

Application of Design Patterns to Geometric Decompositions *

V. Balaji † Thomas L. Clune ‡ Robert W. Numrich § Brice T. Womack ¶

March 14, 2005

1 Introduction

We describe an abstract architectural design pattern that represents many of the parallel codes used in the Earth Sciences such as weather, climate and ocean models. Our prototype example is a grid-based code, a code that implements finite-difference operators over fields defined on a grid of coordinates representing physical space. These codes involve the decomposition of data, the distribution of data across a parallel architecture, and the assignment of work to that data. A typical approach to this problem is to partition the physical coordinate space into subdomains. The actual software representation of these subdomains is then extended so as to include regions of overlap with neighboring subdomains. These overlap regions are called halo cells, ghost cells, or guard cells in various communities, and contain redundant copies of information properly associated with the interior of the corresponding neighboring subdomains. The width of these overlapping regions is determined by the requirements of a particular finite difference stencil used to discretize a particular differential operator. The subdomains interact with each other in some manner so they know when to update their halo cells with fresh data from their neighbors.

The difficult part of such an approach is deciding on what a subdomain means, what a neighbor to a subdomain means, how to synchronize among subdomains at the appropriate times, and how to move data from one subdomain to another. These problems have been solved over and over again using many different approaches with specific implementations depending on the target computer, for example, a shared memory version for one computer and a distributed memory version for another or a mixed shared plus distributed version for yet another. Implementations may be specific to specific finite difference stencils, to specific curvilinear coordinate systems, to specific differential operators or to specific approximations to the physical model.

The **Composite(163)** pattern, with the number in parentheses referring to the page number in reference [2], closely represents the domain decomposition procedure used in many of our codes [4]. It allows us to maintain a unified picture of subdomains whether we want to think of them in a global setting, the problem as a whole, or in a local setting, the problem as decomposed into local pieces. Methods associated with the composite have the same interface at each level of the decomposition and allow the software developer to move from one view of the problem to another in a consistent manner. We combine the composite pattern with the **Iterator(257)** pattern, which allows the developer to traverse the decomposition at either the global level or the local level without introducing dependencies related to a specific decomposition.

Different memory models commonly used for high performance computing have different performance implications that affect the design for our composite. Ideally, we would like to change our

*papers/patterns/patterns_v7.tex

†Princeton University and NOAA Geophysical Fluid Dynamics Laboratory, Princeton, NJ

‡NASA Goddard Space Flight Center, Greenbelt, MD

§University of Minnesota, Minneapolis, MN

¶Northrop Grumman IT TASC, Greenbelt, MD

composite across memory and processor boundaries depending on the underlying computer architecture. By applying the **Builder(97)** pattern, we can isolate the composite's creation step and use the **Strategy(315)** pattern to produce our composite in a way that is optimal for the target architecture, for a particular numerical algorithm, or for a particular set of physical or geometric constraints. The programmer selects a build strategy that fits the problem, and the **Builder(97)** returns a decomposition appropriate for that strategy. The programmer changes the decomposition by selecting a different strategy without having to change the rest of the code, which can be written assuming only an abstract decomposition.

For numerical algorithms within our field the data dependencies are frequently bi-directional. Therefore, each part of the composite's decomposition are both data producers and data consumers. Each piece has data dependencies with adjacent parts of the problem domain. The **Observer(293)** pattern is a likely candidate for decoupling the interaction between these subdomains. But we need a variation of the observation pattern that allows bi-directional observation.

To decouple the communication mechanism from any particular implementation, we can create an interface for the data exchange that is based on the Request/Require/Release abstraction [1]. We then use the **Mediator (293)** pattern to allow additional flexibility in defining the method for exchanging data. As part of the strategy used to build the decomposition, we configure mediators that decouple each individual subdomain from neighboring subdomains. By encapsulating information about a subdomains halo regions and the communication methods used to perform the data exchange with other subdomains, we can create an implementation that is efficiently adaptable to various decompositions that best fit the problem depending on whether the target computer is a shared memory architecture, a distributed memory architecture, or a cluster architecture with a combination of shared and distributed memory.

Our discussion is related to ideas discussed by Mattson, et.al. [3]. In particular, their discussions of Geometric Decomposition in Section 4.6, Distributed Arrays in Section 5.10, Communication in Section 6.4, Task Decomposition in Section 3.2, and Data Decomposition in Section 3.3 have aspects related to our architectural design. Our contribution is that we relate these ideas to well-defined design patterns as described by Gamma, et.al. [2] and combine them into an abstract description of a widespread approach to the design of parallel programs, which has been used over and over again not only in the Earth Sciences community but also in many other communities of computational physical science.

2 Description of Our Architecture Pattern

2.1 Composite Pattern for Mapping Subdomains

The composite pattern allows the programmer to represent a complex hierarchical structure so that individual objects and compositions of those objects are handled in a uniform way. In the case of grid-based codes, the objects we have in mind are physical grids, which we think of as the global domain, and partitions of that grid, which we think of as subdomains of the grid. Normally this kind of partitioning is referred to as domain decomposition in contrast with the pattern classification where it is referred to as a composition. This difference in description is largely a function of which elements of the problem are considered as givens. For domain decomposition, the global domain is primary with the notion of subdomains arising secondarily from the requirement to support parallelization. In most other cases, however, the composition pattern arises when a secondary requirement arises to manage a collection of objects from a given class.

2.2 Builder Pattern for Building the Composite

We use the builder pattern to create the composition based on a particular strategy. For our purposes, the builder creates the composition and defines a map for the complete problem domain that

contains all the information necessary to relate the global domain to its collection of subdomains in either direction. Every subdomain is assigned to a specific processor. Each subdomain knows its relation to the global domain and has a decomposition and architectural neutral interface for accessing data from its neighboring subdomains both locally and on other processors. The programmer is supplied with appropriate interfaces on each composite piece that allow access to whatever information is needed to understand the relationship between local and global subdomains.

2.3 Strategy Pattern for Picking the Best Map

We use the strategy pattern to configure the optimal kind of composite to build, or, in other words, to decide what kind of domain decomposition to use. Each build strategy is designed to create a domain decomposition that works well on a specific target computer architecture and has the appropriate communication mode for the numerical algorithm, the right geometry and boundary conditions for the physical problem, and provides reasonable load balancing.

2.4 Observer Pattern for Synchronization between Subdomains

Each subdomain of the composite implements a bi-directional variation of the observer pattern and defines the synchronization requirements between subdomains. For example, a subdomain may become an observer to a neighboring subdomain by issuing a `READ_REQUEST` to that subdomain. A subdomain responds to its observers when it has finished updating its local data and it is safe for its observers to read. The observers receive the signal through a `READ_REQUIRE`, which blocks until it receives the signal from the corresponding subdomain. A common simplification from the general case is to assume that all subdomains follow the same sequence of events in order. Such is not true of all applications, but is sufficiently common to warrant mention.

2.5 Mediator Pattern for Adapting Communication Methods

We use the mediator pattern to manage the mapping of data dependencies of halo regions among neighboring subdomains and to change the underlying communication mechanism used for the `REQUEST`, `REQUIRE` and `RELEASE` [1] methods of the composite's subdomains. Depending on the specific build strategy used to create the composite, the methods may be implemented using MPI, SHMEM, MPI-2, Co-Array Fortran, ARMCI, or Threads to handle the data exchange between subdomains. However, by moving the expression of parallel data dependency *above* the machine, this formalism allows considerable freedom in developing new innovative software expressions matched to innovations in parallel architectures.

2.6 Iterator Pattern for Traversing the Composite

In a typical grid-code, each processor works only on the set of subdomains that are located in its own local memory. We therefore see a need to provide a local iterator over the composite pattern that allows the programmer to perform work on local subdomains independent of the specific composition or the specific strategy used to build the composition. We also expect to provide a global iterator that allows the programmer to iterate over subdomains that do not reside in the local memory of the physical processor doing the work. Both iterators would be based on the same pattern hiding the specifics of the underlying composition and allowing the programmer to specify different compositions without changing the inner code structure that actually does the work on subdomains. There would also be an iterator over neighboring subdomains. This permits the development of *global task queues*, and a software architecture permitting low-latency task-switching when tasks stall for memory.

References

- [1] V. Balaji and Robert W. Numrich. A uniform programming model for complex distributed data objects in distributed and shared memory. In *Proceedings ECMWF High Performance Computing Workshop*. World Scientific, 2004.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [3] Timothy G. Mattson, Beverly A. Snaders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.
- [4] Robert W. Numrich. Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax. *Parallel Computing*, 2005.