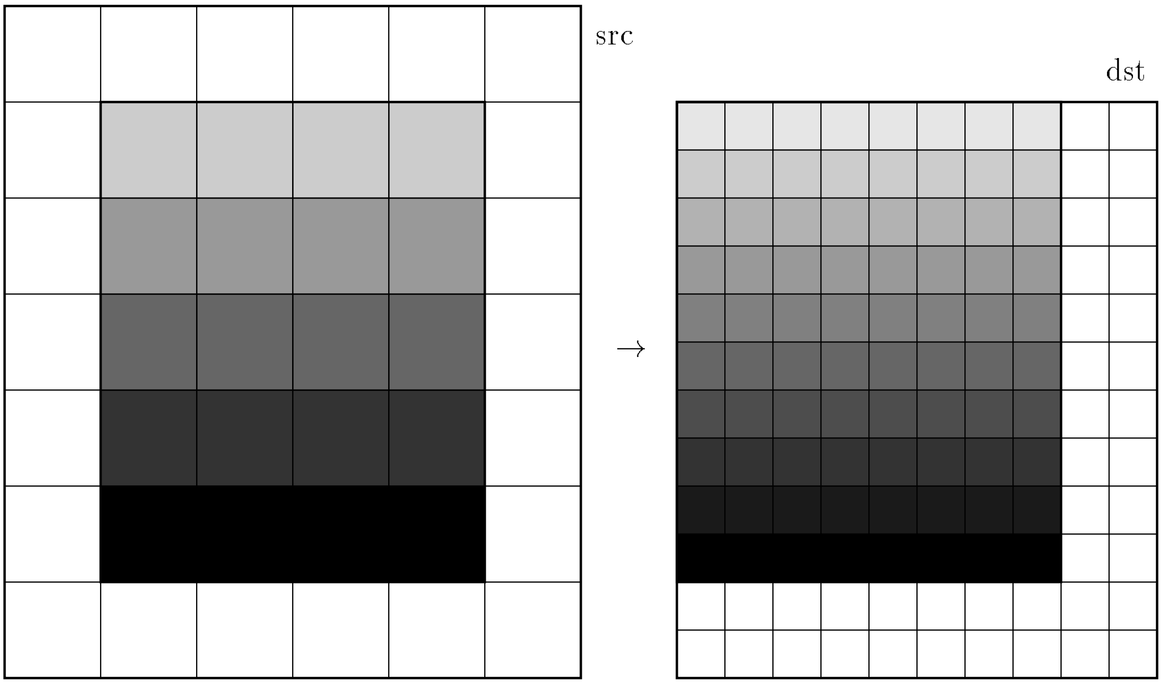


Figure 6.31: Injection interpolation

6.2.4.6 Injection Operations

Injection operations obtain a finer result from coarser input. A common example of an injection operation is interpolation from a coarse to a fine grid (Figure 6.31). Injection operations take a set of offset arguments for each operand, as well as a refinement factor, and an origin point for each grid. The origins are employed to determine which coarse cell corresponds to each fine cell, though this issue typically arises only in the case of interpolations.

The special case that injection operations address separately from the general case is the case of a refinement factor whose entries are all one. In this case, the corresponding offset operation is applied.

Unary and interpolation injection operations have a single form, which corresponds to a coarse source operand and a fine destination operand. Binary injection operations, on the other hand, come in three forms, depending on which of the source operands is coarse: the first source operand, the second, or both. For example, consider the standard correction algorithm. During correction, the postcorrection operand, which has the finer resolution, is subtracted from the coarser precorrection operand, and the result goes into the postcorrection operand (Figure 6.32).

6.2.4.7 Projection Operations

Projection operations transfer information from finer to coarser operands, typically by reduction of the values on the fine loci that overlay a coarse locus. A common example of a projection operation is the determination of whether any of the flags in a set of fine loci are set, and mapping the result to the associated coarse locus (depicted in Figure 6.33, with code fragment in Figure 6.34), such as would be used in projecting the selection flags of a fine grid onto its coarser parent.

Projection operations take a set of offset arguments for each operand, except for the staggering, which is the same for all. The staggering determines which axes to reduce along; specifically, reduction only occurs along axes that are centered, rather than on the mesh lines.

The special case that projection operations addresses separately from the general case is the case of a refinement factor whose components are all one, in which case the operation

```
Void RealField_injection_from_first_subtract (
         RealField dstrealfield,
         RealField srcrealfield1, RealField srcrealfield2,
         AxisSet axisset,
         Staggering dststaggering,
         Staggering srcstaggering1, Staggering srcstaggering2,
         IntegerVector dstaxisloci,
         IntegerVector srcaxisloci1, IntegerVector srcaxisloci2,
         IntegerVector size,
         IndexPoint dststart,
         IndexPoint srcstart1, IndexPoint srcstart2,
         IndexPoint dstorigin,
         IndexPoint srcorigin1, IndexPoint srcorigin2,
         IntegerVector refinement_factor, Integer axes)
{ ...
  switch (axes) {
    ...
    case 2:
      dstart1 = dststart[0]; s1start1 = srcstart1[0];
      dorigin1 = dstorigin[0]; s1origin1 = srcorigin1[0];
      size1 = size[0]; ref1 = refinement_factor[0];
      dal1 = dstaxisloci[0]; s1al1 = srcaxisloci1[0];
      dstart2 = dststart[1]; s1start2 = srcstart1[1];
      dorigin2 = dstorigin[1]; s1origin2 = srcorigin1[1];
      size2 = size[1]; ref2 = refinement_factor[1];
      for (a2 = 0; a2 < size2; a2++)
        for (a1 = 0; a1 < size1; a1++)
          dstrealfield[a1+dstart1 + dal1*(a2+dstart2)] =
            srcrealfield1[a1/ref1+s1start1 + s1al1*(a2/ref2+s1start2)] -
            dstrealfield[a1+dstart1 + dal1*(a2+dstart2)];
      break;
    ...
  }
}
```

Figure 6.32: Injection subtraction with coarse minuend

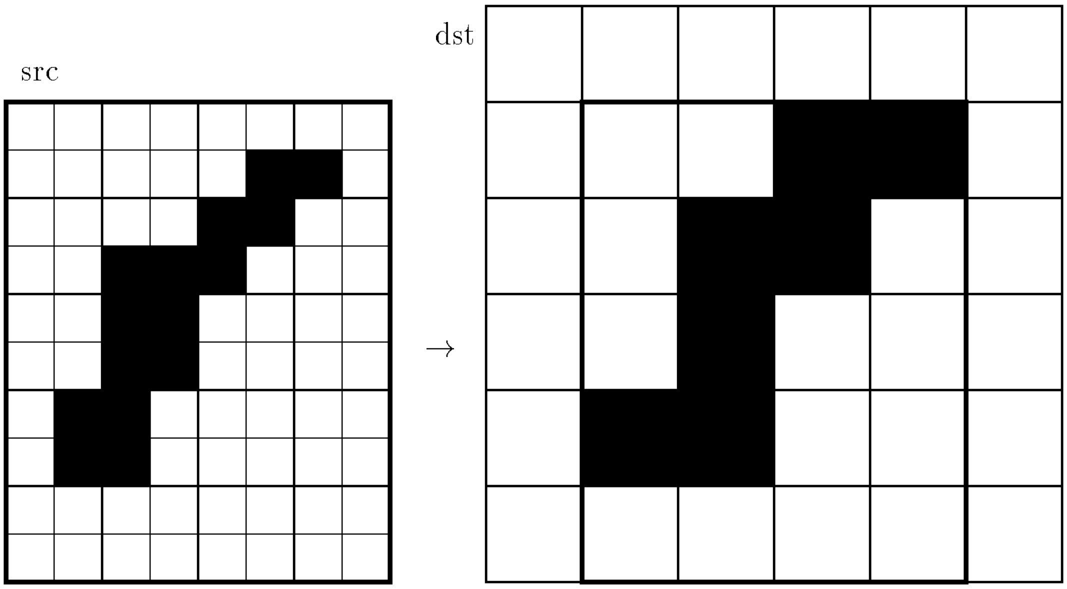


Figure 6.33: Projection *any true*

is reduced to a related offset operation; for example, the projection "any true" operation is simplified to an offset copy, because it projects only a single "fine" boolean onto the coarse boolean.

6.2.5 Summary

The HAMR function library is large and extensive, covering a great many cases for each operation. While a typical application may utilize many of the functions in the library, the overwhelming majority go unused. However, predicting the specific computational needs of a particular application *a priori* can be difficult, so the design decision that most obviously addresses the wide range of potential requirements incorporates as many combinations of circumstances as possible. As a result, the design and implementation of the library required

```
Void BooleanField_projection_any_true (
         BooleanField dstbooleanfield, BooleanField srcbooleanfield,
         AxisSet axisset, Staggering staggering,
         IntegerVector dstaxisloci, IntegerVector srcaxisloci,
         IntegerVector size, IndexPoint dststart, IndexPoint srcstart,
         IndexPoint dstorigin, IndexPoint srcorigin,
         IntegerVector refinement_factor, Integer axes)
{ ...
  switch (axes) {
    ...
    case 2:
      dstart1 = dststart[0]; sstart1 = srcstart[0];
      dorigin1 = dstorigin[0]; sorigin1 = srcorigin[0];
      size1 = size[0]; ref1 = refinement_factor[0];
      reflen1 = Staggering_axis_offset_from_center(staggering,0,axes) ?
                  ref1 : 1;
      dal1 = dstaxisloci[0]; sal1 = srcaxisloci[0];
      dstart2 = dststart[1]; sstart2 = srcstart[1];
      dorigin2 = dstorigin[1]; sorigin2 = srcorigin[1];
      size2 = size[1]; ref2 = refinement_factor[1];
      reflen2 = Staggering_axis_offset_from_center(staggering,1,axes) ?
                  ref2 : 1;
      for (r2 = 0; r2 < reflen2; r2++)
        for (r1 = 0; r1 < reflen1; r1++)
          for (a2 = 0; a2 < size2; a2++)
            for (a1 = 0; a1 < size1; a1++)
              dstbooleanfield[a1+dstart1 + dal1*(a2+dstart2)] =
                dstbooleanfield[a1+dstart1 + dal1*(a2+dstart2)] ||
                srcbooleanfield
                    [a1*ref1+sstart1+r1 + sal1*(a2*ref2+sstart2+r2)];
    ...
  }
}
```

Figure 6.34: Projection *any true* code

more than a year of programming and testing effort, an effort which, at the time, yielded no tangible results, since the library itself had relatively little value outside the context of an AMR system.

However, this task proved well-chosen. Not only did the flexibility of the library ease the programming burden of the rest of the system, it also simplified extending the library when unanticipated needs arose. For example, the need for striding operations was not clear until implementation of the AMR algorithms. At that time, however, the intrinsic properties of the library's design considerably simplified the process of incorporating the striding operations, which included all operation categories. Thus this new set of operations was designed, implemented and tested in a single week.

Ultimately, however, the primary advantage of this library design paradigm is in each function's encapsulation of a wide range of potential circumstances. This approach simplifies the construction of general-purpose algorithms, and thus is ideal for a system like HAMR.

6.3 HAMR Autonomous Grid Hierarchy Management

HAMR provides autonomous data management by the means described in Chapter 5, with sufficient functionality to create, copy and delete data structures, and to execute methods on the structures. While the description of the data management paradigm in Chapter 5 was top down, progressing from the gross structure of the data to the finer details, this description will be bottom up, beginning with the declaration that the application scientist writes, the parser which converts the user-generated declaration to machine-readable form, the declaration data structure, the specification, the HAMR data structure, data item macros, and predefined data items.

6.3.1 HAMR Declaration

To implement the declaration concepts described in Section 5.4, HAMR provides a simple declaration language, and a parser to convert the information into a form accessible by the specification. The overall declaration is composed of a set of module declarations, each of which declares the attributes of the module, as well as all of the data items associated with the module.

The declaration language itself is arbitrary; that is, its approach to syntax is one of many possibilities. In this case, the syntax is a combination of C and Pascal conventions. However, what matters in this case is not the particulars of the syntax, but rather the data, methods and relationships the declaration language can express.

6.3.1.1 Module Header Declarations

Consider the standard controller, shown in Figure 6.35. The module declaration begins with a keyword, `Controller`, which indicates the module type, followed by the name of the module, `standard controller`. The parentheses after the module name indicate that it corresponds to a function, whose name is obtained by concatenating the stratum associated with the module type — which in the case of a control algorithm is the hierarchy — with

```
Controller standard controller();

Integer root timesteps:
  Hierarchy, Initialize root timesteps initialize method;
VoidMethod root timesteps initialize method:
  Fixed, Archetype root timesteps, Value root timesteps initialize;

Void root timesteps initialize(): Hierarchy.
```

Figure 6.35: Module declaration

```
Void Hierarchy_standard_controller (Hierarchy hier)
{ Index t;
  for (t = 0; t < Hierarchy_root_timesteps(hier); t++)
    Level_integrator_method_execute(Hierarchy_level(hier)[ROOT]); }
```

Figure 6.36: Standard controller function (simplified)

the module name. Thus, `Hierarchy_standard_controller` (Figure 6.36) is the function associated with the module `standard controller`.

Module declarations can take on two other forms. First, the module can have no corresponding function (Figure 6.37). This case is identical to the previous case, except that the function associated with the module is empty — or more accurately, is a pointer to the

```
Selector slope selector;

Real epsilon:   Hierarchy, Value 0.001;
Real threshold: Hierarchy, Value 0.200;

Void slope select(): Grid.
```

Figure 6.37: Module declaration with no corresponding function

```
Generic Cluster cluster;

IndexRegionList cluster: Level, Temporary, Axes Space, Elements clusters;
Integer clusters: Level, Value 0;
Real minimum cluster efficiency:
  Level, Initialize minimum cluster efficiency initialize method;
Integer minimum cluster cells along any axis:
  Fixed, Initialize minimum cluster cells along any axis initialize method;
Integer maximum cluster cells along any axis:
  Level, Initialize maximum cluster cells along any axis initialize method;
Integer maximum cluster cells:
  Level, Initialize maximum cluster cells initialize method;
```

Figure 6.38: Generic cluster declaration

```
Solver CMHOG Euler solve():
  CoordinateSystem CARTESIAN1D CARTESIAN2D CARTESIAN3D, Mesh ISOTROPIC;
```

Figure 6.39: Solver declaration on isotropic Cartesian meshes

empty function `Void_do_nothing`. The other form that a module declaration can take is a *generic* module, which is a module that is always active and that has no associated function. For example, every clustering algorithm requires certain parameters, such as the clustering efficiency (Figure 6.38). Since these parameters are common, they can be encapsulated in a single module, which is always active, and so the module's data items are available to all clustering algorithms.

In addition to module type, name and function attributes, non-generic modules can have attributes that determine whether they are active. Specifically, a module can be tied to a particular subset of ranks, coordinate systems and mesh types (Figure 6.39). If the rank, coordinate system and mesh type are not among those declared, then the module is automatically deactivated. For example, a solver may be applicable only to isotropic Cartesian meshes in one and two dimensions, so if the mesh has polar coordinates, the solver is automatically deactivated. If no such attributes are declared, then by default the module is active for all cases.

6.3.1.2 Data Item Declarations

Subsequent to the declaration of the module header, the module contains declarations for the individual data items. The module in Figure 6.35 has two data items: `root timesteps`, an integer scalar that is encapsulated by the hierarchy and that describes the number of timesteps that the control algorithm will perform at the root level, and `root timesteps initialize method`, which is a method that initializes the value of `root timesteps`:

```
VoidMethod root timesteps initialize method:
  Fixed, Archetype root timesteps, Value root timesteps initialize;
```

Finally, the last declaration in the module is

```
Void root timesteps initialize(): Hierarchy.
```

This declaration is a function prototype rather than a data item. It indicates that the function value of the method `root timesteps initialize method` operates on the hierarchy; that is, the function associated with the value `root timesteps initialize` is

```
Hierarchy_root_timesteps_initialize
```

This example illustrates an important point: the stratum on which a method is stored is not necessarily the stratum on which its function values operate. Here, the method — that is, the data item containing the function pointer — is stored on the fixed data structure, but the function pointed to by the method operates on the hierarchy. A more potent example of this principle is the case of an injection routine, which is the same for all grids on the same level, and is therefore stored on the level data structure, but which is applied to individual grids (Figure 6.40).

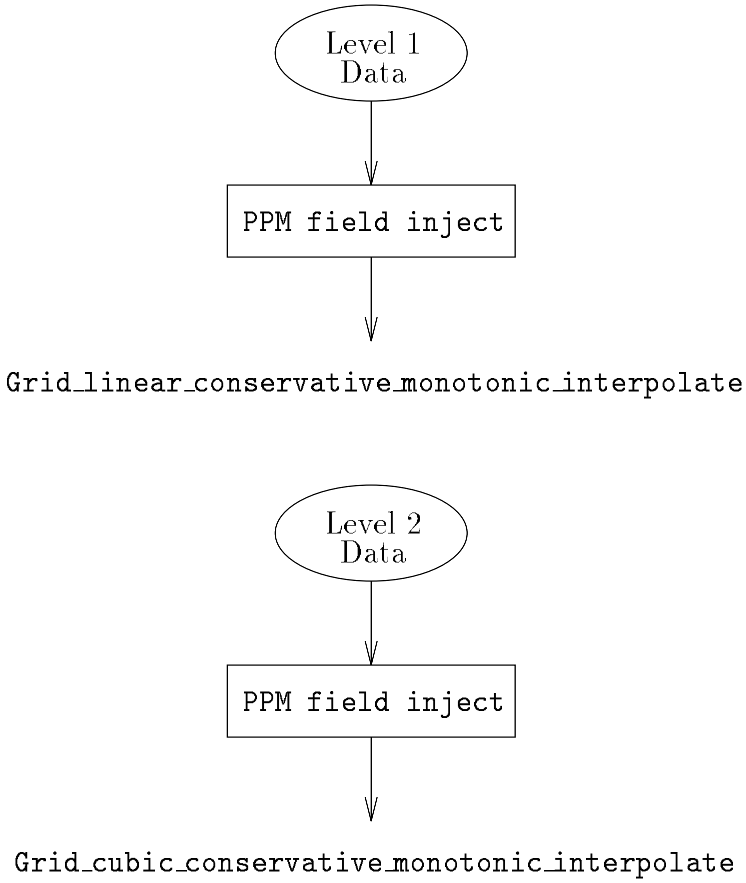
Data item declarations generally take on a simple form:

<*data-type*> <*data-item-name*>: <*stratum*>, <*permanence*>, <*attribute-list*>;

Here, a <*data-type*> is any of the valid data types described in Section 6.1, <*data-item-name*> is an identifier, the <*stratum*> is the one that encapsulates the data item, and the <*attribute-list*> depends on the data type.

6.3.1.3 Structured Data Declarations

Structured data item declarations (Figure 6.41) have attribute lists that explicitly identify their structural attributes. For example, an `ArrayListBox` declaration includes the number of arrays, the number of lists per array and the number of elements per list. The one exception to this rule arises in the case that the structured data type has dimensions of varying sizes — that is, in the case of `Array`, `ArrayList` and `ArrayArray` types. For these types, the




```
VoidMethod PPM field inject: Level,
  Value linear conservative monotonic interpolate,
  Value cubic  conservative monotonic interpolate;
Void linear conservative monotonic interpolate(): Grid;
Void cubic  conservative monotonic interpolate(): Grid;
```

Figure 6.40: Injection declaration

```
Integer grids: Level;
Integer time levels:
  Hierarchy, Value 2;
RealList relative time:
  Level, Permanent, Elements time levels;
IntegerList parents per grid:
  Level, Permanent, Elements grids;
IntegerArray boundary regions per parent:
  Level, Permanent, Elements parents per grid;
RealArrayList boundary region physical domain:
  Level, Permanent, Elements boundary regions per parent;
IndexRegionArrayBox boundary in exterior:
  Grid, Enduring, Lists axes, Elements extrema;
IndexRegionArrayListBox grid boundary in domain:
  Level, Temporary, Arrays grids, Lists axes, Elements extrema;
```

Figure 6.41: Structured data item declarations

only structural attribute that must be explicitly declared is the number of elements, because that attribute will itself be a reference to a structured data item, and so it will inherit its referent's less complex structural attributes. For example, an `ArrayArray` will have as its `Elements` attribute a reference to an `IntegerArrayList`, which in turn has as its `Elements` attribute a reference to an `IntegerArray`, and so on. The `Elements` attribute of the `IntegerArrayList` is inherited by the `ArrayArray` as its `Lists` attribute, the `Elements` attribute of the `IntegerArray` is inherited by the `IntegerArrayList` as its `Lists` attribute and therefore by the `ArrayArray` as its `Arrays` attribute, and so on.

6.3.1.4 Dimensional Data Declarations

Dimensional data declarations (Figure 6.42) have attributes that describe the structural

```
RealAxisDescription node position:
  Level, Permanent, AxisStaggering Node, Stencil mesh stencil;
RealAxisContribution contribution: Injection,
  Increment linear increment, Contributors linear contributors;
```

Figure 6.42: Dimensional data item declarations

properties along each axis. In the case of `AxisDescription` declarations, one of the attributes, the `Stencil`, is optional: if there is no attribute declared, then the stencil is empty by default, which means that the length along each axis will be the size of the interior of the computational domain of the encapsulating level. The `AxisContribution` case, however, is a bit more complex. In the first place, all `AxisContribution` data items are encapsulated in levels, and all are permanent, so declaring the stratum and permanence would be superfluous. Also, `AxisContribution` data items have some unique attributes, namely the relationship between the levels that the interpolation is performed on (that is, injection or projection), the increment of the source loci surrounding the destination locus, and the number of contributors, which is to say the number of functions on the solution at the loci that contribute to the interpolated value.

6.3.1.5 Spatial Data Declarations

Spatial declarations (Figure 6.43) have by far the greatest number of attributes. These attributes include not only stratum and permanence, but also structural attributes such as axis set and staggering, variform structural attributes such as stencil and relative time list, set information such as member names, and functional attributes, both data and methods.

```
Real SpacetimeVariableSet PPM field: Grid, Permanent,
  AxisSet AxisSet_all_axes(Space),
  Staggering Staggering_cell_center(Space),
  Stencil PPM field space stencil, Time PPM field time,
  Member density, Member energy, Member u, Member v, Member w,
  Precorrection PPM field parent flux correction,
  Postcorrection PPM field filial flux correction sum,
  Selection PPM field selected,
  Initialize PPM field initialize method,
  Inject PPM field inject method,
  Project PPM field project method,
  Extrapolate PPM field extrapolate method,
  Correct standard correct method,
  Select PPM field select method;
```

Figure 6.43: Spatial data item declaration

Only the first four attributes (or the rank attribute, for `Maximal` sets), are required; all of the other attributes are optional. (In principle, only first two attributes are absolutely necessary, since defaults could be chosen for the rank, axis set and staggering attributes.) For example, if no stencil is declared, then the size of the data item is the size of the computational interior of the grid; in other words, the data item has no ghost boundary. If no member names are declared, then the set has one member whose name is the same as the name of the data item. Alternatively, the set could have a `Members` attribute, with an integer value that determines the number of members of the set; this approach is especially useful for declaring a set of multipurpose temporary variables. Finally, any missing functional data attributes correspond to null pointers, and any missing functional method attributes correspond to empty functions.

```
VoidMethod PPM field inject method:
  Level, Archetype PPM field,
  Value CMHOG inject from coarser;
Void CMHOG inject from coarser():
  Grid;
VoidSetMethod PPM field extrapolate method:
  Level, Archetype PPM field,
  Member u, Value standard member reflected boundary axis1 velocity negate,
  Member v, Value standard member reflected boundary axis2 velocity negate,
  Member w, Value standard member reflected boundary axis3 velocity negate;
Void standard member reflected boundary axis1 velocity negate(): Grid;
Void standard member reflected boundary axis2 velocity negate(): Grid;
Void standard member reflected boundary axis3 velocity negate(): Grid;
```

Figure 6.44: Method data item declarations

6.3.1.6 Method Declarations

Method declarations (Figure 6.44) typically require at least one attribute other than stratum, namely the value(s). In the case of a `Method`, this attribute is the only required attribute, but for a `SetMethod`, an archetype attribute is also required, as well as member attributes, whose purpose is to map the members of the method to the members of the archetypal spatial set.

The number of method value attributes is arbitrary, because a method can refer to any number of different possible functions, as shown in Figure 6.40. In the case of a `SetMethod` declaration, the order of the attributes indicates which method values are associated with each member; that is, the `Value` attributes following each `Member` attribute indicate the function values associated with that member.

6.3.2 HAMR Declaration Parser

The HAMR declaration language is implemented by a parser, which is a preprocessor that converts the declaration into a form that the specification management software can use.

The actual parsing — that is, the conversion from the declaration language to an internal representation — is fairly trivial: aside from the data item headers themselves and the stratum and permanence declarations, the attributes typically have the form

$$<keyword> <value>$$

a form which lends itself to fairly straightforward parsing.

The parsing step is implemented by a loop containing a simple *switch* statement. When a module is parsed, its module declaration is stored, and after each data item declaration (including attributes) is parsed, the item's name and attribute values are stored in a queue of data items. After all the data items in the module are parsed, the queue is converted into a static module representation, which is then inserted into a queue of modules. When all the modules have been parsed, this queue is converted into a static list of data items, each tagged with its module's identifier.

Within a declaration, identifiers such as data item names can be expressed in either of two forms: as delimited strings or without delimiters. In the former case, the identifier can be a collection of any (non-quote) characters delimited by double quotes, including characters that play a specific role in the declaration syntax, such as parentheses, colons, commas, semicolons and periods. In the latter case, the identifier can contain any characters except

Declaration:	`"This is the name of a (silly) data item:"`
Name:	`This is the name of a (silly) data item:`
Token:	`This_is_the_name_of_a_silly_data_item`
Comparison:	`THISISTHENAMEOFASILLYDATAITEM`

Figure 6.45: Identifier representations

the special characters listed, and the identifier is terminated by the colon.

Once the complete set of data items has been obtained, implicit information can be derived. First, alternative forms of each data item and member name are obtained. The parser represents such identifiers in three ways: in their original form as laid out in the declaration, as C-like identifiers consisting of alphanumeric characters and underscores, and as alphanumeric strings, all upper case, used for comparisons (Figure 6.45). Multiple copies of the latter two forms are generated, one based on just the data item name and one that concatenates the module name and the data item name, in order to disambiguate in the event that there are multiple data items with the same name — for example, in the case of the referents of variform attributes — or multiple sets with the same member names. (In the case of member names, three versions are generated: the original, the concatenation of the data item name and the member name, and the concatenation of the module name, data item name and member name.) Next, all attribute references are resolved, by comparing the value of each attribute to the names of all data items or members of the appropriate types. (An unresolvable reference causes the parser to terminate with an error report.) Third, reciprocal attributes are determined: for each reciprocal attribute of a data item, all data items of the appropriate types are examined to determine whether the reciprocated attribute refers to the reciprocating data item. Finally, the results of the parse are output. All of the results of the parse take the form of C macros, forming several categories:

- number of data items;
- data item indices;
- attribute macros for attributes that are based on the rank of the domain;
- macros associating the data items with fields of appropriate data structure;
- declaration lists.

The first three categories are straightforward; for example,

```
#define GRID_BOOLEAN_SPACEPARAMETERSETS   3
#define OVERALL_SELECTION_INDEX           0
#define Grid_Boolean_SpaceParameterSet_staggering(i,rank) \
        ... (i == OVERALL_SELECTION_INDEX) ? Staggering_center(rank) : ...
```

The field macros provide a clean programming interface to data items, but they are a bit more complicated; for example

```
#define Grid_overall_selection(grid) \
        Grid_Boolean_SpaceParameterSet(grid)[OVERALL_SELECTION_INDEX]
```

Finally, the declaration lists provide the link from the declaration to the specification, since these lists contain the values of the various attributes of the data items. For example, consider the list of names of `BooleanSpaceParameterSet` data items. This list might be defined as

```
#define GRID_BOOLEAN_SPACEPARAMETERSET_NAME \
        { "overall selection", "PPM field selection", "gravity selection" }
```

Other lists can be similarly defined, not only lists of scalars and strings but also lists of function pointers:

```
#define FIXED_VOID_METHOD_VALUE \
        { Grid_item1_initialize, Grid_item2_initialize, ... }
```

6.3.3 HAMR Declaration Data Structure

The declaration lists produced by the parser provide a powerful means of supplying the declaration information to the specification, using a feature of the C language that is rarely exploited to its full potential. Specifically, the C language allows dimensioned arrays of scalars, pointers or strings to be initialized with predefined lists.

For example, consider the name list macro defined in Section 6.3.2. This list can be used as the initialization list for a statically dimensioned list of strings:

```
static String Grid_Boolean_SpaceParameterSet_name[] =
  GRID_BOOLEAN_SPACEPARAMETERSET_NAME;
```

which the C preprocessor converts to

```
static String Grid_Boolean_SpaceParameterSet_name[] =
  { "overall selection", "PPM field selection", "gravity selection" };
```

When the C compiler parses this declaration, it constructs a static array of the appropriate size in memory, which contains the appropriate values.

In this manner, the values of all of the declaration lists are automatically declared as static arrays that are global with respect to the object file that contains them, and hidden from the functions in all other object files. In fact, preprocessor directives guarantee the declaration of only the minimal collection of such static arrays:

```
#ifdef GRID_BOOLEAN_SPACEPARAMETERSET_NAME
static String Grid_Boolean_SpaceParameterSet_name[] =
  GRID_BOOLEAN_SPACEPARAMETERSET_NAME;
#endif  /* #ifdef GRID_BOOLEAN_SPACEPARAMETERSET_NAME */
```

An important advantage of this approach is that function pointers can be initialized in this manner, just as if they were scalars:

```
static VoidMethod Fixed_Void_Method_value[] =
  { Grid_shock_tube_initialize, Grid_comet_initialize };
```

To the parser, these function names are simply strings constructed while parsing the declaration; to the compiler, they refer to specific, existing functions, whose addresses are loaded into this convenient predimensioned list.

This arrangement also illustrates the reason that the parsing algorithm must be separate from HAMR itself: if the parser were activated at run time rather than before compile time, it would be unable to convert the strings in method declarations into actual function pointers, because there would be no platform-independent means of determining the function that matched the string — and even a platform-dependent approach would be likely to be unacceptably cumbersome. If the parser is a preprocessing step, however, this determination is not required, because the parser considers all of these results to be strings to be output, and it is not in a position to know that the output it creates will be used as macro definitions, from which the compiler will construct the static lists. Thus, the C preprocessor and the compiler combine to perform the task of determining the method values' function pointers, a task that would be at best extremely difficult to code explicitly, but that is trivial when addressed in this manner. In fact, this kind of software engineering approach drove much of the development of HAMR, because one means of achieving portability is to take advantage of a small number of either commonly-available or homegrown filters.

To make the declaration lists available outside the object file in which they are defined, HAMR provides a declaration data structure (Figure 6.46), which is essentially a collection of pointers that are set to point to the appropriate static list (Figure 6.47), or are null in the case that no appropriate list exists. The routine that sets the pointers is in the same object file as the lists themselves, and so the lists need not be accessed by any other routine. In fact, the object file that contains the static lists contains only the functions to set the declaration structure's pointers to the appropriate static lists.

6.3.4 HAMR Specification

The process of constructing the HAMR specification (Figure 6.48) is divided into several steps, which are variously performed by:

- the application designer;
- special-purpose preprocessors;

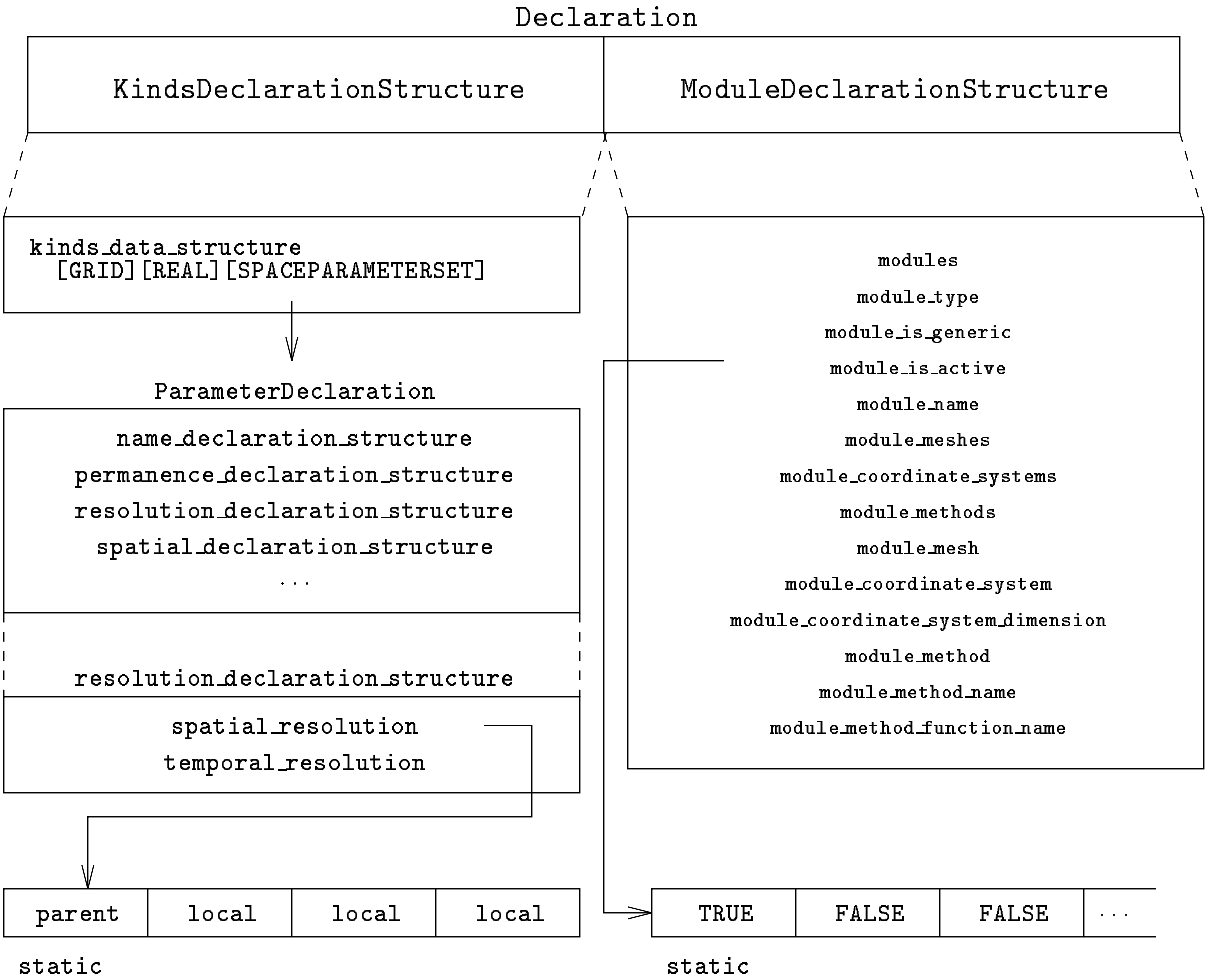


Figure 6.46: Declaration structure

```
#ifdef GRID_BOOLEAN_SPACEPARAMETERSETS
#  if TypeParameterType_has_name(BOOLEAN,SPACEPARAMETERSET)
        if (Declaration_parameter_declaration(
                  decl,GRID,BOOLEAN,SPACEPARAMETERSET) !=
            (ParameterDeclaration)NULL) {
#         ifdef GRID_BOOLEAN_SPACEPARAMETERSET_NAME
            Declaration_name(decl,GRID,BOOLEAN,SPACEPARAMETERSET) =
              (StringList)SystemLevel_Type_ParameterType_name;
#         endif /* #ifdef GRID_BOOLEAN_SPACEPARAMETERSET_NAME */
        } /* if Declaration_parameter_declaration */
#  endif /* #if TypeParameterType_has_name */
#endif  /* #ifdef GRID_BOOLEAN_SPACEPARAMETERSETS */
```

Figure 6.47: Setting a pointer on the declaration structure

- the standard C preprocessor;
- the C compiler;
- the resulting executables that the compiler produces.

The first two steps, that of designing the declaration and parsing it, are performed as described in Sections 6.3.1 and 6.3.2, by the application designer and by the parser, respectively; the result is several header files containing the set of macro definitions that describe the declarations. In addition, the functions that implement both the standard AMR algorithms and the application algorithms are passed through a filter that extracts C function headers and converts them to prototypes, producing additional header files. Next, the source code for the functions that build the declaration data structure and assign pointers to the static declaration lists is compiled, as is the associated set of functions for building the specification from the declaration data structure.

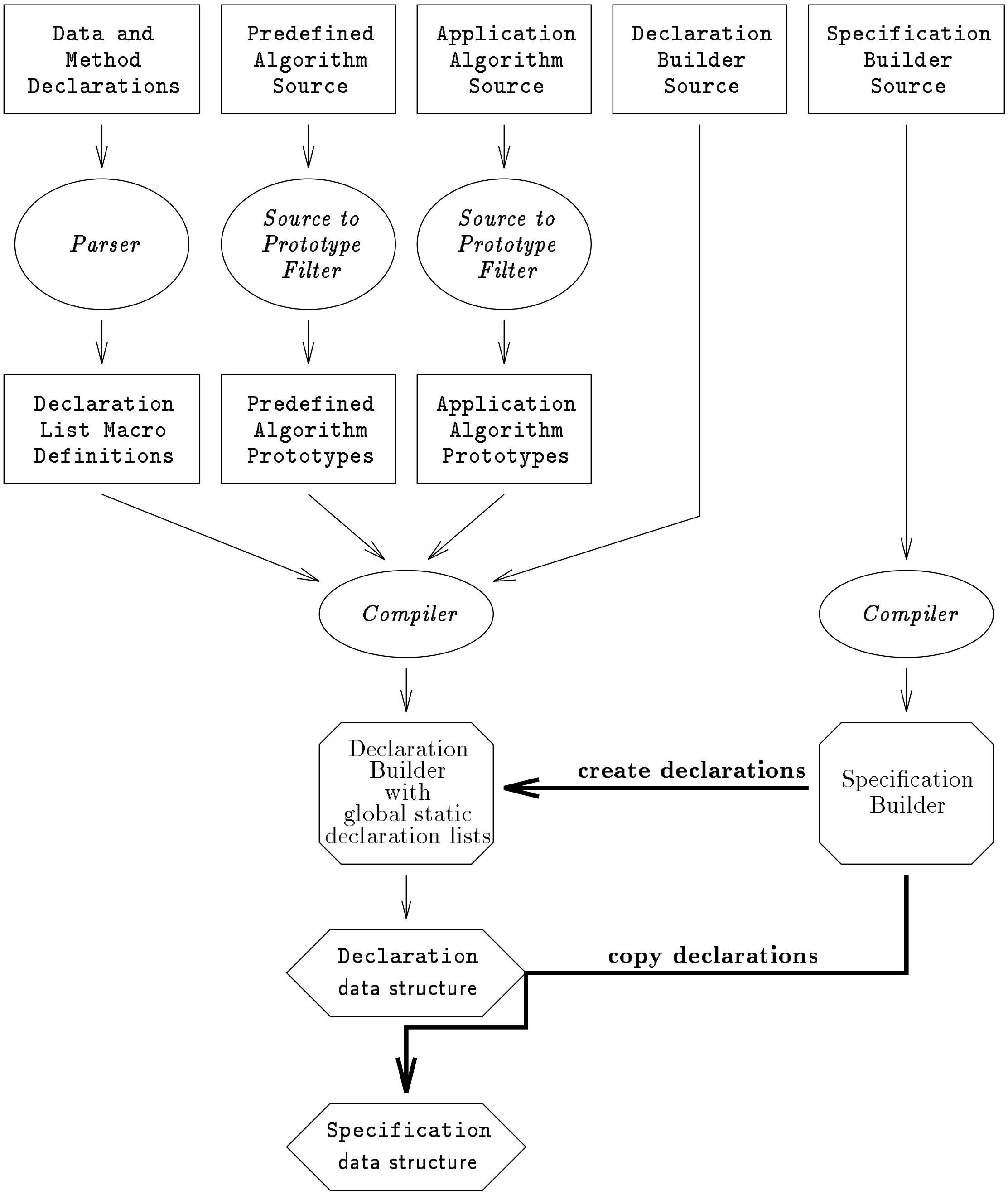


Figure 6.48: Building a specification

When a HAMR application is run, its first task is to build the specification, which is achieved by building the declaration data structure, and then copying each of the static declaration lists into a dynamically allocated instance of an appropriate data type, which not only contains all of the appropriate information, but also provides conveniently indexed access to its values (Figure 6.49). Thus, the HAMR specification has essentially the same design as the declaration structure, but instead of mere pointers to existing static lists, the specification contains its own copies of the lists, which need not be lists themselves; that is, they can be more complex structured types such as `Arrays` and `ArrayLists`. However, HAMR structured types are allocated contiguously, so the values of the specification arrays can be directly copied from the static lists that the declaration structure points to, simply by dereferencing the more complex structures down to contiguous lists, in much the same manner as operations performed on complex structured types. Once the specification has been built via these copies, the declaration data structure can be discarded, and the static declaration lists, which consume a trivial amount of memory even for the most complicated sets of declarations, are thereafter ignored.

The specification is divided into two main sections, one which is indexed by module and one that is indexed by data item. It also contains a few additional attributes that describe geometric properties of the overall domain: the rank, mesh type and coordinate system.

The section that is indexed by module includes the number of modules, and the following module attributes for each module:

- the module type;

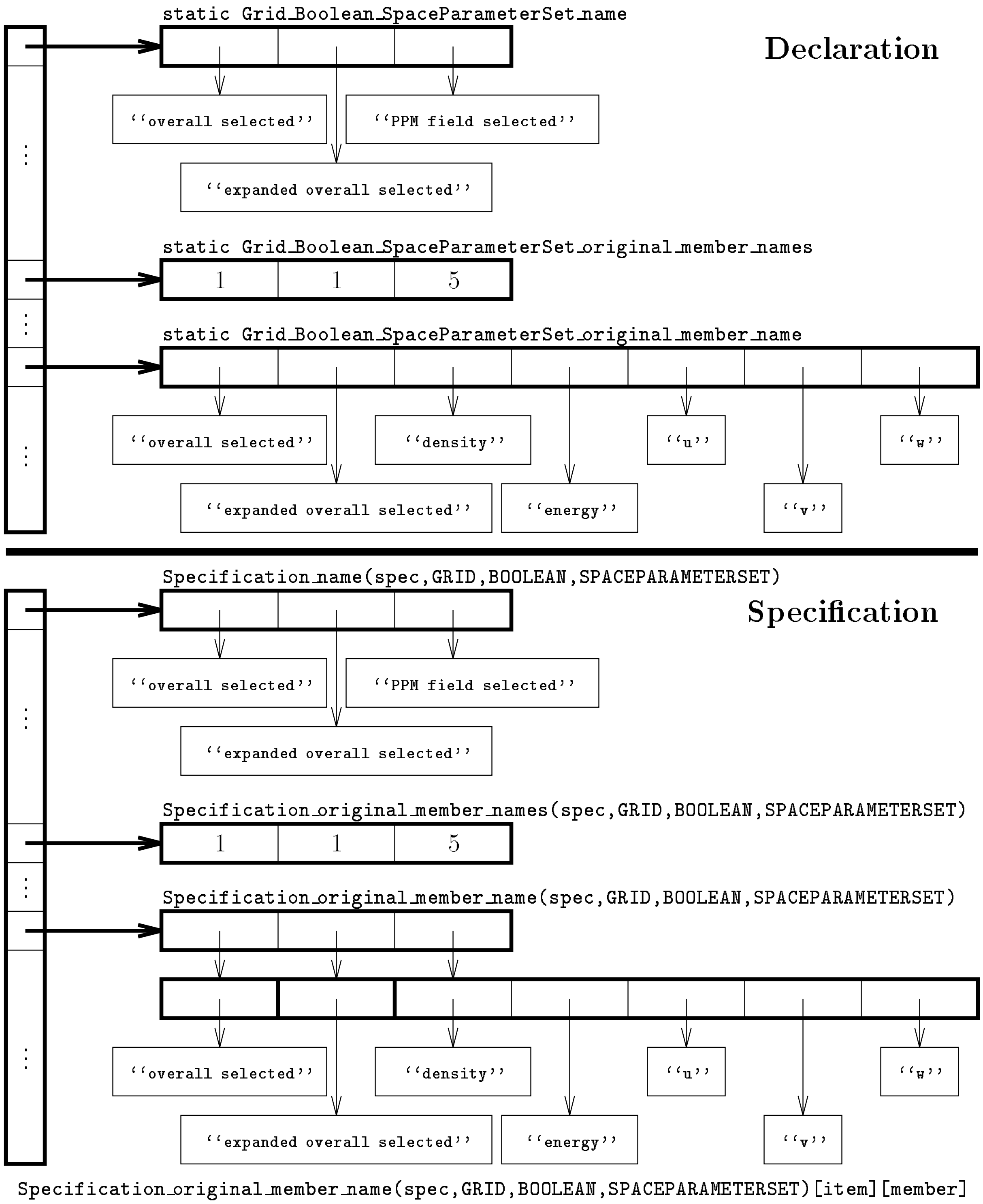


Figure 6.49: Declaration and specification versions of set member information

- the module name;
- the module function pointer, and the name of the module function;
- flags for whether the module is generic and whether it is active;
- the list of meshes under which the module is active, and the number of such meshes;
- the list of coordinate systems under which the module is active, and the number of such coordinate systems;
- the list of ranks of the coordinate systems under which the module is active.

Essentially, these attributes simply identify the module and the circumstances under which it is active.

As for the section indexed by data item, it contains considerably more attributes, including:

- the data item name;
- the module that contains the data item;
- the permanence;
- the spatial and temporal resolution;
- the list of initial or potential values the data item or its members can take on, and the number of such values;

- spatial attributes such as axis set and staggering;
- information about the names of set members;
- the referential attributes of the data item, including the structural attributes of structured types, the set archetype, and functional data and method attributes;
- the variform referential attributes, such as stencil and relative time list;
- the reciprocal attributes for all of the referential attributes, both uniform and variform.

Referential attribute information is coded as three enumerated entries — the stratum, the element type and the parameter type — and one index, which indicates the particular data item of the stratiform type. This approach has the advantage that the specification can fully code all references, without having any access to the data structures that contain the data items that the coded references describe, let alone to the data items themselves. Thus, any part of the overall data structure can learn about any of the referential relationships between data items, regardless of whether it can access the referers or referents.

6.3.5 The HAMR Data Structure

The HAMR data structure is an implementation of the data structure described in Section 5.1, comprising all of the parts shown in Figure 5.4. For each stratum, HAMR defines two data structures: the stratum itself, and the structure of stratum data. For example, the two hierarchy data structures are `Hierarchy` and `HierarchyData`. While the stratum structures are quite simple — essentially just a few pointers — the structures of stratum data are far more complicated.

Each structure of stratum data consists of two major parts: a set of pointers to lists of data items, one for each data type, and a structure of attribute appendices (Figure 6.50). Essentially, the structure of stratum data provides a hierarchical collection of *slots* that can be filled according to the needs asserted by the specification. For the data items themselves, the higher level slot is a framework — a list of pointers to instances of the appropriate data type — that is created when the structure itself is created, and whose length is determined by the appropriate entry in the specification. The lower level slots are the individual pointers — that is, the elements of the data type pointer list — that are filled on demand, according to both the extent categories of the data items and the needs of the algorithms that operate on the data structure. In contrast, the attribute appendix structure is an array of pointers to appendices, rather than a set of individual slots, but it is also filled on demand. The advantage of this arrangement of appendices is that it allows all appendix operations to be generic with respect to the data type that the appendix describes, as opposed to the operations on the data items (or their slots) themselves, each of which is specific to an individual data type.

So, on each structure of stratum data, every data type is represented by two pointers, one to a list of slots that hold instances of the type, and one to a list of slots that hold appendices that describe the instances of the type. More precisely, the cost is one pointer for each existing data type, and one pointer for each combination of element type and parameter

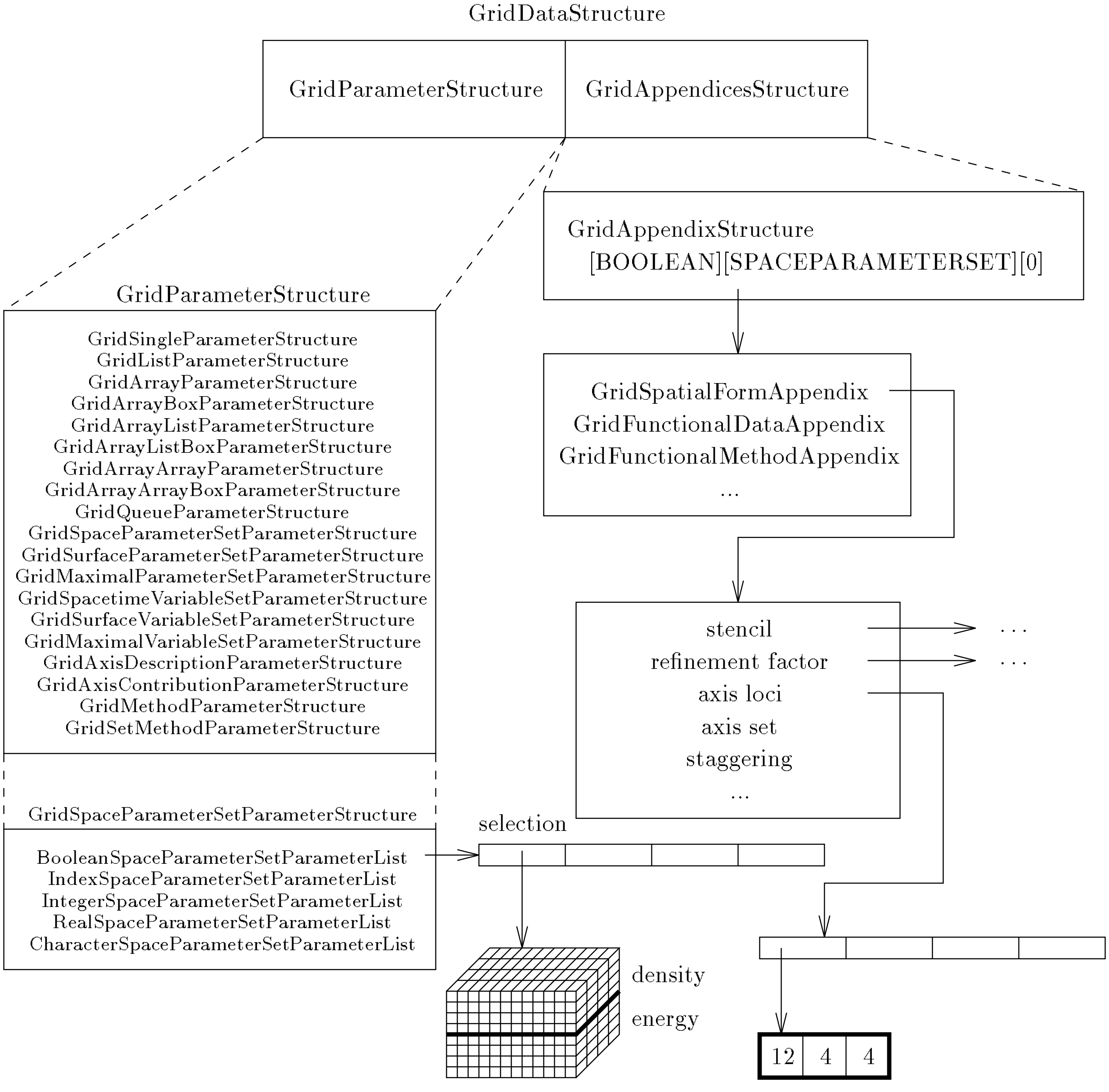


Figure 6.50: Structure of stratum data

type, since the array of appendix pointers includes some combinations that are not types; for example, the `VoidList` type combination is not a valid data type.[7] Aside from a very small number of additional data fields — for example, a structure of level data also has pointers to the immediately coarser and immediately finer structures of level data — a structure of stratum data when first allocated consists only of these two sets of pointers, which are initially null. Additional memory consumption is driven entirely by demand.

While the hierarchical slot arrangement of a data type is quite simple, the structural properties of an appendix are far more complex. This complexity arises because of the desire to minimize the amount of memory consumed by the data structure. Thus, just as a list of pointers to data items is allocated on demand, each appendix structure is allocated on demand, in essentially the same manner.

The appendix structure is itself merely a collection of pointers, specifically to the various categories of attributes that the various data types require. For example, each appendix contains pointers to structures that contain the attributes that describe

- the computational shapes of structured types;
- the computational shapes of spatial types;
- the time information attached to a relative time level list;

[7]There are a total of 164 valid data types: thirteen element types for each of the nine structured parameter types, four element types for `AxisDescriptions`, one element type for `AxisContributions`, five element types for each of the six spatial parameter types, and six element types for each of the two method parameter types. In contrast, there are 266 combinations of element type and parameter type.

- the temporal information of spatiotemporal types;
- functional data references;
- functional method references;
- method value indices.

Encapsulating these pointers within a single structure incurs very little cost — a few unused pointers for each data type — while ensuring that operations on the appendix structure are appropriate for all data types.

This same strategy applies equally well at the bottom level of the appendix structure. For example, the structure that contains the computational shape attributes for spatial types has slots for the stencil, thickness, refinement factor, axis loci, and so on, yet only `Surface` sets require a thickness attribute. Thus, just as with each category of attributes, space for each individual attribute is allocated only if required by the data type to which the appendix belongs.

Each individual attribute is represented by a list, with one entry for each data item of the associated type. Some lists, such as for functional method attributes, contain references — that is, pointers to other data items. Other lists, such as for the staggering, contain scalar attribute values; in the case of the staggering, the attribute is a list of staggering values. Finally, some lists, such as for the axis loci, contain a small structure for each data item.

Creating a structure of stratum data, then, is a straightforward process. First, the base structure itself is allocated, and all its constituent pointers are set to null. Second, for each data type, the appropriate appendix framework is allocated if there are data items of that type, and its values are initialized to null values. Finally, for each data type, the appropriate data item framework is allocated, and all its data item pointers are set to null.

6.3.6 Data Item Macros

Many implementations of Berger's AMR strategy, including HAMR, rely on elaborate chains of pointers to obtain data from a variety of sources. For example, Haupt [Hau95] reports constructs such as

```
grid%gfcn(ESTRE)%tlev(1)%data
```

while Bryan [Bry96c] employs such chains as

```
Temp->GridHierarchyEntry->ParentGrid->GridData->AreSubgridsStatic()
```

HAMR would have this problem as well; in fact, it would have a much more severe form of the problem, since it has much more elaborate data structure definitions. A typical HAMR pointer chain looks like:

```
grid->grid_data->level_data->hierarchy_data->
  hierarchy_data_parameter_structure.
    hierarchy_data_single_parameter_structure.
      Hierarchy_IntegerStencil_Single[0]
```

To avoid this problem, HAMR provides a filter that extracts data members from a C *struct* or *union* definition and converts them into macros. The filter acts as a very simplified type definition parser, examining the definition and determining which identifiers are data types and which are members. From these members, macros can be constructed whose names are the concatenation of the encapsulating data structure type and the member name. For example, a member of a level structure called `finer_level` would have as its macro definition:

```
#define Level_finer_level(level)    ((level)->finer_level)
```

The filter parses not only the data structure definition, but also any `#include` directives in the file. For each `#include` directive that refers to a type definition file — that is, an include of a file whose name is of the form `filename_typedef.h` — the resulting file of macro definitions will also include the associated file of macro definitions; for example,

```
#include <griddata_typedef.h>
```

in the file containing the data structure definition is converted to

```
#include <griddata_macro.h>
```

in the resulting file containing the macros. In addition, on discovering such an `#include` directive, the filter parses the associated macro file, and creates new macros that associate the data structure members associated by the macros in the included file to the data structure in the top level file. Thus, for example, if the included macro file contains a definition for `LevelData_Level_Real_List(leveldata)`, then a definition

```
#define Level_Level_Real_List(l) \
        LevelData_Level_Real_List(Level_level_data(l))
```

will be placed in the macro file that the filter is currently producing. In fact, since the filter applies this approach to all data structure definitions, the filter produces a set of macros that ultimately associates the highest level data structures with structure members on the lowest level data structures; for example,

```
#define Grid_Fixed_Real_Single(g) \
        GridData_Fixed_Real_Single(Grid_grid_data(g))
```

In this example, the macro associates the grid structure to a real scalar encapsulated by the structure of fixed data, because the expansion of the top level macro is itself a macro, whose expansion is a macro, and so on.

In addition, the declaration parser provides another level of macros on top of the macros provided by the filter. These macros associate the name of a data item with the appropriate data structure member, by combining the data structure member macro with the appropriate index into the data structure member, which is of course a list of pointers to data items of the appropriate stratiform type. The set of macros for each scope includes all broader scopes, and therefore the application programmer is not only provided with a clean, intuitive interface to the data, but is also relieved of the burden of being constantly aware of the specific, detailed, hierarchical arrangement of data structures that constitute the overall HAMR data structure system.

6.3.7 Predefined Data Items

One of the primary advantages of this generalized approach to data management is that it allows general purpose data items to be declared, stored and managed in precisely the same manner as application-specific data items. For example, every grid has associated with it an `IndexRegion` called `locus_in_domain`, which indicates the diagonally opposite endpoints of the computational subdomain that the grid subtends. This region is declared as a data item encapsulated by the grid, and the data management system treats it in the same manner as any other data item.

To take full advantage of this capability, HAMR includes a set of predefined modules that describe the standard data items and methods that are available for all applications. These modules include the standard data items that all data structures require, as well as the standard AMR algorithms, and a number of methods that are likely to prove useful to application developers.

Standard data items enscapsulated in the structure of hierarchy data are declared in a generic module called `Hierarchy data`. These data items include:

- flags indicating which axes have periodic exterior interfaces;
- flags indicating which exterior interfaces are reflecting;
- the number of cells along each axis at the coarsest resolution;
- the maximal active stencil;

- the maximum depth allowed;
- a queue of regions indicating the arrangement of root level grids.

The module also includes methods that initialize these various data items.

Standard data items enscapsulated in the structure of level data are declared in a generic module called `Level data`. These data items include:

- the depth of the level;
- the refinement factor with respect to the immediately coarser level;
- the number of cells along each axis of the domain at the level's resolution;
- the buffer region;
- the refinement period;
- a queue of regions that are to be selected;
- a queue of regions that are *not* to be selected.

The module also includes methods that initialize these various data items.

The queues of regions to be selected or not selected are a useful addition to HAMR, because they provide the ability to use static grids, or to declare certain regions as permanently refined. In fact, the use of these queues is slightly more subtle: their values are regions not of computational space but of computational *spacetime*. Thus, for example, a particular region can be highly refined for a period of time, then less refined afterwards, or vice versa.

Standard data items enscapsulated in the structure of grid data are declared in a generic module called `Grid data`. These data items include the locus of the subgrid in the computational domain, and the various lists of relationships between the grid and other grids, including the relationships between its interior and its parents, between its boundary and the parents of its boundary, between its boundary and its siblings, between it and its children, and the set of regions of its boundary that are on the exterior of the computational domain.

An important point about these relationships is that they are static with respect to the lifetime of the grid, except for the relationships to children. The reason the relationships are static is that the arrangement of grids at a particular level is fixed at regridding, and does not change until the next regridding — at which time, the grids in question are discarded, because they are replaced by new grids, which cover the region that the phenomenon of interest has moved into. Thus, while a parent, over the course of its lifetime, will have many different sets of children, a child over its lifetime will have exactly one set of parents. As a result, interactions between levels can be expressed more cleanly, and with more confidence in the immutability of the relationships, if they are expressed as an interaction between a grid and its parents, rather than between a grid and its children. (In some cases the latter is unavoidable, but to the extent that it can be avoided, it should be.)

Various mesh types have data items that are specifically associated with them. For example, isotropic meshes require a coarsest cell size stored on the hierarchy, and a local cell size stored on the level; in contrast, rectilinear meshes require the positions of the nodes and cell centers at the root level resolution, as well as the sizes of the cells, which are stored on the hierarchy, and analogous information at the local resolution stored on each level.

Just as standard data items are declared, so too are the standard AMR algorithms. These include most of the algorithms described in Section 6.4. Typically, the modules for these algorithms look like:

```
Integrator standard integrator().
```

However, a few of them declare data items as well. For example, the standard refiner declares a field of overall selection flags on each grid, onto which are mapped the selection flags from each variable, which are application specific data items. Thus, for example, the clustering algorithm needs to know nothing about the application variables — such as density, energy and velocity — because their selection flags are subsumed by the overall selection field.

6.3.8 Summary

While the HAMR data management system is based on the theoretical underpinnings laid out in Chapter 5, its capabilities extend beyond those precepts to provide tremendous power and flexibility, which are available with minimal effort on the part of the application scientist. This aspect of HAMR makes it a natural way to express and perform complex, sophisticated simulations, even if they do not require adaptive techniques, or even if the adaptive techniques they require do not match Berger's strategy. More importantly, however, they provide an ideal bridge between an application scientist and the AMR algorithms necessary for utilizing Berger's strategy.

6.4 Algorithms for Berger's AMR in HAMR

HAMR implements the standard algorithms that perform the operations required by Berger's AMR strategy. These algorithms include:

- control;
- integration;
- boundary collection;
- extrapolation;
- refinement;
- regridding;
- Richardson truncation error estimation;
- selection;
- clustering;
- correction;
- projection.

However, HAMR does not merely copy the existing AMR algorithms. Instead, it implements versions of them which take full advantage of the expressive power of the HAMR data

```
Void Hierarchy_standard_controller (Hierarchy hier)
{
  if (Hierarchy_root_timesteps(hier) < 1) return;
  for (ts = 0; ts < Hierarchy_root_timesteps(hier); ts++) {
    Level_integrator_method_execute(Hierarchy_level(hier)[0]);
    Hierarchy_outputter_method_execute(hier);
  }
}
```

Figure 6.51: Standard controller

structure and data management infrastructure. Thus, despite the fact that HAMR is based on the work of Berger and collaborators, these algorithms themselves constitute a significant contribution to the AMR canon.

6.4.1 Control Algorithm

The control algorithm provided for HAMR loops over the chosen number of root level integrations, performing two operations for each: the integration itself, and an output operation (Figure 6.51).

6.4.2 Integration

HAMR's standard integration algorithm (Figure 6.52) implements the recursive algorithm described in Section 4.4. The halting condition for the algorithm is that it has reached an empty level, at which point it returns without computing. Otherwise, the algorithm refines if appropriate. Then, it collects boundary values. Next, it determines the timestep interval,

```
Void Level_standard_integrator (Level lev) {
  if (Level_grids(lev) < 1) return;
  if (Level_old_timestep(lev) % Level_refinement_period(lev)) == 0)
    Level_refiner_method_execute(lev);
  Level_standard_boundary_collector(lev); Level_timer_method_execute(lev);
  Level_solver_method_execute(lev);
  if (Level_not_finest_existing(lev) &&
      (Level_grids(Level_finer_level(lev)) > 0))
    for (r = 0; r < Level_refinement_factor_from_finer(lev)[TIME]; r++)
      Level_standard_integrator(Level_finer_level(lev));
  Level_incrementer_method_execute(lev);
  if (Level_not_finest_existing(lev) &&
      (Level_grids(Level_finer_level(lev)) > 0)) {
    Level_corrector_method_execute(lev);
    Level_projector_method_execute(Level_finer_level(lev));
  }
}
```

Figure 6.52: Standard integration

an operation that for many applications is empty; however, some applications constrain the time interval not to overtake the minimum sound speed on the grid, so this operation can be an application-specific constraint enforcement. Afterwards, the solver is called for all grids at the current level. Then, the integrator performs the r recursive calls to itself on the immediately finer level, followed by incrementing the time information. Finally, if the level is not the finest, then correction and projection are performed.

6.4.3 Refinement

HAMR's standard refinement algorithm (Figure 6.53) implements the recursive refinement algorithm described in Section 4.4. First, if the level is the deepest permitted, then no re-

```
Void Level_standard_refiner (Level lev) {
  if (Level_depth(lev) == Level_maximum_depth_allowed(lev))
    return;
  Level_standard_boundary_collector(lev);
  Level_overall_selected_create(lev);
  if (Level_grids(Level_finer_level(lev)) > 0)
    Level_standard_refiner(Level_finer_level(lev));
  Level_selected_sets_create(lev);
  Level_selector_method_execute(lev);
  Level_merge_selected_sets_into_overall_selected(lev);
  Level_select_domain_exterior(lev);
  Level_merge_permanently_selected_into_overall_selected(lev);
  Level_selected_sets_delete(lev);
  Level_expand_overall_selected(lev);
  if (Level_not_root(lev))
    Level_merge_overall_selected_into_coarser(lev);
  Level_clusterer_method_execute(lev);
  Level_overall_selected_delete(lev);
  Level_standard_regridder(Level_finer_level(lev));
  Level_clusters_delete(lev);
  if ((Level_grids(Level_finer_level(lev)) > 0) &&
      Level_is_at_initial_timestep(lev))
    Level_standard_refiner(lev);
  if (Level_not_finest_allowed(lev))
    Level_get_children(lev);
}
```

Figure 6.53: Standard refiner

finement is possible, so the operation terminates. Otherwise, boundary values are collected, in case the selection criterion requires them. Next, the flag fields that indicate the selected cells are allocated, in order to allow finer levels to map their flags into them, to ensure coverage. Then, if finer levels have grids, the refinement operation is called recursively. After the recursion terminates, the selection algorithm is applied, and the results are merged together to provide the aggregate selection on each grid, including the expansion to cover buffers and boundary regions. These expanded selection flags are merged with the selection flags at the immediately coarser level, to ensure that these grids are fully covered, and then the clustering algorithm is applied. The selection flag fields are deallocated, and then the regridding algorithm is applied. Next, if the refinement is occurring during initialization, then the refinement algorithm is applied recursively. Finally, the grids determine their relationships with their children.

6.4.4 Regridding

HAMR's standard regridding algorithm (Figure 6.54) replaces a set of grids with a new set that more properly covers the phenomena of interest. The algorithm begins by ensuring that the level is not the coarsest level, since the grids at the coarsest level are never altered, since they delimit the computational domain. Next, it deletes all enduring data items on the grids, since they will not be needed for regridding. If there are new grids to replace the old grids, then they are created, and their sibling and parent relationships are determined. The values on the new grids are injected from their parents, and then more accurate values are

```
Void Level_standard_regridder (Level lev) {
  if (Level_is_coarsest_existing(lev))
    return;
  old_grids = Level_grids(lev);
  old_grid = Level_grid(lev);
  for (g = old_grids - 1; g >= 0; g--)
    Grid_parameter_all_enduring_delete(Level_grid(lev)[g]);
  if (clusters == 0)
    { Level_grids(lev) = 0; Level_grid(lev) = NULL; }
  else {
    Level_grids(lev) = clusters;
    Level_grid(lev) = GridList_framework_allocate(Level_grids(lev));
    for (g = 0; g < new_grids; g++)
      Level_grid(lev)[g] = Grid_create(lev, g, cluster[g]);
    Level_get_siblings(lev);
    Level_get_parents(lev);
    if (Level_not_finest_existing(lev))
      Level_get_parents(Level_finer_level(lev));
    for (g = 0; g < new_grids; g++)
      Grid_inject_interior_from_coarser(Level_grid(lev)[g]);
    if (old_grids > 0)
      Level_copy_from_overlaps(lev, old_grid, old_grids);
  }
  if (old_grids > 0) {
    for (g = old_grids - 1; g >= 0; g--)
       Grid_delete(old_grid[g]);
    old_grid = GridList_framework_free(old_grid, old_grids);
  }
  Level_get_exterior_regions(lev);
  for (g = 0; g < new_grids; g++)
    Grid_parameter_all_enduring_create(Level_grid(lev)[g]);
}
```

Figure 6.54: Standard regridder

```
Void Grid_standard_boundary_collector (Grid grid) {
  if (Grid_not_at_coarsest_level(grid))
    Grid_standard_parent_boundary_collect(grid);
  Grid_standard_sibling_boundary_collect(grid);
  Grid_standard_boundary_reflect(grid);
  Grid_extrapolator_method_execute(grid);
}
```

Figure 6.55: Standard collector

```
Void Grid_standard_parent_boundary_collect (Grid grid) {
  for (i = 0; i < items; i++)
    for (par = 0; par < Grid_parents(grid); par++)
      for (member = 0; member < members; member++)
        Grid_inject_method_execute(
          grid, Grid_parent(grid)[par], REAL, SPACETIMEVARIABLESET, p,
          Grid_parent_in_boundary(grid)[par][MINIMUM]),
          Grid_parent_in_boundary(grid)[par][MAXIMUM]),
          Grid_boundary_in_parent(grid)[par][MINIMUM]),
          Grid_boundary_in_parent(grid)[par][MAXIMUM]));
}
```

Figure 6.56: Collection from parents

copied from any of the old grids that overlap the new grids. The old grids are then deleted, the exterior region information is obtained for them, and then the enduring data items are created.

6.4.5 Boundary Collection

HAMR's collection algorithm (Figure 6.55) obtains the ghost boundary values for a grid. Boundary values are collected from parents (Figure 6.56), siblings (Figure 6.57), from the

```
Void Grid_standard_sibling_boundary_collect (Grid grid) {
  for (i = 0; i < items; i++)
    for (sib = 0; sib < Grid_siblings(grid); sib++)
      for (member = 0; member < members; member++)
        RealField_offset_copy(
          Grid_Grid_Real_SpacetimeVariableSet(grid)
              [i][mb][old_time_level],
          Grid_Grid_Real_SpacetimeVariableSet(sibgrid)
              [i][mb][old_time_level],
          axis_set, staggering, sib_staggering, axis_loci, sib_axis_loci,
          sibling_boundary_size,
          Grid_sibling_in_boundary(grid)[MINIMUM],
          Grid_boundary_in_sibling(grid)[MINIMUM],
          dimension);
}
```

Figure 6.57: Collection from siblings

grid itself, and from the exterior, in that order. In this way, the most accurate value replaces less accurate values: the values injected from the coarser level are replaced by values copied from elsewhere on the local level, and values on the exterior of the problem domain are extrapolated.

The importance of this approach can be seen in Figure 6.58, which depicts the boundary collection for the protostellar jet example shown in Figure 2.13. Because the jet inflow is on an otherwise reflected boundary, the reflection must occur before the extrapolation, which in this case sets the jet inflow values on a small subset of the cells on the reflected exterior boundary. If the order were different, the jet inflow values would be replaced by reflecting the cells abutting the jet.

Reflecting and periodic boundaries are special cases that are addressed in a manner very

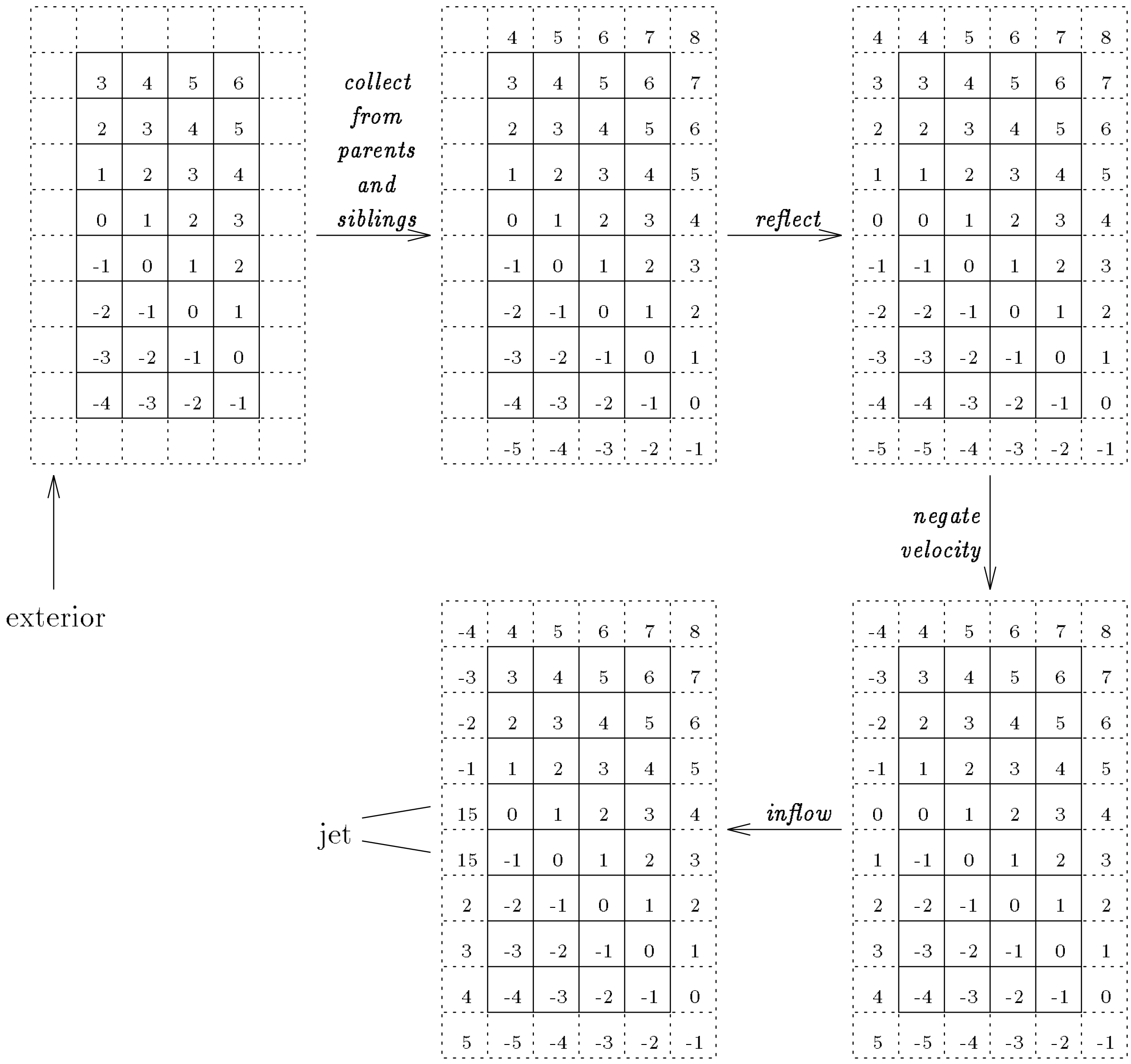


Figure 6.58: Boundary collection order

different from other boundaries, and very different from one another. Periodic boundaries are implemented not with special code, but by translating the domain when determining sibling relationships (Figure 6.59). Thus, obtaining periodic boundary values requires no additional instrumentation in the sibling boundary collection algorithms, and minimal additional instrumentation in the algorithm that determines sibling relationships, since the translation requires nothing more than making extra copies of the input list of grid regions and adding a constant to them.

Reflecting boundaries, on the other hand, require an entire additional algorithm. For each grid, the algorithm determines whether any of its boundaries reflect — based on a set of flags encapsulated on the hierarchy, and the position of the grid within the overall domain — and if so, it copies their outermost interior values into the exterior boundary region, using a stride of -1 along the reflecting axis.

Some physical quantities, such as velocities, negate their values on reflection. Since the boundary reflection algorithm is not in a position to know whether a particular variable has this property, the negation can be implemented as the extrapolator for the variable; that is, each variable that requires negation can have as its `extrapolate` attribute a function that negates it if the exterior region is along the appropriate axis. (For example, the x component of velocity should only be negated along a reflecting boundary that is perpendicular to the x-axis.) If additional instrumentation is required, for example in the case of the small jet inflow within a reflecting boundary, then the call to the negation algorithm can be bundled with the inflow routine.

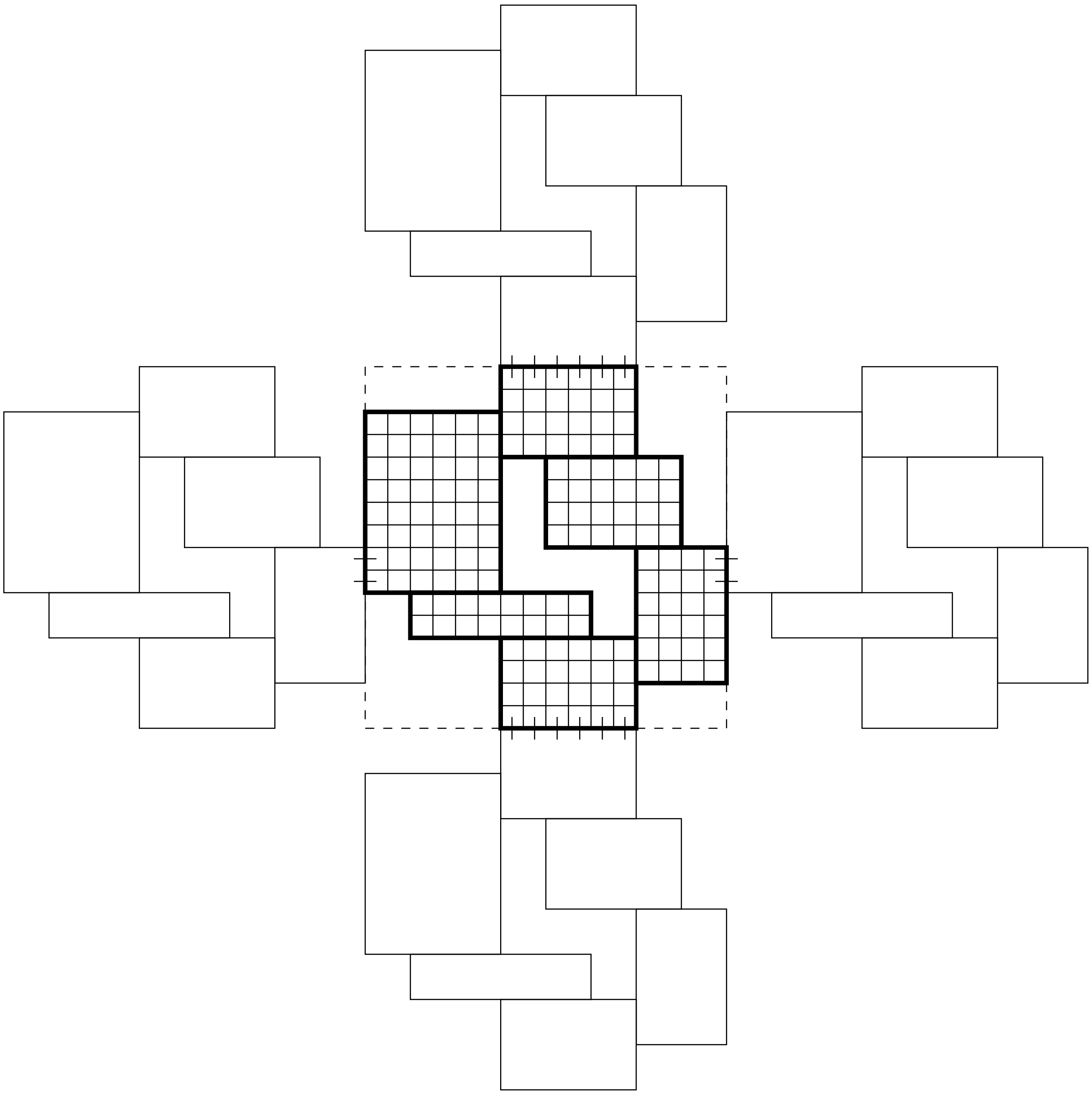


Figure 6.59: Periodic boundaries implemented as translated siblings

```
Void Level_standard_time_incrementer (Level lev) {
  for (i = 0; i < real_list_items; i++)
    if (Level_is_time(lev, REAL, LIST, i)) {
      Level_new_time_level(lev,REAL,LIST)[i] =
        (Level_new_time_level(lev,REAL,LIST) [i] + 1) %
        Level_elements_scalar(lev,REAL,LIST,i);
      Level_old_time_level(lev,REAL,LIST)[i] =
        (Level_old_time_level(lev,REAL,LIST) [i] + 1) %
        Level_elements_scalar(lev,REAL,LIST,i);
      Level_absolute_level_timestep(lev,REAL,LIST)[i][new_time_level] =
        Level_absolute_level_timestep(lev,REAL,LIST)[i][old_time_level] + 1;
      Level_time_interval_from_previous(lev,REAL,LIST)[i][new_time_level] =
        Level_root_time_interval(lev) /
        Level_axis_aggregate_refinement_factor_from_root(lev,TIME);
      Level_absolute_time(lev,REAL,LIST)[i][new_time_level] =
        Level_absolute_time(lev,REAL,LIST)[i][old_time_level] +
        Level_time_interval_from_previous(lev,REAL,LIST)[i][new_time_level];
    }
  Level_old_time_match(lev);
}
```

Figure 6.60: Standard incrementer

6.4.6 Incrementing Time Information

HAMR's standard incrementing algorithm (Figure 6.60) updates the time information for a level after an integration has been completed. For each `RealList` that represents time information, the algorithm increments the old and new time level indices modulo the number of time levels, increments the timestep number, sets the time interval to the root time interval divided by the aggregate time refinement factor, and adds the time interval to the old physical time to obtain the new physical time. When all such updates have been made, the incrementer copies the level's physical time values to all finer levels, to eliminate accumulated roundoff error. This last operation is safe, because the incrementer is called after a set of finer integrations has occurred, at which point the physical time on the finer level has caught up to the physical time on the coarser level, except for roundoff.

Because the incrementer is applied immediately following the completion of the finer iterations, all operations on the grid occur on the old time level, not on the new time level. The only exception is injecting a value from a parent; in this case, the injection is interpolated between the old and new time levels.

6.4.7 Richardson Truncation Error Estimation

Richardson truncation error estimation (described in Section 4.4.6), while both intuitive and mathematically simple, is perhaps the most difficult algorithm to generalize for an arbitrary solver. Indeed, many implementations of Berger's AMR strategy either implement only a simplified version of the algorithm, or require considerable reinstrumentation of the solver. One AMR formulation, for example, required that the solver accept a stride, which would be one in the case of genuine evolution or two in the case of half resolution error estimation calculations. Choptiuk, on the other hand, constantly maintains a half-resolution shadow hierarchy, computing every alternate timestep on it, which results in a half resolution approximation of the error estimation, and which constrains the refinement factor to powers of two.

The aspect of the error estimation algorithm that is the source of all the difficulty is, in fact, the half resolution $\mathbf{Q}_{2h}(\mathrm{u})$ calculation. While in principle this calculation is straightfor-

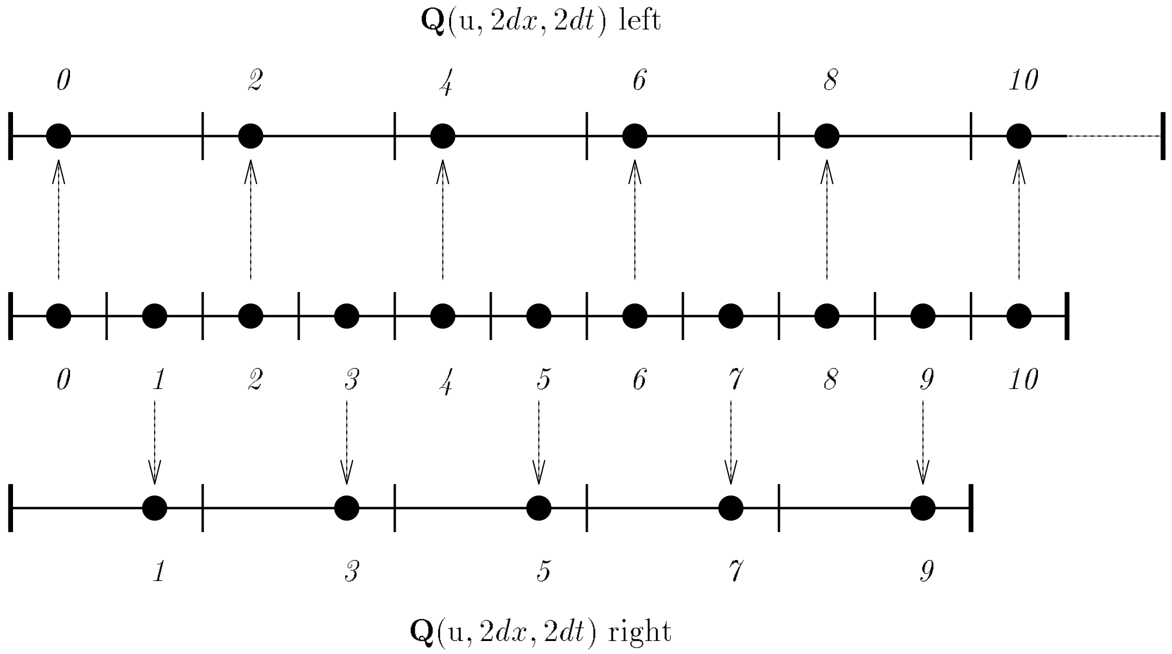


Figure 6.61: Half resolution grids for Richardson truncation error

ward — the solver calculates based on every other value, rather than each value — in practice it is difficult to achieve, because in a generalized AMR system the solver is not, and should not be, in a position to know whether the calculation is full resolution or half resolution, or whether the calculation is genuine or for error estimation. Thus, in order for the truncation error algorithm to be fully generalized, the $\mathbf{Q}_{2h}(\mathrm{u})$ calculation must be performed on a grid that is in every way identical to the standard grid, except that it has half the resolution. Or rather, it must be performed on 2^d such grids, whose values are drawn starting from either the first or second value along each axis of the original grid (as shown in Figure 6.61).

Still, constructing the half resolution grids should not be overly burdensome. But in addition to this requirement, the collection of half resolution grids, taken in aggregate, also require twice as many ghost boundary values as the full resolution grid (as shown in Figure 4.21). These boundary values are drawn not only from parents but from siblings and the exterior as well. It is this requirement that has kept many researchers from implementing generalized error estimators.

The solution that HAMR implements involves constructing a dummy grid that is identical to the original grid in all but two respects. First, every spatial data item on the dummy grid has a double-width ghost boundary; that is, all stencils are doubled. Second, the original grid's boundary relationship information is discarded, and its boundary relationships are recalculated, thereby providing the necessary information for drawing the appropriate double-width set of boundary values, using the existing boundary collection algorithm — whatever that may be. In this manner, proper boundary information is obtained not only from the parents, but also from siblings and exterior regions — including, conceivably, parents, siblings and exterior regions that the original grid does not require (Figure 6.62). After the boundary values have been collected, the expanded grid's non-permanent data items can be deallocated, because the expanded grid will be used only as a source of values for the half resolution grids on which $\mathbf{Q}_{2h}(\mathrm{u})$ is calculated.

In fact, creating the half resolution grids is now trivial. Each of the 2^d half resolution grids is operated on in turn: it is created; its values are copied directly from the expanded grid, starting in the boundary region at either the first or the second locus along each axis and using a stride of 2, thereby automatically obtaining the appropriate boundary values; the solver is applied to it, just as if it were a standard grid, but using a time interval twice

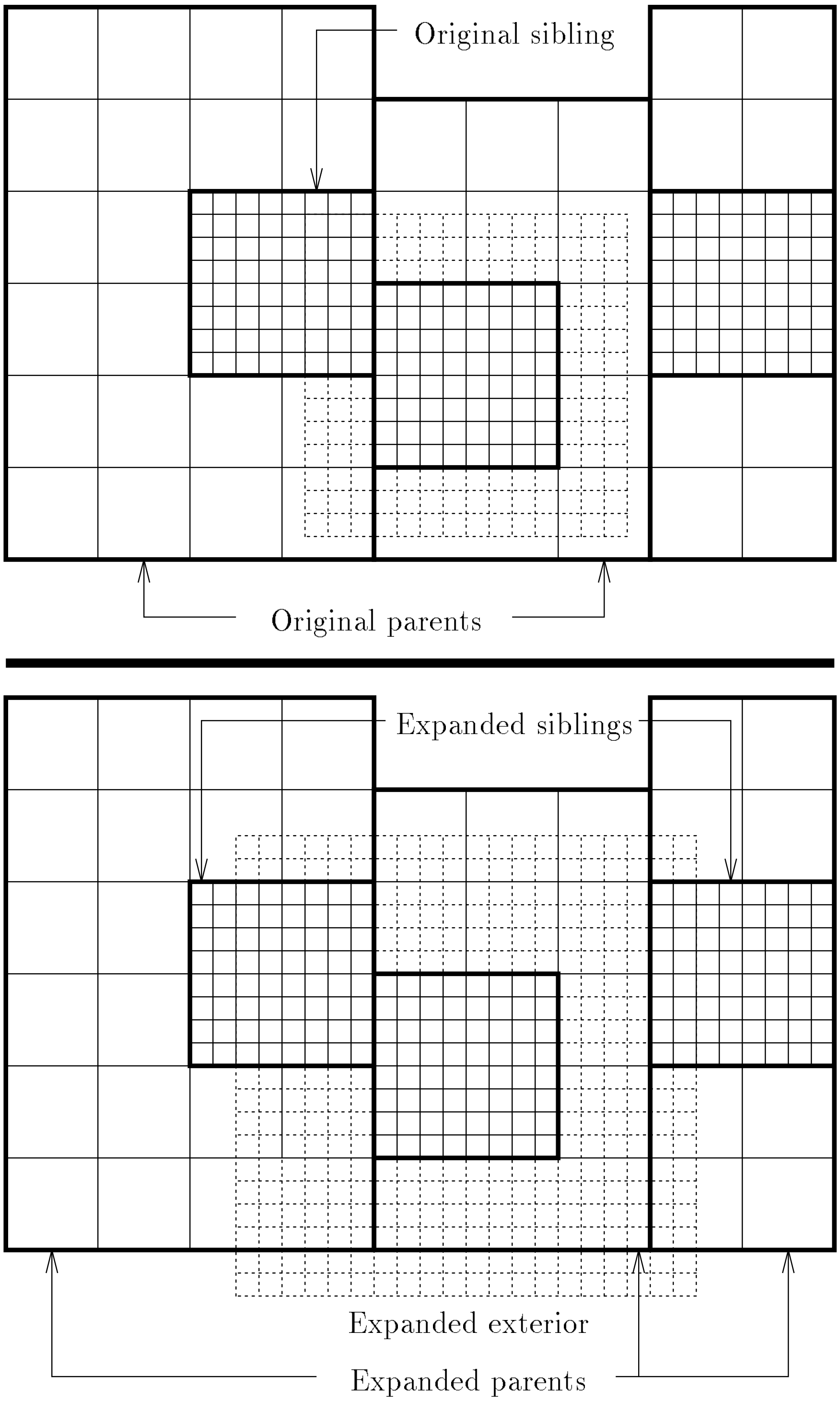


Figure 6.62: Expanded dummy grid boundaries

the length of the original time interval; the values on the interior of its "new" field are copied into the interior of the "new" field of the original grid, again starting at either the first or the second locus along each axis and using a stride of 2; finally, it is deleted. In this manner, all of the half resolution grids are solved, but at no time does more than one such grid exist.

When all of the half resolution grids have been solved and discarded, the expanded grid can be discarded as well, since its purpose is merely to provide values for the half resolution grids. Thus, during the process of computing the half resolution solution, the extra storage required is only the expanded grid — or slightly less, since some of its data items can be deleted — plus a single grid whose size is $1/2^d$ of the original grid, and two copies of the grid data structure. The value of $\mathbf{Q}_{2h}(\mathrm{u})$ for all interior loci is stored in the "new" field of the original grid, a field that is otherwise unused, because refinement on a level can occur only after the most recent timestep has been completed, and that timestep's solution is stored in the "old" field.

Once the half resolution solution has been obtained, the next step is to obtain the full resolution solution over two timesteps. This operation is considerably simpler, because it can be performed on a single grid that is identical to the original. So, a copy of the original grid is created; its boundary values are collected; it is evolved the first time; its time information is incremented; its boundary values are collected again; it is evolved the second time; the values on its "new" field — $\mathbf{Q}_h(\mathbf{Q}_h(\mathrm{u}))$ — are subtracted from the values on the original grid's "new" field, with the result remaining in the original grid's "new" field; finally, it is deleted. Thus, during the operation, the total amount of additional memory consumed is the amount consumed by the original grid, and when the full resolution timesteps are finished, the values on the interior of the original grid's "new" field are $\mathbf{Q}_{2h}(\mathrm{u}) - \mathbf{Q}_h(\mathbf{Q}_h(\mathrm{u}))$, which is, in fact, the Richardson truncation error estimate times a constant.

Superficially, this approach to computing truncation error appears extraordinarily wasteful of memory; after all, an additional 2^d+2 grids are required to obtain the result. On closer examination, however, it becomes clear that the algorithm is insignificantly more wasteful than an approach whose memory consumption is fully optimal, even one which achieves optimal consumption by reinstrumentation of the solver. The reason this is so is that, no matter how one approaches computing the half resolution timestep, the two full resolution timesteps absolutely require a complete copy of the original grid. This requirement arises because these timesteps, if performed on the original grid itself, would overwrite the values of the old time level(s), so the old values must be stored separately from the full resolution grid, and because of the need to store the half resolution result before computing its two full resolution counterparts. In other words, the "new" field is needed for storing the intermediate half resolution result, and the "old" field(s) are needed for retaining the "old" results, which will be required either to compute the next genuine timestep at the level that is being refined, if the grid is on the coarsest level being refined at the moment, or to be copied onto the overlapping replacement grids, if it is on a finer level.

Thus, since an entire additional grid is required for computing the full resolution result, the waste associated with this approach is the difference between the size of the original grid and the sum of the expanded grid and one of the half resolution grids. Careful examination reveals that the relative waste is large only in the case of very small grids, but that in such cases the absolute waste is quite small (aside from the fixed size of the additional grid data structure). For example, an $8 \times 8 \times 8$ grid with a seven point stencil will have a $14 \times 14 \times 14$ bounded field comprising 2744 loci, while its expanded counterpart with a thirteen point stencil will have a $20 \times 20 \times 20$ bounded field comprising 8000 loci, as well as a $4 \times 4 \times 4$ half resolution grid with a seven point stencil, which will have a $10 \times 10 \times 10$ bounded field comprising 1000 loci; thus, the total waste will be $8000 + 1000 - 2744 = 6256$ loci, or about 2.28 times the original bounded field — which is a high relative waste but a very low absolute waste. On the other hand, a $64 \times 64 \times 64$ grid with a seven point stencil will have a $70 \times 70 \times 70$ bounded field comprising 343000 loci, while its expanded counterpart with a thirteen point stencil will have a $76 \times 76 \times 76$ bounded field comprising 438976 loci, as well as a $32 \times 32 \times 32$ half resolution grid with a seven point stencil, which will have a $38 \times 38 \times 38$ bounded field comprising 54872 loci; thus, the total waste will be $438976 + 54872 - 343000 = 150848$ loci, or about 0.44 times the original bounded field — which is a high absolute waste but a low relative waste.

Thus, this approach provides a fully generalized truncation error estimation algorithm, which will function perfectly well regardless of the details of the boundary value collector, the solver, the time incrementer and other such AMR modules, yet it requires very little memory beyond the absolutely optimal memory consumption achievable with radical re-instrumentation of the application algorithms. This approach's generality is achieved not only without reinstrumentation of application routines, but also without declaring a single additional data item.

6.4.8 Flux Correction

Although HAMR's flux correction algorithm (Figure 6.63) is algorithmically simple, it involves a variety of operations, and demonstrates the importance of functional data attributes. The algorithm begins by setting the grid's boundaries to zero, for reasons that will be explained presently. Next, the correction values on all of the child grids are calculated, described below. A query to the specification produces the correction data attribute, from which the proper surfaces of the children are obtained. Then, for each minimum (left) surface of each child, the correction values are subtracted from the coarse cells immediately before the surface, and for each maximum (right) surface, the correction values are added to the coarse cells immediately after the surface.

Interface correction (Figures 6.64 and 6.65) begins with a query to the specification that obtains the precorrection and postcorrection attributes, from which pointers to the correction surfaces are obtained. Then, for each interface, the precorrection surface values are divided by the product of the refinement factors not along that surface — for example, an x-face is divided by $r_y \cdot r_z$. Next, the postcorrection surface values are subtracted from the precorrection surface values, with the result placed in the postcorrection surface. Then, the finer postcorrection surface values are summed and placed into the associated coarser precorrection surface loci. Finally, the results are divided by the sizes of the coarse cell faces.

```
Void Grid_standard_correct (Grid grid, Index i) {
  Grid_old_boundary_set_to_constant(grid, (Real)0);
  for (c = 0; c < Grid_interior_children(grid); c++)
    Grid_standard_correct_interfaces(Grid_interior_child(grid)[c], i);
  prest = Specification_precorrection_data_archetype_component_query(
            Grid_specification(grid),
            GRID, REAL, SPACETIMEVARIABLESET, i, ARCHETYPE_SYSTEM_LEVEL);
  prep  = Specification_precorrection_data_archetype_component_query(
            Grid_specification(grid),
            GRID, REAL, SPACETIMEVARIABLESET, i, ARCHETYPE_PARAMETER);
  for (c = 0; c < Grid_interior_children(grid); c++) {
    precorrection_data =
      *Grid_parameter_address(child_grid,
        prest, REAL, SURFACEPARAMETERSET, prep);
    for (interface_axis = 0; interface_axis < dimension; interface_axis++) {
      IndexPoint_copy(interface_start,
        Grid_child_interior_in_interior(grid)[MINIMUM], dimension);
      interface_start[interface_axis] -= 1;
      for (member = 0; member < members; member++)
        RealField_offset_add(
          grid_data[mb][old_time_level], grid_data[mb][old_time_level],
          precorrection_data[interface_axis][MINIMUM][mb],
          axis_set, staggering, staggering, interface_staggering,
          axis_loci, axis_loci, precorrection_axis_loci,
          precorrection_axis_loci,
          interface_start, interface_start, index_zero, dimension);
      interface_start[interface_axis] =
        Grid_child_interior_in_interior(grid)[MAXIMUM][interface_axis] + 1;
      for (member = 0; member < members; member++)
        RealField_offset_subtract(
          grid_data[mb][old_time_level], grid_data[mb][old_time_level],
          precorrection_data[interface_axis][MAXIMUM][mb],
          axis_set, staggering, staggering, interface_staggering,
          axis_loci, axis_loci, precorrection_axis_loci,
          precorrection_axis_loci,
          interface_start, interface_start, index_zero, dimension);
    } } }
```

Figure 6.63: Standard corrector

```
Void Grid_standard_correct_interfaces (Grid child_grid, Index i)
{
  prest =
    Specification_precorrection_data_archetype_component_query(
      Grid_specification(child_grid),
      GRID, REAL, SPACETIMEVARIABLESET, i, ARCHETYPE_SYSTEM_LEVEL);
  prep  =
    Specification_precorrection_data_archetype_component_query(
      Grid_specification(child_grid),
      GRID, REAL, SPACETIMEVARIABLESET, i, ARCHETYPE_PARAMETER);
  precorrection_data =
    *Grid_parameter_address(child_grid,
      prest, REAL, SURFACEPARAMETERSET, prep);
  postst =
    Specification_postcorrection_data_archetype_component_query(
      Grid_specification(child_grid),
      GRID, REAL, SPACETIMEVARIABLESET, i, ARCHETYPE_SYSTEM_LEVEL);
  postp  =
    Specification_postcorrection_data_archetype_component_query(
      Grid_specification(child_grid),
      GRID, REAL, SPACETIMEVARIABLESET, i, ARCHETYPE_PARAMETER);
  postcorrection_data =
    *Grid_parameter_address(child_grid,
      postst, REAL, SURFACEPARAMETERSET, postp);
```

Figure 6.64: Interface correction (part 1)

```
for (interface_axis = 0; interface_axis < dimension; interface_axis++) {
  for (axis = AXIS1; axis <= maxaxis; axis++) if (axis != interface_axis) {
      interface_refinement_factor *=
        Grid_refinement_factor_from_coarser(grid)[axis];
      interface_size *= Grid_isotropic_cell_size(child_grid)[axis]; }
  for (interface_ext = MINIMUM; interface_ext <= MAXIMUM; interface_ext++)
    for (member = 0; member < members; member++) {
      RealField_contiguous_divide_by_constant(
        precorrection_data[member][interface_axis][interface_ext],
        precorrection_data[member][interface_axis][interface_ext],
        axis_set, staggering, precorrection_axis_loci, dimension,
        (Real)interface_refinement_factor);
      RealField_refinement_from_first_subtract(
        postcorrection_data[member][interface_axis][interface_ext],
        precorrection_data[member][interface_axis][interface_ext],
        postcorrection_data[member][interface_axis][interface_ext],
        axis_set, staggering, staggering, staggering,
        postcorrection_axis_loci,
        precorrection_axis_loci, postcorrection_axis_loci,
        postcorrection_axis_loci,
        index_zero, index_zero, index_zero, ...
        Grid_refinement_factor_from_coarser(child_grid), dimension);
      RealField_projection_sum(
        precorrection_data[member][interface_axis][interface_ext],
        postcorrection_data[member][interface_axis][interface_ext],
        axis_set, staggering_with_interface_axis_offset_from_center,
        precorrection_axis_loci,
        postcorrection_axis_loci, precorrection_axis_loci,
        index_zero, index_zero, index_zero, index_zero,
        Grid_refinement_factor_from_coarser(child_grid), dimension);
      RealField_contiguous_divide_by_constant(
        precorrection_data[member][interface_axis][interface_ext],
        precorrection_data[member][interface_axis][interface_ext],
        axis_set, staggering, precorrection_axis_loci, dimension,
        (Real)interface_size);
    } } }
```

Figure 6.65: Interface correction (part 2)

More formally, on a z-face, for example,

$$
\begin{aligned}
pre_{z_{\min}} &= F^{\text{pre}} \\
post_{z_{\min}} &= \textstyle\sum_{i=1}^{r_t} f^{\text{post}} \\
pre_{z_{\min}} &\leftarrow \frac{pre_{z_{\min}}}{r_x r_y} && \equiv \frac{F^{\text{pre}}}{r_x r_y} \\
post_{z_{\min}} &\leftarrow pre_{z_{\min}} - post_{z_{\min}} && \equiv \frac{F^{\text{pre}}}{r_x r_y} - \textstyle\sum_{i=1}^{r_t} f^{\text{post}} \\
pre_{z_{\min}} &\leftarrow \textstyle\sum_{i=1}^{r_x} \sum_{j=1}^{r_y} post_{z_{\min}} && \equiv \textstyle\sum_{i=1}^{r_x} \sum_{j=1}^{r_y} \left(\frac{F^{\text{pre}}}{r_x r_y} - \sum_{i=1}^{r_t} f^{\text{post}} \right) \\
& && \equiv F^{\text{pre}} - \textstyle\sum_{i=1}^{r_x} \sum_{j=1}^{r_y} \sum_{i=1}^{r_t} f^{\text{post}} \\
pre_{z_{\min}} &\leftarrow \frac{pre_{z_{\min}}}{\Delta x \cdot \Delta y} && \equiv \frac{F^{\text{pre}} - \sum_{i=1}^{r_x} \sum_{j=1}^{r_y} \sum_{i=1}^{r_t} f^{\text{post}}}{\Delta x \cdot \Delta y}
\end{aligned}
$$

Thus, the proper correction is obtained from the fluxes, without requiring any extra storage for intermediate results and at the cost of a single additional division by a constant.

When a fine grid abuts the interface of its coarser parent (Figure 6.66), the correction values along that interface are added to the ghost boundary of the parent, which as noted above has been set to zero. Then, after all of the grids have been corrected, the grid boundaries are added to the sibling interiors that overlap them, in much the same manner as sibling values are copied into the ghost boundaries during boundary collection. Since the boundaries were set to zero before correction, any non-zero values in the boundaries are correction values that should be applied to computed coarse cells immediately adjacent to the fine interfaces, but that were not because those coarse cells are on other grids. Those other grids are the siblings of the corrected grid, so by adding the grid boundaries to the overlapping sibling interiors, the correction values are properly applied. Because of grid coverage, all

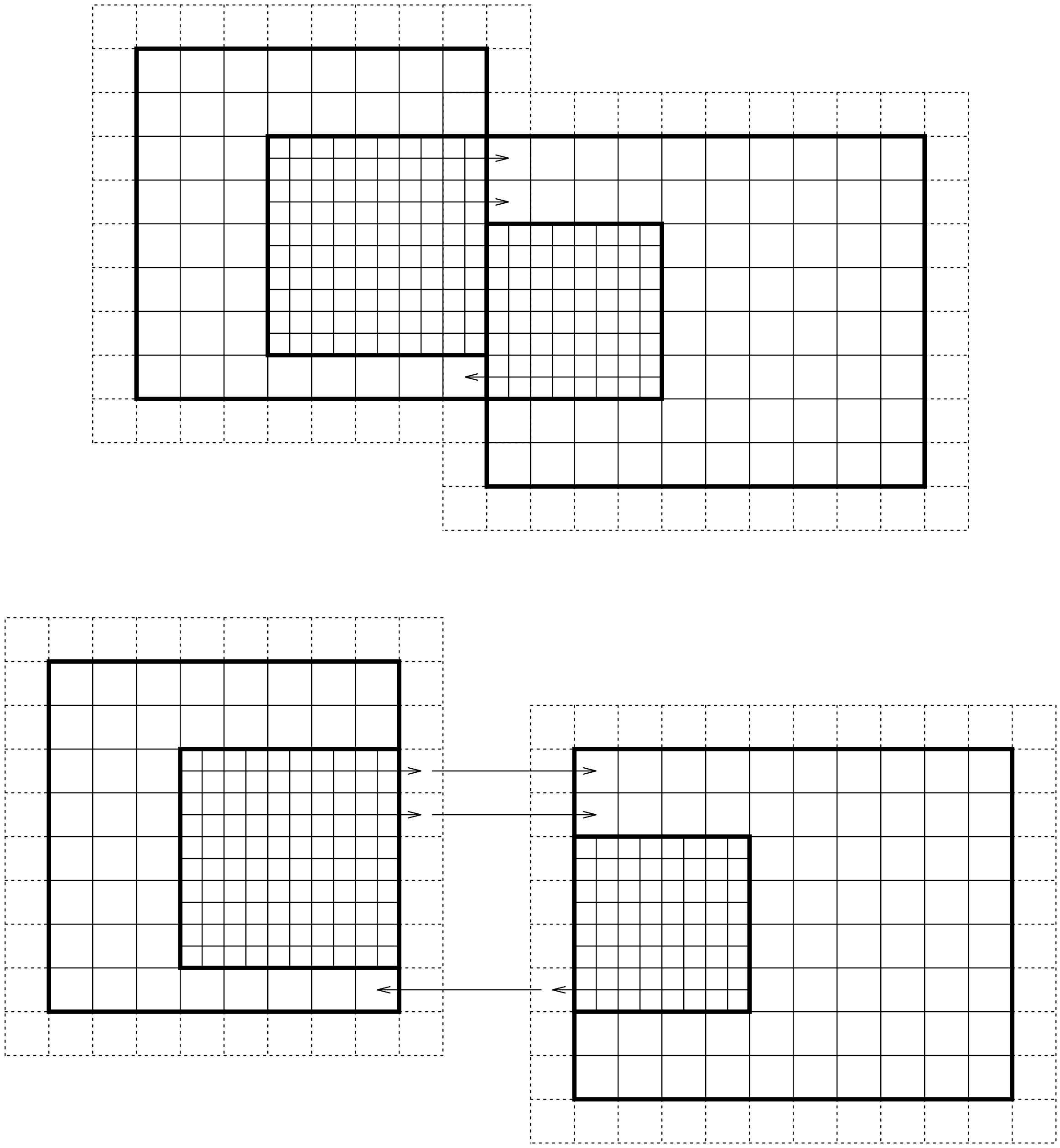


Figure 6.66: Sibling correction

corrections are constrained to apply to cells within the corrected grid, to cells within siblings, or to cells on the exterior; in the last case, the correction values are superfluous.

6.4.9 Summary

Ultimately, the AMR algorithms contained in HAMR constitute an achievement of what has, until now, been merely a long-term interest of the AMR community: a general-purpose AMR framework. Without the underlying data infrastructure that HAMR provides, general-purpose AMR is at best difficult, not only because of the data needs of simulations but also because of the need for the ability to express the data and method relationships. Thus, the contribution of this dissertation is expressed clearly by HAMR's capabilities: to incorporate adaptivity into an almost limitless variety of structured multiscale simulations.

6.5 HAMR Summary

Fundamentally, the distinguishing characteristics of HAMR that set it apart from other implementations of Berger's AMR strategy are those qualities which promote generality and autonomy. Each component of the HAMR system – the data types, the function library, the data structure and management, and the implementations of the AMR algorithms — contribute to these two properties. The wide variety of data types makes possible the incorporation of applications and algorithms whose data needs are complex. The flexibility in performing various basic operations, which the many AMR algorithms require, are provided by the function library in a self-consistent, intuitive manner. The data management infrastructure provides not only autonomy, but also the ability to decouple data management issues from both the AMR algorithms and the application. Finally, the AMR algorithms themselves take full advantage of the flexibility and generality that are a natural outgrowth of the properties and capabilities of the other components.

Designing, implementing and testing HAMR was a long and laborious process, the extent of which was not anticipated at the outset. During the overwhelming majority of this process, no aspect of the components had any significant value beyond its anticipated future role within the overall environment. HAMR's many layers are so completely integrated that it was impossible to employ it for anything useful prior to completion, and this facet of its development proved intensely frustrating to all involved.

In the end, however, HAMR has proven itself, in design, implementation, and execution.