

POOMA

A C++ Toolkit for High-Performance Parallel Scientific Computing

Jeffrey D. Oldham
CodeSourcery, LLC

POOMA: A C++ Toolkit for High-Performance Parallel Scientific Computing

by Jeffrey D. Oldham

Copyright © 2002 by CodeSourcery, LLC (<http://www.codesourcery.com/>)

All rights reserved. This document may not be redistributed in any form without the express permission of the author.

Revision History

Revision 1.00 2002 Jan 31 Revised by: jdo

First publication.

Table of Contents

Acknowledgements	vii
1. Introduction.....	8
1.1. POOMA Goals.....	8
1.2. POOMA is Open-Source Software	12
1.3. History of POOMA.....	13
2. Programming with Templates.....	15
2.1. Templates Execute at Compile-Time	15
2.2. Template Programming for POOMA Users	16
2.3. Template Programming Used to Write POOMA.....	21
3. A Tutorial Introduction	29
3.1. Installing POOMA	31
3.2. Hand-Coded Implementation.....	32
3.3. Element-wise Array Implementation	35
3.4. Data-Parallel Array Implementation	39
3.5. Stencil Array Implementation.....	43
3.6. Distributed Array Implementation	47
3.7. Data-Parallel Field Implementation.....	56
3.8. Distributed Field Implementation	61
4. Overview of POOMA Concepts.....	66
4.1. POOMA Containers.....	67
4.1.1. Choosing a Container.....	68
4.1.2. Declaring Sequential Containers.....	69
4.1.3. Declaring Distributed Containers	71
4.2. Computation Modes.....	72
4.3. Computation Environment.....	73
5. Array Containers	75
5.1. Containers	75
5.2. Arrays.....	75
5.3. Domains	76
5.3.1. Declaring Domains	78
5.3.1.1. Locs.....	79
5.3.1.2. Intervals.....	81
5.3.1.3. Ranges.....	83
5.3.1.4. Grids	85
5.3.2. Using Domains.....	88
5.4. Declaring Arrays.....	89

5.5. Using Arrays	97
5.6. DynamicArrays	103
6. Engines	109
6.1. The Concept	109
6.2. Types of Engines	110
7. Data-Parallel Expressions	114
7.1. Expressions with More Than One Container Value	114
7.2. Using Data-Parallel Expressions	114
7.3. Implementation of Data-Parallel Statements	126
7.3.1. Naïve Implementation	127
7.3.2. Portable Expression Template Engine	128
8. Container Views	136
A. Obtaining and Installing POOMA	137
A.1. Supporting Distributed Computation	137
A.1.1. Obtaining and Installing the MM Shared Memory Library	137
A.1.2. Obtaining and Installing the Cheetah Messaging Library	138
A.1.3. Configuring POOMA When Using Cheetah	140
Glossary	141

List of Tables

2-1. Correspondences Between Run-Time and Compile-Time Constructs	18
2-2. More Correspondences Between Run-Time and Compile-Time Constructs	21
4-1. POOMA Concepts	66
4-2. POOMA Container Summary	67
4-3. Choosing a POOMA Container	69
5-1. Declaring One-Dimensional Locs	79
5-2. Declaring Multidimensional Locs	80
5-3. Declaring One-Dimensional Intervals	81
5-4. Declaring Multidimensional Intervals	82
5-5. Declaring One-Dimensional Ranges	83
5-6. Declaring Multidimensional Ranges	84
5-7. Declaring One-Dimensional Grids	86
5-8. Declaring Multidimensional Grids	87
5-9. Some Domain Accessors	88
5-10. Declaring Arrays	92
5-11. Initializing Arrays' Domains	95
5-12. Array Internal Type Definitions and Compile-Time Constants	100
5-13. Array Accessors	101
5-14. Changing a DynamicArray's Domain	104
6-1. Types of Engines	110
7-1. Operators Permissible for Data-Parallel Expressions	118
7-2. Mathematical Functions Permissible for Data-Parallel Expressions	119
7-3. Comparison Functions Permissible for Data-Parallel Expressions	124
7-4. Miscellaneous Functions Permissible for Data-Parallel Expressions	126

List of Figures

1-1. How POOMA Fits Into the Scientific Process	9
3-1. Doof2d Averagings	29
3-2. Adding Arrays	42
3-3. Applying a Stencil to an Array	46
3-4. The POOMA Distributed Computation Model	53
4-1. Concepts For Declaring Containers	69
4-2. Array and Field Mathematical and Computational Concepts	70
7-1. Adding Arrays with Different Domains	115
7-2. Annotated Parse Tree for $-A + 2 * B$	128

List of Examples

2-1. Classes Storing Pairs of Values	17
2-2. Templated Class Storing Pairs of Values	17
3-1. Hand-Coded Implementation of Doof2d	32
3-2. Element-wise Array Implementation of Doof2d	35
3-3. Data-Parallel Array Implementation of Doof2d	39
3-4. Stencil Array Implementation of Doof2d	43
3-5. Distributed Stencil Array Implementation of Doof2d	48
3-6. Data-Parallel Field Implementation of Doof2d	57
3-7. Distributed Data-Parallel Field Implementation of Doof2d.....	61
5-1. Copying Arrays.....	98
5-2. Using Array Member Functions	102
5-3. Example Using DynamicArrays	106

Acknowledgements

This book would not have been completed without the help and encouragement of a lot of people and organizations. Los Alamos National Laboratory funded the writing of this manual and the development of the POOMA Toolkit. John Reynders conceived, advocated, and headed POOMA development in its early days, and Scott Haney continued the leadership. Susan Atlas, Subhankar Banerjee, Timothy Cleland, Julian Cummings, James Crotinger, David Forslund, Salman Habib, Scott Haney, Paul Hinker, William Humphrey, Steve Karmesin, Graham Mark, Jeffrey D. Oldham, Ji Qiang, John Reynders, Robert Ryne, Stephen Smith, M. Srikant, Marydell Tholburn, and Timothy Williams all helped develop POOMA. Rod Oldehoeft and Jeff Brown of Los Alamos National Laboratory supported CodeSourcery's and Proximation's work, including the development of this manual. John Hall, Don Marshall, Jean Marshall, and the rest of the BLANCA team at Los Alamos worked closely with the developers and provided valuable suggestions for improvements.

I am grateful to James Crotinger, Mark Mitchell, and Stephen Smith who answered my many questions during the writing of this book.

Jeffrey D. Oldham, 2002 January

Chapter 1. Introduction

The Parallel Object-Oriented Methods and Applications (POOMA) Toolkit is a C++ toolkit for writing high-performance scientific programs. The toolkit provides a variety of tools:

- containers and other abstractions suitable for scientific computation,
- support for a variety of computation modes including data-parallel expressions, stencil-based computations, and lazy evaluation,
- support for writing parallel and distributed programs,
- automatic creation of all interprocessor communication for parallel and distributed programs, and
- automatic out-of-order execution and loop rearrangement for fast program execution.

Since the toolkit provides high-level abstractions, POOMA programs are much shorter than corresponding Fortran or C programs and require less time to write and less time to debug. Using these high-level abstractions, the same code runs on a sequential, parallel, and distributed computers. It runs almost as fast as carefully crafted machine-specific hand-written programs. The toolkit is open-source software, available for no cost, and compatible with any modern C++ compiler.

1.1. POOMA Goals

The goals for the POOMA Toolkit have remained unchanged since its conception in 1994:

1. Code portability across serial, distributed, and parallel architectures without any change to the source code.
2. Development of reusable, cross-problem-domain components to enable rapid application development.
3. Code efficiency for kernels and components relevant to scientific simulation.
4. Toolkit design and development driven by applications from a diverse set of scientific problem domains.
5. Shorter time from problem inception to working parallel simulations.

Below, we discuss how POOMA achieves these goals.

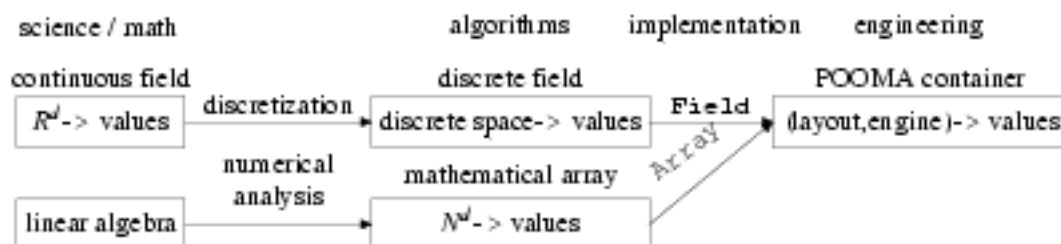
Code Portability for Sequential and Distributed Programs

The same POOMA programs run on sequential, distributed, and parallel computers. No change in source code is required. Two or three lines specify how each container's data should be distributed among available processors. Using these directives and run-time information about the computer's configuration, the toolkit automatically distributes pieces of the container domains, called *patches*, among the available processors. If a computation needs values from another patch, POOMA automatically passes the values to the patch where it is needed. The same program, and even the same executable, works regardless of the number of the available processors and the size of the containers' domains. A programmer interested in only sequential execution can omit the two or three lines specifying how the domains are to be distributed.

Rapid Application Development

The POOMA Toolkit is designed to enable rapid development of scientific and distributed applications. For example, its vector, matrix, and tensor classes model the corresponding mathematical concepts. Its `Array` and `Field` classes model the discrete spaces and mathematical arrays frequently found in computational science and math. See Figure 1-1. The left column indicates theoretical science and math concepts, the middle column computational science and math concepts, and the right column computer science implementations. For example, theoretical physics frequently uses continuous fields in three-dimension space, while algorithms for a corresponding computational physics problem usually uses discrete fields. POOMA containers, classes, and functions ease engineering computer programs for these algorithms. For example, the POOMA `Field` container models discrete fields: both map locations in discrete space to values and permit computations of spatial distances and values. The POOMA `Array` container models the mathematical concept of an array, frequently used in numerical analysis.

Figure 1-1. How POOMA Fits Into the Scientific Process



In the translation from theoretical science to computational science to computer programs, POOMA eases the implementation of algorithms as computer programs.

POOMA containers support a variety of computation modes, easing translation of algorithms into code. For example, many algorithms for solving partial differential equations use stencil-based computations so POOMA supports stencil-based computations on `Arrays` and `Fields`. POOMA also supports data-parallel computation similar to Fortran 90 syntax. To ease implementing computations where one `Field`'s values are a function of several other `Field`'s values, the programmer can specify a *relation*. Relations are lazily evaluated: whenever the dependent `Field`'s values are needed and they are dependent on a `Field` whose values have changed, the values are computed. Relations also assists correctness by eliminating the frequently forgotten need for a programmer to ensure a `Field`'s values are up-to-date before being used.

Efficient Code

POOMA incorporates a variety of techniques to ensure it produces code that executes as quickly as special-case, hand-written code. These techniques include extensive use of templates, out-of-order evaluation, use of guard layers, and production of fast inner loops.

POOMA's uses of C++ templates ensures as much as work as possible occurs at compile time, not run time. This speeds programs' execution. Since more code is produced at compile time, more code is available to the compiler's optimizer, further speeding execution. The POOMA `Array` container benefits from the use of template parameters. Their use permits the use of specialized data storage classes called *engines*. An `Array`'s `Engine` template parameter specifies how data is stored and indexed. Some `Arrays` expect almost all values to be used, while others might be mostly empty. In the latter case, using a specialized engine storing the few nonzero values greatly reduces storage requirements. Using engines also permits fast creation of container views, known as *array sections* in Fortran 90. A view's engine is the same as the original container's engine, but the view object's restricted domain is a subset of the original domain. Space requirements and execution time to use views are minimal.

Using templates also permits containers to support polymorphic indexing, e.g., indexing both by integers and by three-dimensional coordinates. A container uses templated indexing functions that defer indexing operations to its engine's index operators. Since the container uses templates, the `Engine` can define indexing functions with different function arguments, without the need to add corresponding container functions. Some of these benefits of using templates can be expressed without them, but doing so increases execution time. For example, a container could have a pointer to an engine object, but this requires a pointer dereference for each operation. Implementing polymorphic indexing without templates would require adding virtual functions corresponding to each of the indexing functions.

To ensure multiprocessor POOMA programs execute quickly, it is important that interprocessor communication overlaps with intraprocessor computations as much as possible and that communication is minimized. Asynchronous communication, out-of-order evaluation, and use of guard layers all help achieve these goals. POOMA uses the asynchronous communication facilities of the Cheetah communication library. When a processor needs data that is stored or computed by another processor, a message is sent between the two. If synchronous communication was used, the sender must issue an explicit send, and the recipient must issue an explicit receive, synchronizing the two processors. Cheetah permits the sender to put and get data without synchronizing with the recipient processor, and it also permits invoking functions at remote sites to ensure desired data is up-to-date. Thus, out-of-order evaluation must be supported. Out-of-order evaluation also has another benefit: Only computations directly or indirectly related to values that are printed need occur.

Surrounding a patch with *guard layers* can help reduce interprocessor communication. For distributed computation, each container's domain is split into pieces distributed among the available processors. Frequently, computing a container value is local, involving just the value itself and a few neighbors, but computing a value near the edge of a processor's domain may require knowing a few values from a neighboring domain. Guard layers permit these values to be copied locally so they need not be repeatedly communicated.

POOMA uses the PETE Library to ensure inner loops involving POOMA's object-oriented containers run as quickly as hand-coded loops. PETE (the Portable Expression Template Engine) uses expression-template technology to convert data-parallel statements into efficient loops without any intermediate computations. For example, consider evaluating the statement

```
A += -B + 2 * C;
```

where A and C are `vector<double>`s and B is a `vector<int>`. Naïve evaluation might introduce intermediaries for `-B`, `2*C`, and their sum. The presence of these intermediaries in inner loops can measurably slow performance. To produce a loop without intermediaries, PETE stores each expression as a parse tree. Using its templates, the parse tree is converted, at compile time, to a loop directly evaluating each component of the result without computing intermediate values. For example, the code corresponding to the statement above is

```
vector<double>::iterator iterA = A.begin();
vector<int>::const_iterator iterB = B.begin();
vector<double>::const_iterator iterC = C.begin();
```

```

while (iterA != A.end()) {
    *iterA += -*iterB + 2 * *iterC;
    ++iterA; ++iterB; ++iterC;
}

```

Furthermore, since the code is available at compile time, not run time, it can be further optimized, e.g., moving any loop-invariant code out of the loop.

Used for Diverse Set of Scientific Problems

POOMA has been used to solve a wide variety of scientific problems. Most recently, physicists at Los Alamos National Laboratory implemented an entire library of hydrodynamics codes as part of the U.S. government's science-based Stockpile Stewardship Program. Other applications include a matrix solver, an accelerator code simulating the dynamics of high-intensity charged particle beams in linear accelerators, and a Monte Carlo neutron transport code.

Easy Implementation

POOMA's tools greatly reduce the time to implement applications. As we noted above, POOMA's containers and expression syntax model the computational models and algorithms most frequently found in scientific programs. These high-level tools are known to be correct and reduce the time to debug programs. Since the same programs run on one processor and multiple processors, programmers can write and test programs using their one or two-processor personal computers. With no additional work, the same program runs on computers with hundreds of processors; the code is exactly the same, and the toolkit automatically handles distribution of the data, all data communication, and all synchronization. The net result is a significant reduction in programming time. For example, a team of two physicists and two support people at Los Alamos National Laboratory implemented a suite of hydrodynamics kernels in six months. Their work replaced a previous suite of less-powerful kernels which had taken sixteen people several years to implement and debug. Despite not have previously implemented any of the kernels, they implemented one new kernel every three days, including the time to read the corresponding scientific papers!

1.2. POOMA is Open-Source Software

The POOMA Toolkit is open-source software. Anyone may download, read, redistribute, and modify the POOMA source code. If an application requires a specialized container

not already available, any programmer may add it. Any programmer can extend it to solve problems in previously unsupported domains. Companies using the toolkit can read the source code to ensure it has no security holes. It may be downloaded at no cost and used for perpetuity. There are no annual licenses and no on-going costs. By keeping their own copies, companies are guaranteed the software will never disappear. In summary, the POOMA Toolkit is low-risk software.

1.3. History of POOMA

The POOMA Toolkit was developed at Los Alamos National Laboratory to assist nuclear fusion and fission research. In 1994, the toolkit grew out of the Object-Oriented Particle Simulation Class Library developed for particle-in-cell simulations. The goals of the Framework, as it was called at the time, were driven by the Numerical Tokamak's "Parallel Platform Paradox":

The average time required to implement a moderate-sized application on a parallel computer architecture is equivalent to the half-life of the latest parallel supercomputer.

The framework's goal of being able to quickly write efficient scientific code that could be run on a wide variety of platforms remains unchanged today. Development, mainly at the Advanced Computing Laboratory at Los Alamos, proceeded rapidly. A matrix solver application was written using the framework. Support for hydrodynamics, Monte Carlo simulations, and molecular dynamics modeling soon followed.

By 1998, POOMA was part of the U.S. Department of Energy's Accelerated Strategic Computing Initiative (ASCI). The Comprehensive Test Ban Treaty forbid nuclear weapons testing so they were instead simulated using computers. ASCI's goal was to radically advance the state of the art in high-performance computing and numerical simulations so the nuclear weapon simulations could use 100-teraflop parallel computers. A linear accelerator code `linac` and a Monte Carlo neutron transport code `MC++` were among the codes written.

POOMA 2 involved a new conceptual framework and a complete rewriting of the source code to improve performance. The `Array` class was introduced with its use of `Engines`, separating container use from container storage. A new asynchronous scheduler permitted out-of-order execution to improve cache coherency. Incorporating the Portable Expression Template Engine (PETE) permitted faster loop execution. Soon, container views and `ConstantFunction` and `IndexFunction` Engines were added. Release 2.1.0 included `Fields` with their spatial extent and `Dynami-cArrays` with the ability to dynamically change domain size. Support for particles

and their interaction with `Fields` were added. The POOMA messaging implementation was revised in release 2.3.0. Use of the Cheetah Library separated POOMA from the actual messaging library used, and support for applications running on clusters of computers was added. CodeSourcery, LLC (<http://www.codesourcery.com/>), and Proximation, LLC (<http://www.proximation.com/>), took over POOMA development from Los Alamos National Laboratory. During the past two years, the `Field` abstraction and implementation was improved to increase its flexibility, add support for multiple values and materials in the same cell, and permit lazy evaluation. Simultaneously, the execution speed of the inner loops was greatly increased.

Chapter 2. Programming with Templates

POOMA extensively uses C++ *templates* to support type polymorphism without incurring any run-time cost. In this chapter, we briefly introduce using templates in C++ programs by relating them to “ordinary” C++ constructs such as values, objects, and classes. The two main concepts underlying C++ templates will occur repeatedly:

- Template programming constructs execute at compile time, not run time. That is, template operations occur within the compiler, not when a program runs.
- Templates permit declaring families of classes using a single declaration. For example, the `Array` template declaration permits using `Arrays` with many different value types, e.g., arrays of integers, arrays of floating point numbers, and arrays of arrays.

For those interested in the implementation of POOMA, we close the section with a discussion of some template programming concepts used in the implementation but not likely to be used by POOMA users.

2.1. Templates Execute at Compile-Time

POOMA uses C++ templates to support type polymorphism without incurring any run-time cost as a program executes. All template operations are performed at compile time by the compiler.

Prior to the introduction of templates, almost all of a program’s interesting computation occurred when it was executed. When writing the program, the programmer, at *programming time*, would specify which statements and expressions will occur and which types to use. At *compile time*, the compiler would convert the program’s source code into an executable program. Even though the compiler uses the types to produce the executable, no interesting computation would occur. At *run time*, the resulting executable program would actually perform the operations.

The introduction of templates permits interesting computation to occur while the compiler produces the executable. Most interesting is template instantiation, which produces a type at compile time. For example, the `Array` “type” definition requires template parameters `Dim`, `T`, and `EngineTag`, specifying its dimension, the type of its values,

and its `Engine` type. To use this, a programmer specifies values for the template parameters: `Array<2, double, Brick>` specifies a dimension of 2, a value type of `double`, and the `Brick` `Engine` type. At compile time, the compiler creates a type definition by substituting the values for the template parameters in the templated type definition. The substitution is analogous to the run-time application of a function to specific values.

All computation not involving run-time input or output can occur at programming time, compile time, or run time, whichever is more convenient. At programming time, a programmer can perform computations by hand rather than writing code to compute it. C++ templates are Turing-complete so they can compute anything computable. Unfortunately, syntax for compile-time computation is more difficult than for run-time computation. Also current compilers are not as efficient as code executed by hardware. Run-time C++ constructs are Turing-complete so using templates is unnecessary. Thus, we can shift computation to the time which best trades off the ease of expressing syntax with the speed of computation by programmer, compiler, or computer chip. For example, POOMA uses expression template technology to speed run-time execution of data-parallel statements. The POOMA developers decided to shift some of the computation from run-time to compile-time using template computations. The resulting run-time code runs more quickly, but compiling the code takes longer. Also, programming time for the POOMA developers increased significantly, but, most users, who are usually most concerned about decreasing run times, benefited.

2.2. Template Programming for POOMA Users

Most POOMA users need only understand a subset of available constructs for template programming. These constructs include

- reading template declarations and understanding template parameters, both of which are used in this book.
- template instantiation, i.e., specifying a particular type by specifying values for template parameters.
- nested type names, which are types specified within a class definition.

We discuss each of these below.

Templates generalize writing class declarations by permitting class declarations dependent on other types. For example, consider writing a class storing a pair of integers and a class storing a pair of doubles. See Example 2-1. Almost all of the code for the two

definitions is the same. Both of these definitions define a class with a constructor and storing two values named `left` and `right` having the same type. Only the classes' names and its use of types differ.

Example 2-1. Classes Storing Pairs of Values

```
// Declare a class storing a pair of integers.
struct pairOfInts {
    pairOfInts(const int& left, const int& right)
        : left_(left), right_(right) {}

    int left_;
    int right_;
};

// Declare a class storing a pair of doubles.
struct pairOfDoubles {
    pairOfDoubles(const double& left, const double& right)
        : left_(left), right_(right) {}

    double left_;
    double right_;
};
```

Using templates, we can use a template parameter to represent their different uses of types and write one templated class definition. See Example 2-2. The templated class definition is a copy of the common portions of the two preceding definitions. Because the two definitions differ only in their use of the `int` and `double` types, we replace these concrete types with a template parameter `T`. We *precede*, not follow, the class definition with `template <typename T>`. The constructor's parameters' types are changed to `T`, as are the data members' types.

Example 2-2. Templated Class Storing Pairs of Values

```
// Declare a template class storing a pair of values
// with the same type.
template <typename T> // (1)
```

```

struct pair {
    pair(const T& left, const T& right)  // (2)
        : left_(left), right_(right) {}

    T left_;  // (3)
    T right_;
};

// Use a class storing a pair of integers. (4)
pair<int> pair1;

// Use a class storing a pair of doubles;
pair<double> pair2;

```

- (1) Template parameters are written before, not after, a class name.
- (2) The constructor has two parameters of type `const T&`.
- (3) An object stores two values having type `T`.
- (4) To use a templated class, specify the template parameter's argument after the class's name and surrounded by angle brackets (`<>`).

To use a template class definition, template arguments follow the class name surrounded by angle brackets (`<>`). For example, `pair<int>` *instantiates* the `pair` template class definition with `T` equal to `int`. That is, the compiler creates a definition for `pair<int>` by copying `pair`'s template definition and substituting `int` for each occurrence of `T`. The copy omits the template parameter declaration `template <typename T>` at the beginning of its definition. The result is a definition exactly the same as `pairOfInts`.

As we mentioned above, template instantiation is analogous to function application. A template class is analogous to a function; it is a function from types and constants to classes. The analogy between compile-time and run-time programming constructs can be extended. Table 2-1 lists these correspondences. For example, at run time, values consist of things such as integers, floating point numbers, pointers, functions, and objects. Programs compute by operating on these values. The compile-time values include types, and compile-time operations use these types. For both run-time and compile-time programming, C++ defines default sets of values that all conforming compilers must support. For example, 3 and `6.022e+23` are run-time values that any C++ compiler must accept. It must also accept the `int`, `bool`, and `int*` types.

Table 2-1. Correspondences Between Run-Time and Compile-Time Constructs

programming construct	run time	compile time
values	integers, strings, objects, functions, ...	types, ...
create a value to store multiple values	object creation	class definition
values stored within a collection	data member, member function	nested type name, nested class, static member function, constant integral values
placeholder for “any particular value”	variable, e.g., “any int”	template argument, e.g., “any type”
packaging repeated operations	A function generalizes a particular operation applied to different values. The function parameters are placeholders for particular values.	A template class generalizes a particular class definition using different types. The template parameters are placeholders for particular values.
application	Use a function by appending function arguments surrounded by parentheses.	Use a template class by appending template arguments surrounded by angle brackets (<>).

The set of supported run-time and compile-time values can be extended. Run-time values can be extended by creating new objects. Although not part of the default set of values, these objects are treated and operated on as values. To extend the set of compile-time values, class definitions are written. For example, Example 2-1 declares two new types `pairOfInts` and `pairOfDoubles`. Although not part of the set of built-in types, these types can be used in the same way that any other types can be used, e.g., declaring variables.

Functions generalize similar run-time operations, while template class generalize similar class definitions. A function definition generalizes a repeated run-time operation. For example, consider repeatedly printing the largest of two numbers:

```
std::cout << (3 > 4 ? 3 : 4) << std::endl;
std::cout << (4 > -13 ? 4 : -13) << std::endl;
```

```
std::cout << (23 > 4 ? 23 : 4) << std::endl;
std::cout << (0 > 3 ? 0 : 3) << std::endl;
```

Each statement is exactly the same except for the repeated two values. Thus, we can generalize these statements writing a function:

```
void maxOut(int a, int b)
{ std::cout << (a > b ? a : b) << std::endl; }
```

The function's body consists of the statement with variables substituted for the two particular values. Each parameter variable is a placeholder that, when used, holds one particular value among the set of possible integral values. The function must be named to permit its use, and declarations for its two parameters follow. Using the function simplifies the code:

```
maxOut(3, 4);
maxOut(4, -13);
maxOut(23, 4);
maxOut(0, 3);
```

To use a function, the function's name precedes parentheses surrounding specific values for its parameters, but the function's return type is omitted.

A template class definition generalizes repeated class definitions. If two class definitions differ only in a few types, template parameters can be substituted. Each parameter is a placeholder that, when used, holds one particular value, i.e., type, among the set of possible values. The class definition is named to permit its use, and declarations for its parameters precede it. The example found in the previous section illustrates this transformation. Compare the original, untemplated classes in Example 2-1 with the templated class in Example 2-2. Note the notation for the template class parameters. `template <typename T>` *precedes* the class definition. The keyword `typename` indicates the template parameter is a type. `T` is the template parameter's name. (We could have used any other identifier such as `pairElementType` or `foo`.) Note that using `class` is equivalent to using `typename` so `template <class T>` is equivalent to `template <typename T>`. While declaring a template class requires prefix notation, using a templated class requires postfix notation. The class's name precedes angle brackets (`<>`) surrounding specific values, i.e., types, for its parameters. As we showed above, `pair<int>` *instantiates* the template class `pair` with `int` for its type parameter `T`.

In template programming, nested type names store compile-time data that can be used within template classes. Since compile-time class definitions are analogous to run-time objects and the latter stores named values, nested type names are values, i.e., types, stored within class definitions. For example, the template class `Array` has an nested type name for the type of its domain:

```
typedef typename Engine_t::Domain_t Domain_t;
```

This typedef, i.e., type definition, defines the type `Domain_t` as equivalent to `Engine_t::Domain_t`. The `::` operator selects the `Domain_t` nested type from inside the `Engine_t` type. This illustrates how to access `Array`'s `Domain_t` when not within `Array`'s scope: `Array<Dim, T, Engine-Tag>::Domain_t`. The analogy between object members and nested type names alludes to its usefulness. Just as run-time object members store information for later use, nested type names store type information for later use at compile time. Using nested type names has no impact on the speed of executing programs.

2.3. Template Programming Used to Write POOMA

The preceding section presented template programming tools needed to read this book and write programs using the POOMA Toolkit. In this section, we present template programming techniques used to implement POOMA. We extend the correspondence between compile-time template programming constructs and run-time constructs started in the previous section. Reading this section is not necessary unless you wish to understand how POOMA is implemented.

In the previous section, we used a correspondence between run-time and compile-time programming constructs to introduce template programming concepts, which occur at compile time. See Table 2-1. In implementing POOMA, more constructs are used. We list these in Table 2-2.

Table 2-2. More Correspondences Between Run-Time and Compile-Time Constructs

programming construct	run time	compile time
values	integers, strings, objects, functions, ...	types, constant integers and enumerations, pointers and references to objects and functions, executable code, ...
operations on values	Integral values support +, -, >, ==, String values support [], ==,	Types may be declared and used. Constant integral and enumeration values can be combined using +, -, >, ==, There are no permitted operations on code.
values stored in a collection	An object stores values.	A <i>traits class</i> contains values describing a type.
extracting values from collections	An object's named values are extracted using the . operator.	A class's nested types and classes are extracted using the :: operator.
control flow to choose among operations	if, while, goto, ...	template class specializations with pattern matching

The only compile-time values described in the previous section were types, but any compile-time constant can also be used. Integral literals, `const` variables, and other constructs can be used, but the main use is enumerations. An *enumeration* is a distinct integral type with named constants. For example, the `Array` declaration declares two separate enumerations:

```
template<int Dim, class T, class EngineTag>
class Array
{
public:
    typedef Engine<Dim, T, EngineTag> Engine_t;
    enum { dimensions = Engine_t::dimensions };
    enum { rank = Engine_t::dimensions };
    ...
}
```

The first enumeration declares the constant `dimensions` to be equal to the value of

the `dimensions` within the `Array`'s `Engine`. The second enumeration declares the constant `rank` to have the same value. Semantically, both indicate the dimensionality of the array's domain. Enumeration constants have integral values so they may be used wherever integers can be used. For example,

```
enum { dimensionPlusRank = dimensions + rank };
```

could be added to the `Array` declaration. Declaring an enumeration is a compile-time construct analogous to assigning an integral value to a variable at run time. Note that an enumerated constant's value cannot be changed.

Enumerations are frequently used in template programming because

- an enumeration declares a new type, which ensures it is available at compile time and
- constant integral values, and thus enumerated constants, can be used in all compile-time expressions and as template arguments.

The use of non-integral constant values such as floating-point numbers at compile time is restricted.

Other compile-time values include pointers to objects and functions, references to objects and functions, and executable code. For example, a pointer to a function sometimes is passed to a template function to perform a specific task. Even though executable code cannot be directly represented in a program, it is a compile-time value which the compiler uses. A simple example is a class that is created by template instantiation, e.g., `pair<int>`. Conceptually, the `int` template argument is substituted throughout the `pair` template class to produce a class definition. Although neither the programmer nor the user sees this class definition, it is represented inside the compiler, which can use and manipulate the code.

Through template programming, the compiler's optimizer can transform complicated code into much simpler code. In Section 7.3, we describe the complicated template code used to implement efficiently data-parallel operations. Although the template code is complicated, the compiler optimization frequently greatly simplifies it to yield simple, fast loops. We illustrate this with a simple template class:

```
template <bool complicatedCase>
struct usuallySimpleClass {
    usuallySimpleClass() {
        if (complicatedCase)
            i = do_some_very_complicated_computation();
        else
```

```

        i = 0;
    }
    int i;
};

```

The `usuallySimpleClass` has one boolean template parameter `complicatedCase`, which should be true only if the constructor must perform some very complicated, time-expensive computation. When instantiated with `false`, the compiler substitutes this value into the template class definition. Since the `if` statement's conditional is false, the compiler optimizer can eliminate the statement, yielding internal code similar to

```

struct usuallySimpleClass<false> {
    usuallySimpleClass() {
        i = 0;
    }
    int i;
};

```

The optimizer might further simplify the code by inlining the constructor's assignment. Because the resulting code is never displayed, the programmer does not know how simplified it is without investigating the resulting assembly code. C++ compilers that translate C++ code into C code may permit inspecting the resulting code. For example, using the `--keep_gen_c` command-line option with the KAI C++ compiler creates a file containing the intermediate code. Unfortunately, reading and understanding the code is frequently difficult.

Each category of values supports a distinct set of operations. For example, the run-time category of integer values supports combination using `+` and `-` and comparison using `>` and `==`. At run time, the category of strings can be compared using `==` and characters can be extracted using subscripts with the `[]` operator. Compile-time operations are more limited. Types may be declared and used. The `sizeof` operator yields the number of bytes to represent an object of the specified type. Enumerations, constant integers, `sizeof` expressions, and simple arithmetic and comparison operators such as `+` and `==` can form constant expressions that can be used at compile time. These values can initialize enumerations and integer constants and be used as template arguments. At compile time, pointers and references to objects and functions can be used as template arguments, while the category of executable code supports no operations. (The compiler's optimizer may simplify it, though.)

At run time, an object can store multiple values, each having its own name. For example, a `pair<int>` object `p` stores two ints named `left_` and `right_`. The `.` operator extracts a named member from an object: `p.left_`. At compile time, a class can store multiple values, each having its own name. These are sometimes called *traits classes*. For example, implementing data-parallel operations requiring storing a tree of types. The `ExpressionTraits<BinaryNode<Op, Left, Right>>` traits class stores the types of a binary node representing the operation of `Op` on left and right children. Its definition

```
template<class Op, class Left, class Right>
struct ExpressionTraits<BinaryNode<Op, Left, Right>>
{
    typedef typename ExpressionTraits<Left>::Type_t Left_t;
    typedef typename ExpressionTraits<Right>::Type_t Right_t;
    typedef typename
        CombineExpressionTraits<Left_t, Right_t>::Type_t Type_t;
};
```

consists of a class definition and internal type definitions. This traits class contains three values, all types and named `Left_t`, `Right_t`, and `Type_t`, representing the type of the left child, the right child, and the entire node, respectively. Many traits classes, such as this one, use internal type definitions to store values. No enumerations or constant values occur in this traits class, but other such classes include them. See Section 7.3 for more details regarding the implementation of data-parallel operators.

The example also illustrates using the `::` operator to extract a member of a traits class. The type `ExpressionTraits<Left>` contains an internal type definition of `Type_t`. Using the `::` operator extracts it: `ExpressionTraits<Left>::Type_t`. Enumerations and other values can also be extracted. For example, `Array<2, int, Brick>::dimensions` yields the dimension of the array's domain.

Control flow determines which code is used. At run time, control-flow statements such as `if`, `while`, and `goto` determine which statements to execute. Template programming uses two mechanisms: template class specializations and pattern matching. These are similar to control flow in functional programming languages. A *template class specialization* is a class definition specific to one or more template arguments. For example, the implementation for data-parallel operations uses the templated `CreateLeaf`. The default definition works for any template argument `T`:

```
template<class T>
struct CreateLeaf
{
    typedef Scalar<T> Leaf_t;
    ...
};
```

The code is different for Expression specializations:

```
template<class T>
struct CreateLeaf<Expression<T>>
{
    typedef typename Expression<T>::Expression_t Leaf_t;
    ...
};
```

The latter code is only used when `CreateLeaf`'s template argument is an `Expression` type.

Pattern matching of template arguments to template parameters determines which template code is used. The code associated with the match that is most specific is the one that is used. For example, `CreateLeaf<int>` uses the first, more general template class definition because the `int` template argument does not match `Expression<T>` for any value of `T`. On the other hand, `CreateLeaf<Expression<int>>` uses the second definition because both the general and the specialized template parameters match so the more specialized ones are preferred. In this case, `T` equals `int`. `CreateLeaf<Expression<Expression<int>>>` also matches the more specialized definition with `T` equaling `Expression<int>`.

Control flow using template specializations and pattern matching is similar to `switch` statements. A `switch` statement has a condition and one or more pairs of case labels and associated code. The code associated with the the case label whose value matches the condition is executed. If no case label matches the condition, the default code, if present, is used. In template programming, instantiating a template, e.g.,

```
CreateLeaf<Expression<int>>
```

serves as the condition. The set of template parameters for the indicated template class, e.g., `CreateLeaf`, are analogous to the case labels, and each has an associated definition. In our example, the set of template parameters are `<class T>`

and `<Expression<class T>>`. The “best match”, if any, indicates the matching code that will be used. In our example, the `<class T>` parameter serves as the default label since it matches any arguments. If no set of template parameters match (which is impossible for our example) or if more than one set are best matches, the code is incorrect.

Functions as well as classes may be templated. All the concepts needed to understand function templates have already been introduced so we illustrate using an example. The templated function `f` takes one parameter of any type:

```
template <typename T>
void f(const T& t) { ... }
```

A *function template* defines an unbounded set of related functions, all with the same name. Our example defines functions equivalent to `f(const int&)`, `f(const bool&)`, `f(const int*&)`, Using a templated class definition with a static member function, we can define an equivalent function:

```
template <typename T>
class F {
    static void f(const T& t) { ... }
};
```

Both the templated class and the templated function take the same template arguments, but the class uses a static member function. Thus, the notation to invoke it is slightly more verbose: `F<T>::f(t)`.

The advantage of a function template is that it can be overloaded, particularly operator functions. For example, the `+` operator is overloaded to add two `Arrays`, which require template parameters to specify:

```
template <int D1, class T1, class E1,
          int D2, class T2, class E2>
// complicated return type omitted
operator+(const Array<D1, T1, E1> & l,
          const Array<D2, T2, E2> & r);
```

Without using function templates, it would not be possible to write expressions such as `a1 + a2`. Member functions can also be templated. This permits, for example, overloading of assignment operators defined within templated classes.

Function objects are frequently useful in run-time code. They consist of a function plus some additional storage and are usually implemented as structures with data members and a function call operator. Analogous classes can be used at compile time. Using the transformation introduced in the previous paragraph, we see that any function can be transformed into a class containing a static member function. Internal type definitions, enumerations, and static constant values can be added to the class. The static member function can use these values during its computation. The `CreateLeaf` structure, introduced above, illustrates this.

```
template<class T>
struct CreateLeaf
{
    typedef Scalar<T> Leaf_t;
    inline static Leaf_t make(const T& a)
        { return Scalar<T>(a); }
};
```

Thus, `CreateLeaf<T>::make` is a function with a complicated name and having access to the class member named `Leaf_t`. Unlike for function objects, the function's name within the class must be given a name.

Chapter 3. A Tutorial Introduction

POOMA provides different containers and processor configurations and supports different implementation styles, as described in Section 1.1. In this chapter, we present several different implementations of the `Doof2d` two-dimensional diffusion simulation program:

- a C-style implementation omitting any use of POOMA and computing each array element individually,
- a POOMA `Array` implementation computing each array element individually,
- a POOMA `Array` implementation using data-parallel statements,
- a POOMA `Array` implementation using stencils, which support local computations,
- a stencil-based POOMA `Array` implementation supporting computation on multiple processors
- a POOMA `Field` implementation using data-parallel statements, and
- a data-parallel POOMA `Field` implementation for multiprocessor execution.

These illustrate the `Array`, `Field`, `Engine`, `layout`, `mesh`, and `Domain` data types. They also illustrate various immediate computation styles (element-wise accesses, data-parallel expressions, and stencil computation) and various processor configurations (one processor and multiple processors).

The `Doof2d` diffusion program starts with a two-dimensional grid of values. To model an initial density, all grid values are zero except for one nonzero value in the center. Each averaging, each grid element, except the outermost ones, updates its value by averaging its value and its eight neighbors. To avoid overwriting grid values before all their uses occur, we use two arrays, reading the first and writing the second and then reversing their roles within each iteration.

We illustrate the averagings in Figure 3-1. Initially, only the center element has nonzero value. To form the first averaging, each element's new value equals the average of its and its neighbors' previous values. Thus, the initial nonzero value spreads to a three-by-three grid. The averaging continues, spreading to a five-by-five grid of nonzero values. Values in the outermost grid cells are always zero.

Figure 3-1. Doof2d Averagings

Array b: Initial Configuration						
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	1000.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
Array a: After the first averaging						
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	111.1	111.1	111.1	0.0	0.0
0.0	0.0	111.1	111.1	111.1	0.0	0.0
0.0	0.0	111.1	111.1	111.1	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0

Array b: After the second averaging

0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	12.3	24.7	37.0	24.7	12.3	0.0
0.0	24.7	49.4	74.1	49.4	24.7	0.0
0.0	37.0	74.1	111.1	74.1	37.0	0.0
0.0	24.7	49.4	74.1	49.4	24.7	0.0
0.0	12.3	24.7	37.0	24.7	12.3	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0

Before presenting the various implementations of Doof2d, we explain how to install the POOMA Toolkit.

3.1. Installing POOMA

In this section, we describe how to obtain, build, and install the POOMA Toolkit. We focus on installing under a Unix-like operating system.

Obtain the POOMA source code `pooma-2.3.0.tgz` from the POOMA download page (<http://pooma.codesourcery.com/pooma/download>) available off the POOMA home page (<http://www.codesourcery.com/pooma/pooma/>). The “tgz” indicates this is a compressed tar archive file. To extract the source files, use `tar xzvf pooma-2.3.0.tgz`. Move into the source code directory `pooma-2.3.0` directory; e.g., `cd pooma-2.3.0`.

Configuring the source code determines file names needed for compilation. First, determine a configuration file in the `config/arch/` directory corresponding to your operating system and compiler. For example, `LINUXgcc.conf` supports compiling under a Linux operating system with `g++`, while `SGI64KCC.conf` supports compiling under a 64-bit SGI Irix operating system with `KCC`. Next, configure the source code: `./configure --arch LINUXgcc --opt --suite LINUXgcc-opt`. The architecture argument to the `--arch` option is the name of the corresponding configuration file, omitting its `.conf` suffix. The `--opt` indicates the POOMA Toolkit will contain optimized source code, which makes the code run more quickly but may impede debugging. Alternatively, use the `--debug` option which supports debugging. The *suite name* can be any arbitrary string. We chose `LINUXgcc-`

`opt` to remind us of the architecture and optimization choice. `configure` creates sub-directories named “`LINUXgcc-opt`” for use when compiling the source files. Comments at the beginning of `lib/suiteName/PoomaConfiguration.h` record the configuration arguments.

To compile the source code, set the `POOMASUITE` environment variable to the suite name and then type `make`. To set the environment variable for the bash shell use `export POOMASUITE=suiteName`, substituting the suite name’s `suiteName`. For the `csh` shell, use `setenv POOMASUITE LINUXgcc-opt`. Issuing the `make` command compiles the POOMA source code files to create the POOMA library. The POOMA makefiles assume the GNU™ Make is available so substitute the proper command to run GNU™ Make if necessary. The POOMA library can be found in, e.g., `lib/LINUXgcc-opt/libpooma-gcc.a`.

3.2. Hand-Coded Implementation

Before implementing `Doof2d` using the POOMA Toolkit, we present a hand-coded implementation of `Doof2d`. See Example 3-1. After querying the user for the number of averagings, the arrays’ memory is allocated. Since the arrays’ size is not known at compile time, the arrays are accessed via pointers to allocated dynamic memory. This memory is deallocated at the program’s end to avoid memory leaks. The arrays are initialized with initial conditions. For the `b` array, all values except the central ones have nonzero values. Only the outermost values of the `a` array need be initialized to zero, but we instead initialize them all using the same loop initializing `b`.

The simulation’s kernel consists of triply nested loops. The outermost loop controls the number of iterations. The two inner nested loops iterate through the arrays’ elements, excepting the outermost elements; note the loop indices range from 1 to `n-2` while the array indices range from 0 to `n-1`. Each `a` value is assigned the average of its corresponding value in `b` and the latter’s neighbors. Values in the two-dimensional grids are accessed using two sets of brackets, e.g., `a[i][j]`. After assigning values to `a`, a second averaging reads values in `a`, writing values in `b`.

After the kernel finishes, the final central value is printed. If the desired number of averagings is even, the value in `b` is printed; otherwise, the value in `a` is used. Finally, the dynamically-allocated memory must be freed to avoid memory leaks.

Example 3-1. Hand-Coded Implementation of `Doof2d`


```

#include <iostream>    // has std::cout, ...
#include <stdlib.h>    // has EXIT_SUCCESS

// Doof2d: C-like, element-wise implementation

int main()
{
    // Ask the user for the number of averagings.  (1)
    long nuAveragings, nuIterations;
    std::cout << "Please enter the number of averagings: ";
    std::cin >> nuAveragings;
    nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.

    // Use two-dimensional grids of values.  (2)
    double **a;
    double **b;

    // Ask the user for the number n of values along one
    // dimension of the grid.  (3)
    long n;
    std::cout << "Please enter the array size: ";
    std::cin >> n;

    // Allocate the arrays.  (4)
    typedef double* doublePtr;
    a = new doublePtr[n];
    b = new doublePtr[n];
    for (int i = 0; i < n; i++) {
        a[i] = new double[n];
        b[i] = new double[n];
    }

    // Set up the initial conditions.
    // All grid values should be zero except for the
    // central value.  (5)
    for (int j = 0; j < n; j++)
        for (int i = 0; i < n; i++)
            a[i][j] = b[i][j] = 0.0;
    b[n/2][n/2] = 1000.0;

```

```

// Average using this weight.  (6)
const double weight = 1.0/9.0;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b.  Write to a.  (7)
    for (int j = 1; j < n-1; j++)
        for (int i = 1; i < n-1; i++)
            a[i][j] = weight *
                (b[i+1][j+1] + b[i+1][j] + b[i+1][j-1] +
                 b[i][j+1] + b[i][j] + b[i][j-1] +
                 b[i-1][j+1] + b[i-1][j] + b[i-1][j-1]);

    // Read from a.  Write to b.  (8)
    for (int j = 1; j < n-1; j++)
        for (int i = 1; i < n-1; i++)
            b[i][j] = weight *
                (a[i+1][j+1] + a[i+1][j] + a[i+1][j-1] +
                 a[i][j+1] + a[i][j] + a[i][j-1] +
                 a[i-1][j+1] + a[i-1][j] + a[i-1][j-1]);
}

// Print out the final central value.  (9)
std::cout <<
    (nuAveragings % 2 ? a[n/2][n/2] : b[n/2][n/2])
    << std::endl;

// Deallocate the arrays.  (10)
for (int i = 0; i < n; i++) {
    delete [] a[i];
    delete [] b[i];
}
delete [] a;
delete [] b;

return EXIT_SUCCESS;
}

```

- (1) The user specifies the desired number of averagings.
- (2) These variables point to the two-dimensional, dynamically-allocated grids so we use a pointer to a pointer to a `double`.
- (3) The user enters the desired grid size. The grid will be a square with `n` by `n` grid cells.
- (4) Memory for the arrays is allocated. By default, the array indices are zero-based.
- (5) Initially, all grid values are zero except for the one nonzero value at the center of the second array. Array positions are indicated using two brackets, e.g., `a[i][j]`. A better implementation might initialize only the outermost values of the `a` array.
- (6) This constant indicates the average's weighting.
- (7) Each `a` value, except an outermost one, is assigned the average of its analogous `b` value and that value's neighbors. Note the loop indices ensure the outermost values are not changed. The `weight`'s value ensures the computation is an average.
- (8) The second averaging computes `b`'s values using values stored in `a`.
- (9) After the averagings finish, the central value is printed.
- (10) The dynamically-allocated memory must be deallocated to avoid memory leaks.

To compile the executable, change directories to the POOMA `examples/Manual/Doof2d` directory. Ensure the `POOMASUITE` environment variable specifies the desired suite name `suiteName`, as we did when compiling POOMA in Section 3.1. Issuing the `make Doof2d-C-element` command creates the executable `suiteName/Doof2d-C-element`.

When running the executable, specify the desired nonnegative number of averagings and the nonnegative number of grid cells along any dimension. The resulting grid has the same number of cells along each dimension. After the executable finishes, the resulting value of the central element is printed.

3.3. Element-wise Array Implementation

The simplest way to use the POOMA Toolkit is to use the POOMA `Array` class instead of C arrays. `Arrays` automatically handle memory allocation and deallocation, support a wider variety of assignments, and can be used in expressions. Example 3-2 implements `Doof2d` using `Arrays` and element-wise accesses. Since the same algorithm is used as Example 3-1, we will concentrate on the differences.

Example 3-2. Element-wise Array Implementation of Doof2d

```

#include <iostream>    // has std::cout, ...
#include <stdlib.h>    // has EXIT_SUCCESS
#include "Pooma/Arrays.h"
    // has POOMA's Array declarations    (1)

// Doof2d: POOMA Arrays, element-wise implementation

int main(int argc, char *argv[])
{
    // Prepare the POOMA library for execution.    (2)
    Pooma::initialize(argc,argv);

    // Ask the user for the number of averagings.
    long nuAveragings, nuIterations;
    std::cout << "Please enter the number of averagings: ";
    std::cin >> nuAveragings;
    nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.

    // Ask the user for the number n of values along
    // one dimension of the grid.
    long n;
    std::cout << "Please enter the array size: ";
    std::cin >> n;

    // Specify the arrays' domains [0,n) x [0,n).    (3)
    Interval<1> N(0, n-1);
    Interval<2> vertDomain(N, N);

    // Create the arrays.    (4)
    // The Array template parameters indicate
    // 2 dimensions, a 'double' value
    // type, and ordinary 'Brick' storage.
    Array<2, double, Brick> a(vertDomain);
    Array<2, double, Brick> b(vertDomain);

    // Set up the initial conditions.

```

```

// All grid values should be zero except for the
// central value.  (5)
for (int j = 1; j < n-1; j++)
    for (int i = 1; i < n-1; i++)
        a(i,j) = b(i,j) = 0.0;
b(n/2,n/2) = 1000.0;

// In the average, weight elements with this value.
const double weight = 1.0/9.0;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b.  Write to a.
    for (int j = 1; j < n-1; j++)
        for (int i = 1; i < n-1; i++)
            a(i,j) = weight * (6)
                (b(i+1,j+1) + b(i+1,j ) + b(i+1,j-1) +
                 b(i ,j+1) + b(i ,j ) + b(i ,j-1) +
                 b(i-1,j+1) + b(i-1,j ) + b(i-1,j-1));

    // Read from a.  Write to b.
    for (int j = 1; j < n-1; j++)
        for (int i = 1; i < n-1; i++)
            b(i,j) = weight *
                (a(i+1,j+1) + a(i+1,j ) + a(i+1,j-1) +
                 a(i ,j+1) + a(i ,j ) + a(i ,j-1) +
                 a(i-1,j+1) + a(i-1,j ) + a(i-1,j-1));
}

// Print out the final central value.
Pooma::blockAndEvaluate();
    // Ensure all computation has finished.
std::cout <<
    (nuAveragings % 2 ? a(n/2,n/2) : b(n/2,n/2))
    << std::endl;

// The arrays are automatically deallocated.  (7)

// Tell the POOMA library execution finished.  (8)
Pooma::finalize();

```

```

    return EXIT_SUCCESS;
}

```

- (1) To use POOMA Arrays, the `Pooma/Arrays.h` must be included.
- (2) The POOMA Toolkit structures must be constructed before their use.
- (3) Before creating an `Array`, its domain must be specified. The `NInterval` represents the one-dimensional integral set $\{0, 1, 2, \dots, n-1\}$. The `Interval<2>` `vertDomain` object represents the entire two-dimensional index domain.
- (4) An `Array`'s template parameters indicate its dimension, its value type, and how the values will be stored or computed. The `Brick Engine` type indicates values will be directly stored. It is responsible for allocating and deallocating storage so `new` and `delete` statements are not necessary. The `vertDomain` specifies the array index domain.
- (5) The first loop initializes all `Array` values to the same scalar value. The second statement illustrates assigning one `Array` value. Indices, separated by commas, are surrounded by parentheses rather than surrounded by square brackets (`[]`).
- (6) `Array` element access uses parentheses, rather than square brackets.
- (7) The `Arrays` deallocate any memory they require, eliminating memory leaks.
- (8) The POOMA Toolkit structures must be destructed after their use.

We describe the use of `Array` and the POOMA Toolkit in Example 3-2. `Arrays`, declared in the `Pooma/Arrays.h`, are first-class objects. They “know” their index domain, can be used in expressions, can be assigned scalar and array values, and handle their own memory allocation and deallocation.

The creation of the `a` and `b` `Arrays` requires an object specifying their index domains. Since these are two-dimensional arrays, their index domains are also two-dimensional. The two-dimensional `Interval<2>` object is the Cartesian product of two one-dimensional `Interval<1>` objects, each specifying the integral set $\{0, 1, 2, \dots, n-1\}$.

An `Array`'s template parameters indicate its dimension, the type of its values, and how the values are stored. Both `a` and `b` are two-dimension arrays storing doubles so their dimension is 2 and their value type is `double`. An `Engine` stores an `Array`'s values. For example, a `Brick Engine` explicitly stores all values. A `CompressibleBrick Engine` also explicitly stores values if more than one value is present, but, if all values are the same, storage for just that value is required. Since an engine can

store its values any way it desires, it might instead compute its values using a function or compute using values stored in separate engines. In practice, most explicitly specified Engines are either `Brick` or `CompressibleBrick`.

Arrays support both element-wise access and scalar assignment. Element-wise access uses parentheses, not square brackets. For example, `b(n/2, n/2)` specifies the central element. The scalar assignment `b = 0.0` assigns the same 0.0 value to all array elements. This is possible because the array knows the extent of its domain. We illustrate these data-parallel statements in the next section.

Any program using the POOMA Toolkit must initialize the toolkit's data structures using `Pooma::initialize(argc, argv)`. This extracts POOMA-specific command-line options from the program's command-line arguments and initializes the interprocessor communication and other data structures. When finished, `Pooma::finalize()` ensures all computation and communication has finished and the data structures are destructed.

3.4. Data-Parallel Array Implementation

POOMA supports data-parallel Array accesses. Many algorithms are more easily expressed using data-parallel expressions. Also, the POOMA Toolkit can sometimes reorder the data-parallel computations to be more efficient or distribute them among various processors. In this section, we concentrate on the differences between the data-parallel implementation of `Doof2d` listed in Example 3-3 and the element-wise implementation listed in the previous section.

Example 3-3. Data-Parallel Array Implementation of `Doof2d`

```
#include <iostream>    // has std::cout, ...
#include <stdlib.h>     // has EXIT_SUCCESS
#include "Pooma/Arrays.h"
    // has POOMA's Array declarations

// Doof2d: POOMA Arrays, data-parallel implementation

int main(int argc, char *argv[])
{
    // Prepare the POOMA library for execution.
    Pooma::initialize(argc, argv);
```

```

// Ask the user for the number of averagings.
long nuAveragings, nuIterations;
std::cout << "Please enter the number of averagings: ";
std::cin >> nuAveragings;
nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.

// Ask the user for the number n of values along one
// dimension of the grid.
long n;
std::cout << "Please enter the array size: ";
std::cin >> n;

// Specify the arrays' domains [0,n) x [0,n).
Interval<1> N(0, n-1);
Interval<2> vertDomain(N, N);

// Set up interior domains [1,n-1) x [1,n-1)
// for computation.  (1)
Interval<1> I(1,n-2);
Interval<1> J(1,n-2);

// Create the arrays.
// The Array template parameters indicate 2 dimensions,
// a 'double' value
// type, and ordinary 'Brick' storage.
Array<2, double, Brick> a(vertDomain);
Array<2, double, Brick> b(vertDomain);

// Set up the initial conditions.
// All grid values should be zero except for the
// central value.
a = b = 0.0;
// Ensure all data-parallel computation finishes
// before accessing a value.  (2)
Pooma::blockAndEvaluate();
b(n/2,n/2) = 1000.0;

// In the average, weight elements with this value.

```



```

const double weight = 1.0/9.0;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b.  Write to a.  (3)
    a(I,J) = weight *
        (b(I+1,J+1) + b(I+1,J  ) + b(I+1,J-1) +
         b(I  ,J+1) + b(I  ,J  ) + b(I  ,J-1) +
         b(I-1,J+1) + b(I-1,J  ) + b(I-1,J-1));

    // Read from a.  Write to b.
    b(I,J) = weight *
        (a(I+1,J+1) + a(I+1,J  ) + a(I+1,J-1) +
         a(I  ,J+1) + a(I  ,J  ) + a(I  ,J-1) +
         a(I-1,J+1) + a(I-1,J  ) + a(I-1,J-1));
}

// Print out the final central value.
Pooma::blockAndEvaluate();
    // Ensure all computation has finished.
std::cout <<
    (nuAveragings % 2 ? a(n/2,n/2) : b(n/2,n/2))
    << std::endl;

// The arrays are automatically deallocated.

// Tell the POOMA library execution has finished.
Pooma::finalize();
return EXIT_SUCCESS;
}

```

- (1) These variables specify one-dimensional domains $\{1, 2, \dots, n-2\}$. Their Cartesian product specifies the domain of the array values that are modified.
- (2) POOMA may reorder computation. `Pooma::blockAndEvaluate` ensures all computation finishes before accessing a particular array element.
- (3) Data-parallel expressions replace nested loops and array element accesses. For example, `a(I,J)` represents the subset of the `a` array having a domain equal to the Cartesian product of `I` and `J`. Intervals can shifted by an additive or multiplicative constant.

Data-parallel expressions use containers and domain objects to indicate a set of parallel expressions. For example, in the program listed above, $a(I, J)$ specifies the subset of a array omitting the outermost elements. The array's `vertDomain` domain consists of the Cartesian product of $\{0, 1, 2, \dots, n-1\}$ with itself, while I and J each specify $\{1, 2, \dots, n-2\}$. Thus, $a(I, J)$ is the subset with a domain of the Cartesian product of $\{1, 2, \dots, n-2\}$ with itself. It is called a *view* of an array. It is itself an `Array`, with a domain and supporting element access, but its storage is the same as a 's. Changing a value in $a(I, J)$ also changes the same value in a . Changing a value in the latter also changes the former if the value is not one of a 's outermost elements. The expression $b(I+1, J+1)$ indicates the subset of b with a domain consisting of the Cartesian product of $\{2, 3, \dots, n-1\}$, i.e., the same domain as $a(I, J)$ but shifted up one unit and to the right one unit. Only an `Interval`'s value, not its name, is important so all uses of J in this program could be replaced by I without changing the semantics.

The statement assigning to $a(I, J)$ illustrates that `Arrays` may participate in expressions. Each addend is a view of an array, which is itself an array. The views' indices are zero-based so their sum can be formed by adding identically indexed elements of each array. For example, the lower, left element of the result equals the sum of the lower, left elements of the addend arrays. Figure 3-2 illustrates adding two arrays.

Figure 3-2. Adding Arrays

$b(I, J) + b(I+1, J+1)$					$b(I, J)$					$b(I+1, J+1)$			
2	34	38	42		2	25	27	29		2	19	21	23
1	28	32	36	=	1	17	19	21	+	1	11	13	15
0	12	16	20		0	9	11	13		0	3	5	7
	0	1	2			0	1	2			0	1	2

When adding arrays, values with the same indices, indicated by the small numbers adjacent to the arrays, are added.

POOMA may reorder computation or distribute them among various processors so, before accessing individual values, the code calls `Pooma::blockAndEvaluate`. Before reading an individual `Array` value, calling this function ensures all computations affecting its value have finished, i.e., it has the correct value. Calling this function is necessary only when accessing individual array elements. For example, before the data-parallel operation of printing an array, POOMA will call `blockAndEvaluate`.

ate itself.

3.5. Stencil Array Implementation

Many scientific computations are localized, computing an array's value by using neighboring values. Encapsulating this local computation in a *stencil* can yield faster code because the compiler can determine that all array accesses use the same array. Each stencil consists of a function object and an indication of which neighbors participate in the function's computation.

Example 3-4. Stencil Array Implementation of Doof2d

```
#include <iostream>    // has std::cout, ...
#include <stdlib.h>     // has EXIT_SUCCESS
#include "Pooma/Arrays.h"
    // has POOMA's Array declarations

// Doof2d: POOMA Arrays, stencil implementation

// Define a stencil class performing computation.  (1)
class DoofNinePt
{
public:
    // Initialize the constant average weighting.
    DoofNinePt() : weight(1.0/9.0) {}

    // This stencil operator is applied to each
    // interior domain position (i,j).  The "C"
    // template parameter permits use of this
    // stencil operator with both Arrays & Fields.  (2)
    template <class C>
    inline
    typename C::Element_t
    operator()(const C& c, int i, int j) const {
        return
            weight *
            (c.read(i+1,j+1)+c.read(i+1,j)+c.read(i+1,j-1)+
```

```

        c.read(i ,j+1)+c.read(i ,j)+c.read(i ,j-1)+
        c.read(i-1,j+1)+c.read(i-1,j)+c.read(i-1,j-1));
    }

    inline int lowerExtent(int) const { return 1; }   (3)
    inline int upperExtent(int) const { return 1; }

private:

    // In the average, weight elements with this value.
    const double weight;
};

int main(int argc, char *argv[])
{
    // Prepare the POOMA library for execution.
    Pooma::initialize(argc,argv);

    // Ask the user for the number of averagings.
    long nuAveragings, nuIterations;
    std::cout << "Please enter the number of averagings: ";
    std::cin >> nuAveragings;
    nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.

    // Ask the user for the number n of values along one
    // dimension of the grid.
    long n;
    std::cout << "Please enter the array size: ";
    std::cin >> n;

    // Specify the arrays' domains [0,n) x [0,n).
    Interval<1> N(0, n-1);
    Interval<2> vertDomain(N, N);

    // Set up interior domains [1,n-1) x [1,n-1) for
    // computation.
    Interval<1> I(1,n-2);
    Interval<2> interiorDomain(I,I);

```

```

// Create the arrays.
// The Array template parameters indicate
// 2 dimensions, a 'double' value
// type, and ordinary 'Brick' storage.
Array<2, double, Brick> a(vertDomain);
Array<2, double, Brick> b(vertDomain);

// Set up the initial conditions.
// All grid values should be zero except for the
// central value.
a = b = 0.0;
// Ensure all data-parallel computation finishes
// before accessing a value.
Pooma::blockAndEvaluate();
b(n/2,n/2) = 1000.0;

// Create a stencil performing the computation.  (4)
Stencil<DoofNinePt> stencil;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b.  Write to a.  (5)
    a(interiorDomain) = stencil(b, interiorDomain);

    // Read from a.  Write to b.
    b(interiorDomain) = stencil(a, interiorDomain);
}

// Print out the final central value.
Pooma::blockAndEvaluate();
// Ensure all computation has finished.
std::cout <<
    (nuAveragings % 2 ? a(n/2,n/2) : b(n/2,n/2))
    << std::endl;

// The arrays are automatically deallocated.

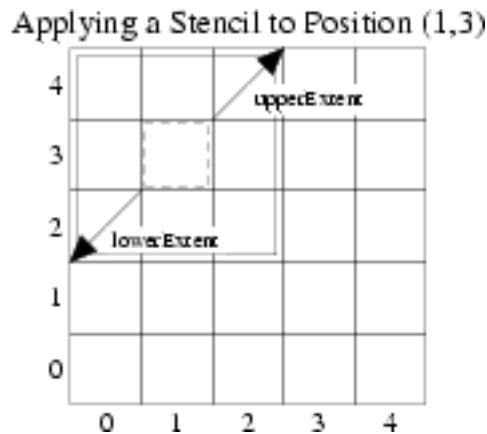
// Tell the POOMA library execution has finished.
Pooma::finalize();
return EXIT_SUCCESS;

```

```
}
```

- (1) A stencil is a function object implementing a local operation on an `Array`.
- (2) POOMA applies this function call operator () to the interior domain of an `Array`. Although not strictly necessary, the function's template parameter `C` permits using this stencil with `Arrays` and other containers. The `read Array` member function supports only reading values, not writing values, thus possibly permitting faster access.
- (3) These two functions indicate the stencil's size. For each dimension, the stencil extends one cell to the left of (or below) its center and also one cell to the right (or above) its center.
- (4) Create the stencil.
- (5) Applying `stencil` to the `b` array and a subset `interiorDomain` of its domain yields an array, which is assigned to a subset of `a`. The stencil's function object is applied to each position in the specified subset of `b`.

Before we describe how to create a stencil, we describe how to apply a stencil to an array, yielding computed values. To compute the value associated with index position (1,3), the stencil's center is placed at (1,3). The stencil's `upperExtent` and `lowerExtent` functions indicate which `Array` values the stencil's function will use. See Figure 3-3. Applying the stencil's function call operator () yields the computed value. To compute multiple `Array` values, apply a stencil to the array and a domain object: `stencil(b, interiorDomain)`. This applies the stencil to each position in the domain. The user must ensure that applying the stencil does not access nonexistent `Array` values.

Figure 3-3. Applying a Stencil to an Array

To compute the value associated with index position (1,3) of an array, place the stencil's center, indicated with dashed lines, at the position (1,3). The computation involves the array values covered by the array and delineated by `upperExtent` and `lowerExtent`.

To create a stencil object, apply the `Stencil` type to a function object class. For example, `Stencil<DoofNinePt> stencil` declares the stencil object. The function object class must define a function call `operator()` with a container parameter and index parameters. The number of index parameters, indicating the stencil's center, must equal the container's dimension. For example, `DoofNinePt` defines `operator()(const C& c, int i, int j)`. We templated the container type `C` although this is not strictly necessary. The two index parameters `i` and `j` ensure the stencil works with two-dimensional containers. The `lowerExtent` function indicates how far to the left (or below) the stencil extends beyond its center. Its parameter indicates a particular dimension. Index parameters `i` and `j` are in dimension 0 and 1. `upperExtent` serves an analogous purpose. The POOMA Toolkit uses these functions when distributing computation among various processors, but it does not use these functions to ensure nonexistent `Array` values are not accessed. Caveat stencil user!

3.6. Distributed Array Implementation

A POOMA program can execute on one or multiple processors. To convert a program designed for uniprocessor execution to a program designed for multiprocessor execution, the programmer need only specify how each container's domain should be split into "patches". The POOMA Toolkit automatically distributes the data among the available

processors and handles any required communication among processors. Example 3-5 illustrates how to write a distributed version of the stencil program (Example 3-4).

Example 3-5. Distributed Stencil Array Implementation of Doof2d

```
#include <iostream>    // has std::cout, ...
#include <stdlib.h>    // has EXIT_SUCCESS
#include "Pooma/Arrays.h"
    // has POOMA's Array declarations

// Doof2d: POOMA Arrays, stencil, multiple
// processor implementation

// Define the stencil class performing the computation.
class DoofNinePt
{
public:
    // Initialize the constant average weighting.
    DoofNinePt() : weight(1.0/9.0) {}

    // This stencil operator is applied to each interior
    // domain position (i,j). The "C" template
    // parameter permits use of this stencil
    // operator with both Arrays and Fields.
    template <class C>
    inline
    typename C::Element_t
    operator()(const C& x, int i, int j) const {
        return
            weight *
            (x.read(i+1,j+1)+x.read(i+1,j)+x.read(i+1,j-1) +
             x.read(i ,j+1)+x.read(i ,j)+x.read(i ,j-1) +
             x.read(i-1,j+1)+x.read(i-1,j)+x.read(i-1,j-1));
    }

    inline int lowerExtent(int) const { return 1; }
    inline int upperExtent(int) const { return 1; }

private:
```



```

    // In the average, weight elements with this value.
    const double weight;
};

int main(int argc, char *argv[])
{
    // Prepare the POOMA library for execution.
    Pooma::initialize(argc,argv);

    // Since multiple copies of this program may simul-
    // taneously run, we cannot use standard input and
    // output. Instead we use command-line arguments,
    // which are replicated, for input, and we use an
    // Inform stream for output.  (1)
    Inform output;

    // Read the program input from the command-line
    // arguments.
    if (argc != 4) {
        // Incorrect number of command-line arguments.
        output <<
            argv[0] <<
            ": number-of-processors number-of-averagings"
            << " number-of-values"
            << std::endl;
        return EXIT_FAILURE;
    }
    char *tail;

    // Determine the number of processors.
    long nuProcessors;
    nuProcessors = strtol(argv[1], &tail, 0);

    // Determine the number of averagings.
    long nuAveragings, nuIterations;
    nuAveragings = strtol(argv[2], &tail, 0);
    nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.

```

```

// Ask the user for the number n of values along
// one dimension of the grid.
long n;
n = strtol(argv[3], &tail, 0);
// The dimension must be a multiple of the number
// of processors since we are using a
// UniformGridLayout.
n=((n+nuProcessors-1)/nuProcessors)*nuProcessors;

// Specify the arrays' domains [0,n) x [0,n).
Interval<1> N(0, n-1);
Interval<2> vertDomain(N, N);

// Set up interior domains [1,n-1) x [1,n-1)
// for computation.
Interval<1> I(1,n-2);
Interval<2> interiorDomain(I,I);

// Create the distributed arrays.

// Partition the arrays' domains uniformly, i.e.,
// each patch has the same size. The first para-
// meter tells how many patches for each dimension.
// Guard layers optimize communication between
// patches. Internal guards surround each patch.
// External guards surround the entire array
// domain.  (2)
UniformGridPartition<2>
    partition(Loc<2>(nuProcessors, nuProcessors),
        GuardLayers<2>(1), // internal
        GuardLayers<2>(0)); // external
UniformGridLayout<2> layout(vertDomain, partition,
    DistributedTag());

// The Array template parameters indicate 2 dims
// and a 'double' value type. MultiPatch indicates
// multiple computation patches, i.e, distributed
// computation. The UniformTag indicates the
// patches should have the same size. Each patch
// has Brick type.  (3)

```

```

Array<2, double, MultiPatch<UniformTag,
    Remote<Brick> > > a(layout);
Array<2, double, MultiPatch<UniformTag,
    Remote<Brick> > > b(layout);

// Set up the initial conditions.
// All grid values should be zero except for the
// central value.
a = b = 0.0;
// Ensure all data-parallel computation finishes
// before accessing a value.
Pooma::blockAndEvaluate();
b(n/2,n/2) = 1000.0;

// Create the stencil performing the computation.
Stencil<DoofNinePt> stencil;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b.  Write to a.  (4)
    a(interiorDomain) = stencil(b, interiorDomain);

    // Read from a.  Write to b.
    b(interiorDomain) = stencil(a, interiorDomain);
}

// Print out the final central value.
Pooma::blockAndEvaluate();
// Ensure all computation has finished.
output <<
    (nuAveragings % 2 ? a(n/2,n/2) : b(n/2,n/2))
    << std::endl;

// The arrays are automatically deallocated.

// Tell the POOMA library execution has finished.
Pooma::finalize();
return EXIT_SUCCESS;
}

```

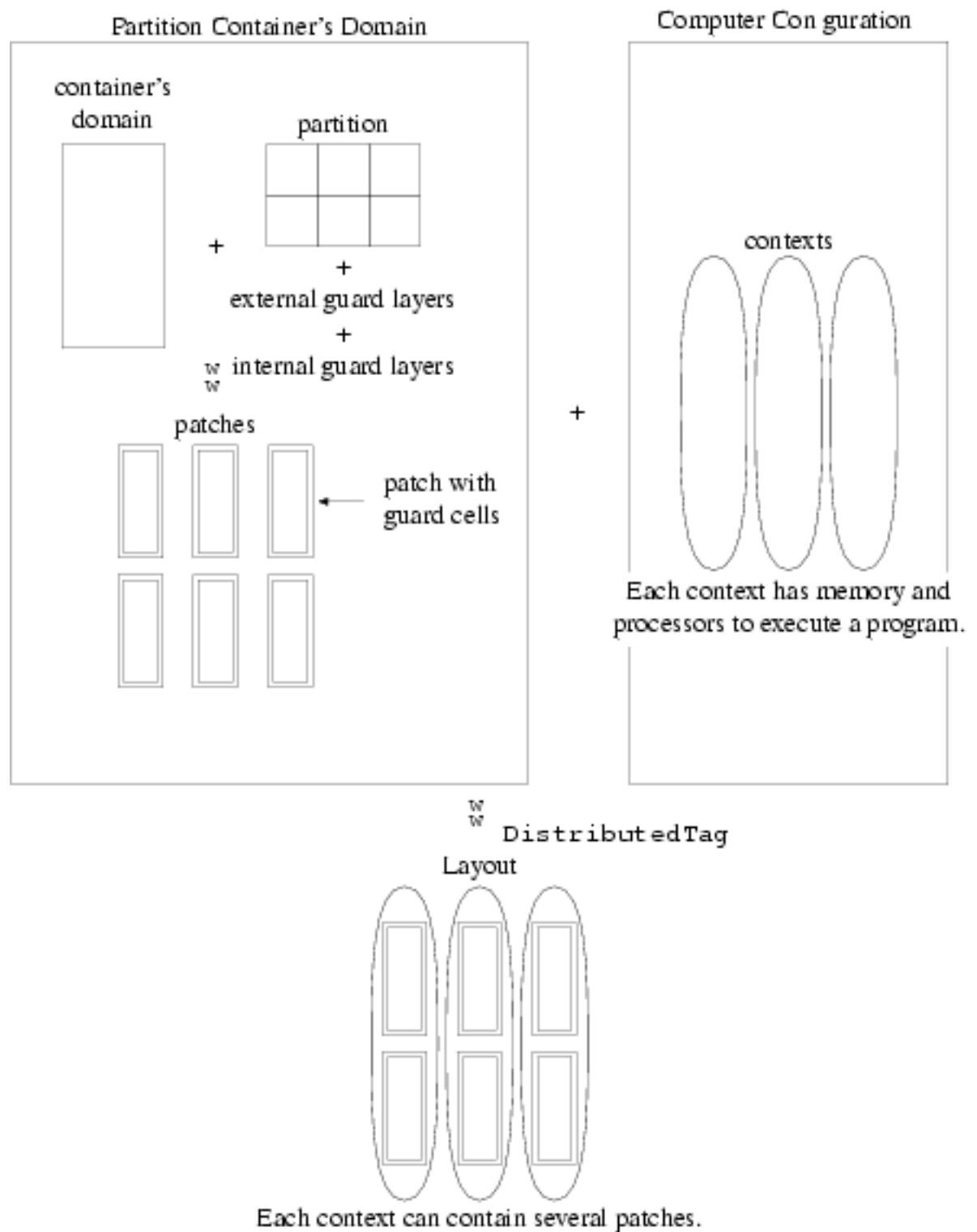
- (1) Multiple copies of a distributed program may simultaneously run, perhaps each having its own input and output. Thus, we use command-line arguments to pass input to the program. Using an `Inform` object ensures only one copy produces output.
- (2) The `UniformGridPartition` declaration specifies how an array's domain will be partitioned, or split, into patches. Guard layers are an optimization that can reduce data communication between patches. The `UniformGridLayout` declaration applies the partition to the given domain, distributing the resulting patches among various processors.
- (3) The `MultiPatch Engine` distributes requests for `Array` values to the associated patches. Since a patch may be associated with a different processor, its "remote" Engine has type `Remote<Brick>`. POOMA automatically distributes the patches among available memories and processors.
- (4) The stencil computation, whether for one processor or multiple processors, is the same.

Supporting distributed computation requires only minor code changes. These changes specify how each container's domain is distributed among the available processors and how input and output occurs. The rest of the program, including all the computations, remains the same. When running, the POOMA executable interacts with the run-time library to determine which processors are available, distributes the containers' domains, and automatically handles all necessary interprocessor communication. The same executable runs on one or many processors. Thus, the programmer can write one program, debugging it on a uniprocessor computer and run it on a supercomputer.

POOMA's distributed computing model separates container domain concepts from computer configuration concepts. See Figure 3-4. The statements in the program indicate how each container's domain will be partitioned. This process is represented in the upper left corner of the figure. A user-specified *partition* specifies how to split the domain into pieces. For example, the illustrated partition splits the domain into three equal-sized pieces along the x-dimension and two equal-sized pieces along the y-dimension. Applying the partition to the domain creates *patches*. The partition also specifies external and internal guard layers. A *guard layer* is a domain surrounding a patch. A patch's computation only reads but does not write these guarded values. An *external guard layer* conceptually surrounds the entire container domain with boundary values whose presence permits all domain computations to be performed the same way even for computed values along the domain's edge. An *internal guard layer* duplicates values from adjacent patches so communication need not occur during a patch's computation. The use of guard layers is an optimization; using external guard layers eases programming and

using internal guard layers reduces communication among processors. Their use is not required.

Figure 3-4. The POOMA Distributed Computation Model



The POOMA distributed computation model creates a layout by combining a partition-

ing of the containers' domains and the computer configuration.

The computer configuration of shared memory and processors is determined by the run-time system. See the upper right portion of Figure 3-4. A *context* is a collection of shared memory and processors that can execute a program or a portion of a program. For example, a two-processor desktop computer might have memory accessible to both processors so it is a context. A supercomputer consisting of desktop computers networked together might have as many contexts as computers. The run-time system, e.g., the Message Passing Interface (MPI) Communications Library or the MM Shared Memory Library (<http://www.engelschall.com/sw/mm/>), communicates the available contexts to the executable. POOMA must be configured for the particular run-time system in use. See Section A.1.

A *layout* combines patches with contexts so the program can be executed. If `DistributedTag` is specified, the patches are distributed among the available contexts. If `ReplicatedTag` is specified, each set of patches is replicated on each context. Regardless, the containers' domains are now distributed among the contexts so the program can run. When a patch needs data from another patch, the POOMA Toolkit sends messages to the desired patch uses the message-passing library. All such communication is automatically performed by the toolkit with no need for programmer or user input.

Incorporating POOMA's distributed computation model into a program requires writing very few lines of code. Example 3-5 illustrates this. The `partition` declaration creates a `UniformGridPartition` splitting each dimension of a container's domain into equally-sized `nuProcessors` pieces. The first `GuardLayers` argument specifies each patch will have copy of adjacent patches' outermost values. This may speed computation because a patch need not synchronize its computation with other patches' processors. Since each value's computation requires knowing its surrounding neighbors, this internal guard layer is one layer deep. The second `GuardLayers` argument specifies no external guard layer. External guard layers simplify computing values along the edges of domains. Since our program already uses only the interior domain for computation, we do not use this feature.

The `layout` declaration creates a `UniformGridLayout` layout. As Example 3-5 illustrates, it needs to know a container's domain, a partition, the computer's contexts, and a `DistributedTag` or `ReplicatedTag`. These comprise `layout`'s three parameters; the contexts are implicitly supplied by the run-time system.

To create a distributed `Array`, it should be created using a `Layout` object and have a `MultiPatch Engine` rather than using a `Domain` object and a `Brick Engine` as we did for the uniprocessor implementations. A distributed implementation uses a `Layout` object, which conceptually specifies a `Domain` object and its distribution

throughout the computer. A `MultiPatch Engine` supports computations using multiple patches. The `UniformTag` indicates the patches all have the same size. Since patches may reside on different contexts, the second template parameter is `Remote`. Its `Brick` template parameter specifies the `Engine` for a particular patch on a particular context. Most distributed programs use

```
MultiPatch<UniformTag, Remote<Brick>>
```

or

```
MultiPatch<UniformTag, Remote<CompressibleBrick>>
```

or `Engines`.

The computations for a distributed implementation are exactly the same as for a sequential implementation. The POOMA Toolkit and a message-passing library automatically perform all the computation.

Input and output for distributed programs is different than for sequential programs. Although the same instructions run on each context, each context may have its own input and output streams. To avoid dealing with multiple input streams, we pass the input via command-line arguments, which are replicated for each context. Using `Inform` streams avoids having multiple output streams print. Any context can print to an `Inform` stream but only text sent to context 0 is displayed. At the beginning of the program, we create an `Inform` object named `output`. Throughout the rest of the program, we use it instead of `std::cout` and `std::cerr`.

The command to run the program is dependent on the run-time system. To use MPI with the Irix 6.5 operating system, one can use the `mpirun` command. For example, `mpirun -np 4 Doof2d-Array-distributed -mpi 2 10 1000` invokes the MPI run-time system with four processors. The `-mpi` option tells the POOMA executable `Doof2d-Array-distributed` to use the MPI Library. The remaining arguments specify the number of processors, the number of averagings, and the array size. The first and last values are the same for each dimension. For example, if three processors are specified, then the x-dimension will have three processors and the y-dimension will have three processors, totaling nine processors. The command `Doof2d-Array-distributed -shmem -np 4 2 10 1000` uses the MM Shared Memory Library (`-shmem`) and four processors. As for MPI, the remaining command-line arguments are specified on a per-dimension basis for the two-dimensional program.

3.7. Data-Parallel Field Implementation

POOMA Arrays support many scientific computations, but other scientific computations require values distributed throughout space, and Arrays have no spatial extent. POOMA Fields, supporting a superset of Array functionality, model values distributed throughout space.

A Field consists of a set of cells distributed through space. Like an Array cell, each Field cell is addressed via indices. Unlike an Array cell, each Field cell can hold multiple values. Like Arrays, Fields can be accessed via data-parallel expressions and stencils and may be distributed across processors. Unlike Array cells, Field cells exist in a multidimensional volume so, e.g., distances between cells and normals to cells can be computed.

In this section, we implement the Doof2d two-dimensional diffusion simulation program using Fields. This simulation does not require any Field-specific features, but we present this program rather than one using Field-specific features to facilitate comparison with the Array versions, especially Example 3-3.

Example 3-6. Data-Parallel Field Implementation of Doof2d

```
#include <iostream>    // has std::cout, ...
#include <stdlib.h>    // has EXIT_SUCCESS
#include "Pooma/Fields.h"
    // has POOMA's Field declarations    (1)

// Doof2d: POOMA Fields, data-parallel implementation

int main(int argc, char *argv[])
{
    // Prepare the POOMA library for execution.
    Pooma::initialize(argc,argv);

    // Ask the user for the number of averagings.
    long nuAveragings, nuIterations;
    std::cout<<"Please enter the number of averagings: ";
    std::cin >> nuAveragings;
    nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.
```

```

// Ask the user for the number n of values along
// one dimension of the grid.
long n;
std::cout << "Please enter the field size: ";
std::cin >> n;

// Specify the fields' domains [0,n) x [0,n).
Interval<1> N(0, n-1);
Interval<2> vertDomain(N, N);

// Set up interior domains [1,n-1) x [1,n-1) for
// computation.
Interval<1> I(1,n-2);
Interval<1> J(1,n-2);

// Specify the fields' mesh, i.e., its spatial
// extent, and its centering type. (2)
DomainLayout<2> layout(vertDomain);
UniformRectilinearMesh<2>
    mesh(layout, Vector<2>(0.0), Vector<2>(1.0, 1.0));
Centering<2> cell =
    canonicalCentering<2>(CellType, Continuous, AllDim);

// Create the fields.
// The Field template parameters indicate a mesh, a
// 'double' value type, and ordinary 'Brick'
// storage. (3)
Field<UniformRectilinearMesh<2>, double, Brick>
    a(cell, layout, mesh);
Field<UniformRectilinearMesh<2>, double, Brick>
    b(cell, layout, mesh);

// Set up the initial conditions.
// All grid values should be zero except for the
// central value.
a = b = 0.0;
// Ensure all data-parallel computation finishes
// before accessing a value.
Pooma::blockAndEvaluate();
b(n/2,n/2) = 1000.0;

```

```

// In the average, weight elements with this value.
const double weight = 1.0/9.0;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b. Write to a. (4)
    a(I,J) = weight *
        (b(I+1,J+1) + b(I+1,J ) + b(I+1,J-1) +
         b(I ,J+1) + b(I ,J ) + b(I ,J-1) +
         b(I-1,J+1) + b(I-1,J ) + b(I-1,J-1));

    // Read from a. Write to b.
    b(I,J) = weight *
        (a(I+1,J+1) + a(I+1,J ) + a(I+1,J-1) +
         a(I ,J+1) + a(I ,J ) + a(I ,J-1) +
         a(I-1,J+1) + a(I-1,J ) + a(I-1,J-1));
}

// Print out the final central value.
Pooma::blockAndEvaluate();
    // Ensure all computation has finished.
std::cout <<
    (nuAveragings % 2 ? a(n/2,n/2) : b(n/2,n/2))
    << std::endl;

// The fields are automatically deallocated.

// Tell the POOMA library execution has finished.
Pooma::finalize();
return EXIT_SUCCESS;
}

```

- (1) To use `Fields`, the `Pooma/Fields.h` must be included.
- (2) These statements specify the spacing and number of `Field` values. First, a layout is specified. Then, a mesh, which specifies the spacing between cells, is created. The `Field`'s centering specifies one cell-centered value per cell.
- (3) `Field`'s first template parameter specifies the type of mesh to use. The other template parameters are similar to `Array`'s. The constructor arguments specify the

`Field`'s centering, its domain of cells, and a mesh specifying the cells' spatial arrangement.

- (4) The computation for `Fields` is the same as for `Arrays` because this example does not use any `Field`-specific features.

As mentioned above, the fundamental difference between `Arrays` and `Fields` is the latter has cells and meshes. The `Field` declarations reflect this. To declare a `Field`, the `Pooma/Fields.h` header file must be included. A `Field`'s domain consists of a set of cells, sometimes called positions when referring to `Arrays`. As for `Arrays`, a `Field`'s domain and its layout must be specified. Since the above program is designed for uniprocessor computation, specifying the domain specifies the layout. A `Field`'s *mesh* specifies its spatial extent. For example, one can ask the mesh for the distance between two cells or for the normals to a particular cell. Cells in a `UniformRectilinearMesh` all have the same size and are parallelepipeds. To create the mesh, one specifies the layout, the location of the spatial point corresponding to the lower, left domain location, and the size of a particular cell. Since this program does not use mesh computations, our choices do not matter. We specify the domain's lower, left corner as spatial location (0.0, 0.0) and each cell's width and height as 1. Thus, the middle of the cell at domain position (3,4) is (3.5, 4.5).

A `Field` cell can contain one or more values although each cell must have the same arrangement of values. For this simulation, we desire one value per cell so we place that position at the cell's center, i.e., a cell centering. The `canonicalCentering` function returns such a centering. .

A `Field` declaration is analogous to an `Array` declaration but must also specify a centering and a mesh. In Example 3-3, the `Array` declaration specifies the array's dimensionality, the value type, the `Engine` type, and a layout. `Field` declarations specify the same values. Its first template parameter specifies the mesh's type, which includes an indication of its dimensionality. The second and third template parameters specify the value type and the `Engine` type. Since a `Field` has a centering and a mesh in addition to a layout, those arguments are also necessary.

`Field` operations are a superset of `Array` operations so the `Doof2d` computations are the same as in Example 3-3. `Field` accesses require parentheses, not square brackets, and accesses to individual values should be preceded by calls to `Pooma::blockAndEvaluate`.

To summarize, `Fields` support multiple values per cell and have spatial extent. Thus, their declarations must specify a centering and a mesh. Otherwise, a `Field` program is similar to one using `Arrays`.

3.8. Distributed Field Implementation

A POOMA program using `Fields` can execute on one or more processors. In Section 3.6, we demonstrated how to modify a uniprocessor stencil `Array` implementation to run on multiple processors. In this section, we demonstrate that the uniprocessor data-parallel `Field` implementation of the previous section can be similarly converted. Only the container declarations change; the computations do not. Since the changes are exactly analogous to those in Section 3.6, our exposition here will be shorter.

Example 3-7. Distributed Data-Parallel Field Implementation of Doof2d

```
#include <stdlib.h> // has EXIT_SUCCESS
#include "Pooma/Fields.h"
    // has POOMA's Field declarations

// Doof2d: POOMA Fields, data-parallel, multiple
// processor implementation

int main(int argc, char *argv[])
{
    // Prepare the POOMA library for execution.
    Pooma::initialize(argc,argv);

    // Since multiple copies of this program may
    // simultaneously run, we cannot use standard input
    // and output. Instead we use command-line
    // arguments, which are replicated, for input, and we
    // use an Inform stream for output.  (1)
    Inform output;

    // Read the program input from the command-line arguments.
    if (argc != 4) {
        // Incorrect number of command-line arguments.
        output << argv[0] <<
            ": number-of-processors number-of-averagings"
            << " number-of-values"
            << std::endl;
        return EXIT_FAILURE;
    }
}
```

```

char *tail;

// Determine the number of processors.
long nuProcessors;
nuProcessors = strtol(argv[1], &tail, 0);

// Determine the number of averagings.
long nuAveragings, nuIterations;
nuAveragings = strtol(argv[2], &tail, 0);
nuIterations = (nuAveragings+1)/2;
    // Each iteration performs two averagings.

// Ask the user for the number n of values along
// one dimension of the grid.
long n;
n = strtol(argv[3], &tail, 0);
// The dimension must be a multiple of the number of
// processors since we are using a UniformGridLayout.
n = ((n+nuProcessors-1) / nuProcessors) * nuProcessors;

// Specify the fields' domains [0,n) x [0,n).
Interval<1> N(0, n-1);
Interval<2> vertDomain(N, N);

// Set up interior domains [1,n-1) x [1,n-1) for
// computation.
Interval<1> I(1,n-2);
Interval<1> J(1,n-2);

// Partition the fields' domains uniformly, i.e.,
// each patch has the same size. The first parameter
// tells how many patches for each dimension. Guard
// layers optimize communication between patches.
// Internal guards surround each patch. External
// guards surround the entire field domain. (2)
UniformGridPartition<2>
    partition(Loc<2>(nuProcessors, nuProcessors),
        GuardLayers<2>(1), // internal
        GuardLayers<2>(0)); // external
UniformGridLayout<2>

```

```

layout(vertDomain, partition, DistributedTag());

// Specify the fields' mesh, i.e., its spatial
// extent, and its centering type.  (3)
UniformRectilinearMesh<2>
    mesh(layout, Vector<2>(0.0), Vector<2>(1.0, 1.0));
Centering<2> cell =
    canonicalCentering<2>(CellType, Continuous, AllDim);

// The Field template parameters indicate a mesh and
// a 'double' value type. MultiPatch indicates
// multiple computation patches, i.e., distributed
// computation. The UniformTag indicates the patches
// should have the same size. Each patch has Brick
// type.  (4)
Field<UniformRectilinearMesh<2>, double,
    MultiPatch<UniformTag, Remote<Brick>>>
    a(cell, layout, mesh);
Field<UniformRectilinearMesh<2>, double,
    MultiPatch<UniformTag, Remote<Brick>>>
    b(cell, layout, mesh);

// Set up the initial conditions.
// All grid values should be zero except for the
// central value.
a = b = 0.0;
// Ensure all data-parallel computation finishes
// before accessing a value.
Pooma::blockAndEvaluate();
b(n/2,n/2) = 1000.0;

// In the average, weight elements with this value.
const double weight = 1.0/9.0;

// Perform the simulation.
for (int k = 0; k < nuIterations; ++k) {
    // Read from b. Write to a.
    a(I,J) = weight *
        (b(I+1,J+1) + b(I+1,J ) + b(I+1,J-1) +
         b(I ,J+1) + b(I ,J ) + b(I ,J-1) +

```

```

        b(I-1,J+1) + b(I-1,J  ) + b(I-1,J-1));

    // Read from a.  Write to b.
    b(I,J) = weight *
        (a(I+1,J+1) + a(I+1,J  ) + a(I+1,J-1) +
         a(I  ,J+1) + a(I  ,J  ) + a(I  ,J-1) +
         a(I-1,J+1) + a(I-1,J  ) + a(I-1,J-1));
}

// Print out the final central value.
Pooma::blockAndEvaluate();
    // Ensure all computation has finished.
output <<
    (nuAveragings % 2 ? a(n/2,n/2) : b(n/2,n/2))
    << std::endl;

// The fields are automatically deallocated.

// Tell the POOMA library execution has finished.
Pooma::finalize();
return EXIT_SUCCESS;
}

```

- (1) Multiple copies of a distributed program may simultaneously run, perhaps each having its own input and output. Thus, we use command-line arguments to pass input to the program. Using an Inform stream ensures only one copy produces output.
- (2) The `UniformGridPartition` declaration specifies how an array's domain will be partitioned, or split, into patches. Guard layers are an optimization that can reduce data communication between patches. The `UniformGridLayout` declaration applies the partition to the given domain, distributing the resulting patches among various processors.
- (3) The mesh and centering declarations are the same for uniprocessor and multiprocessor implementations.
- (4) The `MultiPatch Engine` distributes requests for `Field` values to the associated patch. Since a patch may be associated with a different processor, its "remote" engine has type `Remote<Brick>`. POOMA automatically distributes the patches among available memories and processors.

This program can be viewed as the combination of Example 3-6 and the changes to form the distributed stencil-based Array program from the uniprocessor stencil-based Array program.

- Distributed programs may have multiple processes, each with its own input and output streams. To pass input to these processes, this programs uses command-line arguments, which are replicated for each process. An `Inform` stream accepts data from any context but prints only data from context 0.
- A layout for a distributed program specifies a domain, a partition, and a context mapper. A `DistributedTag` context mapper tag indicates that pieces of the domain should be distributed among patches, while a `ReplicatedTag` context mapper tag indicates the entire domain should be replicated to each patch.
- A `MultiPatch Engine` supports the use of multiple patches, while a `remote` engine supports computation distributed among various contexts. Both are usually necessary for distributed computation.
- The computation for uniprocessor or distributed implementations remains the same. The POOMA Toolkit automatically handles all communication necessary to ensure up-to-date values are available when needed.
- The command to invoke a distributed program is system-dependent. For example, the `mpirun -np 4 Doof2d-Field-distributed -mpi 2 10 1000` command might use MPI communication.

```
Doof2d-Field-distributed -shmem -np 4 2 10 1000
```

might use the MM Shared Memory Library.

Chapter 4. Overview of POOMA Concepts

In the previous chapter, we presented several different implementations of the `Doof2d` simulation program. The implementations illustrate the various containers, computation modes, and computation environments that POOMA supports. In this chapter, we describe the concepts associated with each of these three categories. Specific details needed for their use are deferred to later chapters.

The most important POOMA concepts can be grouped into three separate categories:

containers

data structures holding one or more values and usually accessed using indices

computation modes

styles of expressing computations and accesses to container values

computation environment

description of resources for computing, e.g., single processor or multiprocessor.

Table 4-1 categorizes the POOMA concepts. Many POOMA programs select one possibility from each category. For example, Example 3-4 used `Array` containers and stencils for sequential computation, while Example 3-7 used `Field` containers and data-parallel statements with distributed computation. A program may use multiple containers and various computation modes, but the computation environment is either distributed or not.

Table 4-1. POOMA Concepts

Container	Computation Modes	Computation Environment
<code>Array</code>	element-wise	sequential
<code>DynamicArray</code>	data-parallel	distributed
<code>Field</code>	stencil-based	
<code>Tensor</code>	relational	
<code>TinyMatrix</code>		
<code>Vector</code>		

In the rest of this chapter, we explore these three categories. First, we describe POOMA containers, illustrating the purposes of each, and explaining the concepts needed to declare them. Then, we describe the different computation modes and distributed computation concepts.

4.1. POOMA Containers

Most POOMA programs use *containers* to store groups of values. POOMA containers are objects that store other objects such as numbers or vectors. They control allocation and deallocation of these stored objects and access to them. They are a generalization of C arrays, but POOMA containers are first-class objects so they can be used directly in expressions. They are also similar to C++ containers such as `vector`, `list`, and `stack`. See Table 4-2 for a summary of the containers.

This section describes many concepts, but one need not understand them all to begin programming with the POOMA Toolkit. First, we introduce the different POOMA's containers and describe how to choose an appropriate one for a particular task. Figure 4-1 indicates which concepts must be understood when declaring a particular container. All of these concepts are described in Section 4.1.2 and Section 4.1.3. Use this figure to decide which concepts in the former are relevant. Reading the latter section is necessary only if computing using multiple processors. The programs in the previous chapter illustrate many of these concepts.

Table 4-2 briefly describes the six POOMA containers. They are more fully described in the paragraphs below.

Table 4-2. POOMA Container Summary

<i>Array</i>	container mapping <i>indices</i> to values and that may be used in expressions
<i>DynamicArray</i>	one-dimensional <i>Array</i> whose <i>domain</i> can be dynamically resized
<i>Field</i>	container mapping <i>indices</i> to one or more values and residing in multidimensional space
<i>Tensor</i>	multidimensional mathematical tensor
<i>TinyMatrix</i>	two-dimensional mathematical matrix
<i>Vector</i>	multidimensional mathematical vector

A POOMA *Array* generalizes a C array and maps *indices* to values. Given an index

or position in an `Array`'s domain, it returns the associated value, either by returning a stored value or by computing it. The use of indices, which are usually ordered tuples, permits constant-time access although computing a particular value may require significant time. In addition to the functionality provided by C arrays, the `Array` class automatically handles memory allocation and deallocation, supports a wider variety of assignments, and can be used in expressions. For example, the addition of two arrays can be assigned to an array and the product of a scalar element and an array is permissible.

A POOMA *DynamicArray* extends `Array` capabilities to support a dynamically-changing domain but is restricted to only one dimension. When the `DynamicArray` is resized, its values are preserved.

A POOMA *Field* is an `Array` with spatial extent. Each domain consists of *cells* in one-, two-, or three-dimensional space. Although indexed similarly to `Arrays`, each cell may contain multiple values and multiple materials. A `Field`'s *mesh* stores its spatial characteristics and can yield, e.g., the cell at a particular point, the distance between two cells, or a cell's normals. A `Field` should be used whenever geometric or spatial computations are needed, multiple values per index are desired, or a computation involves more than one material.

A *Tensor* implements a multidimensional mathematical tensor. Since it is a first-class type, it can be used in expressions such as adding two `Tensors`.

A *TinyMatrix* implements a two-dimensional mathematical matrix. Since it is a first-class type, it can be used in expressions such as assignments to matrices and multiplying matrices.

A *Vector* implements a multidimensional mathematical vector, which is an ordered tuple of components. Since it is a first-class type, it can be used in expressions such as adding two `Vectors` and multiplying a `TinyMatrix` and a `Vector`.

The data of an `Array`, `DynamicArray`, or `Field` can be accessed using more than one container by taking a view. A *view* of an existing container `C` is a container whose domain is a subset of `C`'s domain. The subset can equal the original domain. A view acts like a reference in that changing any of the view's values also changes the original container's and vice versa. While users sometimes explicitly create views, they are perhaps more frequently created as temporaries in expressions. For example, if `A` is an `Array` and `I` is a domain, `A(I) - A(I-1)` uses two views to form the difference between adjacent values.

4.1.1. Choosing a Container

The two most commonly used POOMA containers are `Arrays` and `Fields`, while

`Vector`, `TinyMatrix`, and `Tensor` represent mathematical objects. Table 4-3 contains a decision tree describing how to choose an appropriate container.

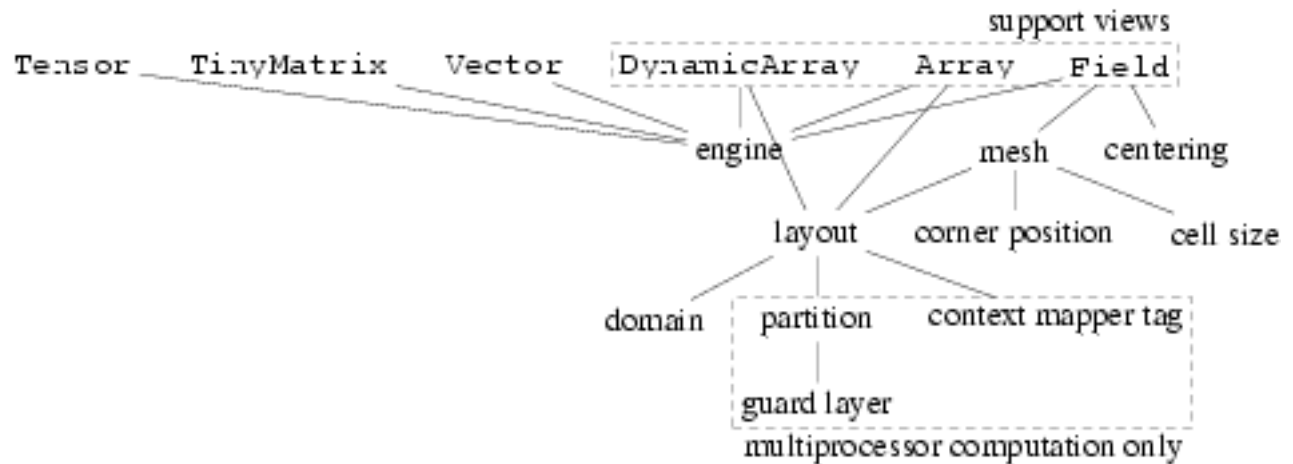
Table 4-3. Choosing a POOMA Container

If modeling mathematical entries,	use a <code>Vector</code> , <code>TinyMatrix</code> , or <code>Tensor</code> .
If indices and values reside in multidimensional space \mathfrak{R}_d ,	use a <code>Field</code> .
If there are multiple values per index,	use a <code>Field</code> .
If there are multiple materials participating in the same computation,	use a <code>Field</code> .
If the domain's size dynamically changes and is one-dimensional,	use a <code>DynamicArray</code> .
Otherwise	use an <code>Array</code> .

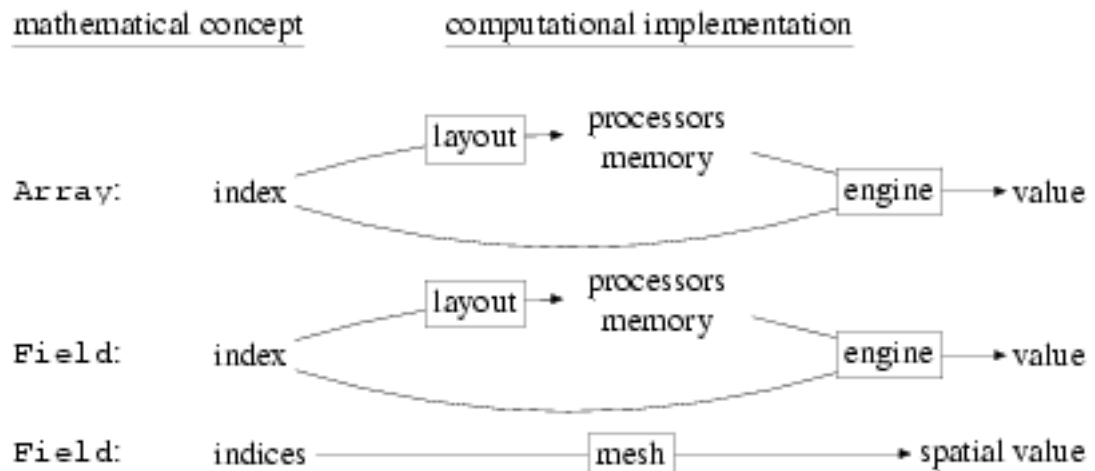
4.1.2. Declaring Sequential Containers

In the previous sections, we introduced the POOMA containers and described how to choose one appropriate for a given task. In this section, we describe the concepts involved in declaring them. Concepts specific to distributed computation are described in the next section.

Figure 4-1 illustrates the containers and the concepts involved in their declarations. The containers are listed in the top row. Lines connect these containers to the components necessary for their declarations. For example, an `Array` declaration requires an engine and a layout. These, in turn, can depend on other POOMA concepts. Declarations necessary only for distributed, or multiprocessor, computation are also indicated. Given a desired container, one can use this figure to determine the concepts needed to declare a particular container.

Figure 4-1. Concepts For Declaring Containers

An *engine* stores and, if necessary, computes a container's values. A container has one or more engines. The separation of a container from its storage permits optimizing a program's space and time requirements. For example, a container returning the same value for all indices can use a constant engine, which need only store one value for the entire domain. A CompressibleBrick Engine reduces its space requirements to a constant whenever all its values are the same. The separation between a container and its engine also permits taking views of containers without copying storage.

Figure 4-2. Array and Field Mathematical and Computational Concepts

A *layout* maps domain *indices* to the processors and computer memory used by a container's engines. See Figure 4-2. A program computes a container's values using these processors and memory. The layout specifies the processors and memory to use for each particular index. A container's layout for a uniprocessor implementation consists of its

domain, the processor, and its memory. For a multiprocessor implementation, the layout maps portions of the domain to (possibly different) processors and memory.

A *domain* is a set of points on which a container can define values. There are several different types of domains. An *interval* consists of all integral points between two endpoints. It is frequently represented using mathematical interval notation $[a,b]$; it contains only the integral points, e.g., $a, a+1, a+2, \dots, b$. The concept is generalized to multiple dimensions by forming direct products of intervals, i.e., all the integral tuples in an n -dimensional space. For example, the two-dimensional containers in the previous chapter are defined on a two-dimensional domain with the both dimensions' spanning the interval $[0,n)$. A domain need not contain all integral points between its endpoints. A *stride* indicates a regular spacing between points. A *range* is a subset of an interval of regularly-spaced points specified by a stride.

A `Field`'s *mesh* maps domain indices to spatial values in \mathcal{R}^d such as distances between cells, edge lengths, and normals to cells. In other words, it provides a `Field`'s spatial extent. See also Figure 4-2. Different mesh types may support different spatial values.

A mesh's *corner position* specifies the point in \mathcal{R}^d corresponding to the cell in the lower, left corner of its domain. Combining this, the domain, and the cell size can specify the mesh's map from indices to \mathcal{R}^d .

A mesh's *cell size* specifies the spatial dimensions of a `Field` cell, e.g., its width, height, and depth, in \mathcal{R}^d . Combining this, the domain, and the corner position can specify the mesh's map from indices to \mathcal{R}^d .

4.1.3. Declaring Distributed Containers

In the previous section, we introduced the important concepts for declaring containers for use on uniprocessor computers. When using multiprocessor computers, we augment these concepts with those for distributed computation. Reading this section is important only for running a program on multiple processors. Many of these concepts were introduced in Section 3.6 and Section 3.8. Figure 3-4 illustrates the POOMA distributed computation model. In this section, we concentrate on the concepts necessary to declare a distributed container.

As we noted in Section 3.6, a POOMA programmer must specify how each container's domain should be distributed among the available processors and memory spaces. Using this information, the toolkit automatically distributes the data among the available processors and handles any required communication among them. The three concepts necessary for declaring distributed containers are a partition, a guard layer, and a context mapper tag.

A *partition* specifies how to divide a container's domain into distributed pieces. For example, the partition illustrated in Figure 3-4 would divide a two-dimensional domain into three equally-sized pieces along the x-dimension and two equally-sized pieces along the y-dimension. Partitions can be independent of the size of container's domain. The example partition will work on any domain as long as the size of its x-dimension is a multiple of three. A domain is separated into disjoint patches.

A *guard layer* surrounds each patch with read-only values. An *external guard layer* specifies values surrounding the entire domain. Its presence eases computation along the domain's edges by permitting the same computations as for more-internal computations. An *internal guard layer* duplicates values from adjacent patches so communication with adjacent patches need not occur during a patch's computation. The use of guard layers is an optimization; using external guard layers eases programming and using internal guard layers reduces communication among processors. Their use is not required.

A *context mapper* indicates how a container's patches are mapped to processors and shared memory. For example, the `DistributedTag` indicates that the patches should be distributed among the processors so each patch occurs once in the entire computation. The `ReplicatedTag` indicates that the patches should be replicated among the processors so each processing unit has its own copy of all the patches. While it could be wasteful to have different processors perform the same computation, replicating a container can reduce possibly more expensive communication costs.

4.2. Computation Modes

POOMA computations can be expressed using a variety of modes. Many POOMA computations involve `Array` or `Field` containers, but how their values are accessed and how the associated algorithms use them varies. For example, element-wise computation involves explicitly accessing a container's values. A data-parallel computation operates on larger subsets of a container's values. Stencil-based computations express a computation as repeatedly applying a local computation to each element of an array. Relation-based computations use relations on containers to establish dependencies among them so the values of one container are updated whenever any other's values change. A program may use any or all of these styles, which are described below.

Element-wise computation accesses individual container values through explicit notation. For example, values in a two-dimensional container `C` might be referenced as `C(3 , 4)` or `C(i , j+1)`. This is the usual notation for non-object-oriented languages such as C.

Data-parallel computation uses expressions to access subsets of a container's values. For example, in Example 3-3, $a(I, J)$ represents the subset of Array a 's values having coordinates in the domain specified by the direct product of one-dimensional Intervals I and J . Using data-parallel expressions frequently eliminates the need for writing explicit loops.

Stencil-based computation uses *stencils* to compute containers' values using neighboring data values. Each stencil consists of a specification of which neighboring values to read and a function using those values. For example, an averaging stencil may access all its adjacent neighbors, averaging them. In POOMA, we represent a stencil using a function object with additional functions indicating which neighboring values are used. Stencil computations are frequently used in solving partial differential equations, image processing, and geometric modeling.

Relation-based computation uses *relations* to create dependences among containers such that the dependent container's values are updated when its values are needed and any of its related containers' values have changed. A relation is specified by a dependent container, independent containers, and a function computing the dependent container's values using the independent containers' values. To avoid excess computation, the dependent container's values are computed only when needed, e.g., for printing the container or for computing the values of another dependent container. Thus, this computation is sometimes called "lazy evaluation".

4.3. Computation Environment

The same POOMA program can execute on a wide variety of computers. The default *sequential computing environment* consists of one processor and its associated memory, as found on a personal computer. In contrast, a *distributed computing environment* may have multiple processors and multiple distributed or shared memories. For example, some desktop computers have dual processors and shared memory, while a large supercomputer may have thousands of processors, perhaps with groups of eight sharing the same memory.

Using distributed computation requires three things:

1. The program must declare how container domains will be distributed.
 2. POOMA must be configured to use a communications library.
 3. The POOMA executable must be run using the communications library.
- All of these were illustrated in Section 3.6 and Section 3.8. Figure 3-4 illustrates the

POOMA distributed computation model. Section 4.1.3 described how to declare containers with distributed domains. Here we present three concepts for distributed computation: patches, context, and a communication library.

A partition divides a container's domain into disjoint *patches*. For distributed computation, the patches are distributed among various processors, which compute the associated values. As illustrated in Figure 3-4, each patch can be surrounded by guard layers.

A *context* is a collection of shared memory and processors that can execute a program or a portion of a program. It can have one or more processors, but all these processors must access the same shared memory. Usually the computer and its operating system, not the programmer, determine the available contexts.

A *communication library* passes messages among contexts. POOMA uses the communication library to copy information among contexts, all of which is hidden from both the programmer and the user. POOMA works with the Message Passing Interface (MPI) Communications Library and the MM Shared Memory Library. See Section A.1 for details.

Chapter 5. Array Containers

A container is an object holding objects. Arrays are one of the two most widely used POOMA containers since they model the mathematical concept of mapping from domain indices to values. POOMA Arrays extend built-in C++ arrays by supporting a wider variety of domains, automatically handling memory allocation, and having first-class status. For example, they may be used as operands and in assignments. In this chapter, we introduce the concept of containers, the mathematical concept of arrays, and the POOMA implementation of Arrays. Before illustrating how to declare Arrays, we introduce Domains, which specify the sets of indices. After describing how to declare the various types of Domains, we describe how to declare and use Arrays.

5.1. Containers

A *container class* is a class whose main purpose is to hold objects. These stored objects, called *container values* or more simply “values” or “elements”, may be accessed and changed, usually using indices. “Container class” is usually abbreviated as “container”.

The six POOMA containers can be categorized into two groups. Mathematical containers include Tensors, TinyMatrixs, and Vectors, which model tensors, matrices, and vectors, respectively. Storage containers include Arrays, DynamicArrays, and Fields. In this chapter, we focus on simplest of these: Arrays. DynamicArrays are also described.

C has built-in arrays, and the C++ Standard Library provides maps, vectors, stacks, and other containers, but the POOMA containers better model scientific computing concepts and provide more functionality. They automatically handle memory allocation and deallocation and can be used in expressions and on the left-hand side of assignments. Since POOMA containers separate the concepts of accessing and using values from the concept of storing values, value storage can be optimized to specific needs. For example, if most of an Array’s values are known to be identical most of the time, a compressible engine can be used. Whenever all the array’s values are identical, it stores only one value. At other times, it stores all the values. Engines will be discussed in Chapter 6.

5.2. Arrays

Mathematically, an array maps domain indices to values. Usually, the domain consists of

a one-dimensional integral interval or it may be a multidimensional domain. POOMA's `Array` container class implements this idea. Given an index, i.e., a position in an `Array`'s `Domain`, it returns the associated value, either by returning a stored value or by computing it. The indices are usually integral tuples but need not be zero-based or even consist of all possible integral tuples in a multidimensional range. Using indices permits constant-time access to values although computing a particular value may require significant time.

POOMA `Arrays` are *first-class objects* so they can be used more easily than built-in C++ arrays. For example, `Arrays` can be used as operands and in assignment statements. The statement `a = a + b;` adds corresponding values of `Arrays` `a` and `b`, assigning the sums to the `Array` `a`. The statement treats each array as an object, rather than requiring the use of one or more loops to access individual values. Data-parallel statements such as this are further discussed in Chapter 7. `Arrays` also handle their own memory allocation and deallocation. For example, the `Array` declaration `Array<2, double, Brick> a(verDomain)` creates an `Array` `a`, allocating whatever memory it needs. When `a` goes out of scope, it and its memory are automatically deallocated. Automatic memory allocation and deallocation also eases copying.

Individual `Array` values can be accessed using parentheses, not square brackets, as for C++ arrays. For example, `a(3,4)` yields the value at position (3,4) of `a`'s two-dimensional domain.

5.3. Domains

A *domain* specifies the set of points on which an array can define values. These indices are the arguments placed within parentheses to select particular values, as described previously. A domain supported both by `Arrays` and by built-in C++ arrays is the interval $[0, n-1]$ of integers containing all integers $\{0, 1, 2, \dots, n-1\}$. For C++, every integer in the interval must be included, and the minimum index must be zero. POOMA expands the set of permissible domains to support intervals with nonzero minimal indices, nonzero strides, and other options.

In POOMA, `Domain` classes implement domains. There are four different categories:

`Loc`

`Domain` with a single point.

Interval

`Domain` with an integral interval $[a,b]$.

Range

`Domain` with an integral interval $[a,b]$ and an integral stride s indicating the gap between indices: $\{a, a+s, a+2s, \dots, b\}$.

Grid

`Domain` with an ascending or descending sequence of integral values. The sequence elements must be individually specified.

One-dimensional and multidimensional versions of the categories are supported. A multidimensional `Domain` consists of the direct product of one-dimensional `Domains`. For example, the first dimension of a two-dimensional interval $[0,3] \times [2,9]$ is the interval $[0,3]$, and its second dimension is the interval $[2,9]$. Its indices are ordered pairs such as $(0,2)$, $(0,3)$, $(1,2)$, $(1,9)$, and $(3,7)$.

Many domains can be represented using domain triplets. That is, a *domain triplet* $[\text{begin}:\text{end}:\text{stride}]$ represents the mathematical set $\{\text{begin}, \text{begin} + \text{stride}, \text{begin} + 2\text{stride}, \dots, \text{end}\}$, where end is in the set only if it equals begin plus some integral multiple of stride . If the stride is negative, its beginning index begin should at least be as large as end if the interval is to be nonempty. The stride can be zero only if begin and end are equal. There are lots of ways to represent an empty interval, e.g., $[1:0:1]$ and $[23,34,-1]$, and POOMA will accept them, but they are all equivalent. The domain triplet notation is easily extended to multiple dimensions by separating different dimension's intervals with commas. For example, $[2:4:2,6:4:-2]$ contains $(2,6)$, $(2,4)$, $(4,6)$, and $(4,4)$.

All the `Domain` categories listed above except `Grid` can be represented using domain triplet notation. Since the triplet $[7:7:1]$ represents $\{7\}$, or more simply 7, it can also represent the one-dimensional `Loc` $\langle 1 \rangle (7)$. Multidimensional `Locs` are similarly represented. For example, $[0:0:1,10:10:1,2:2:1]$ represents `Loc` $\langle 3 \rangle (0,10,2)$, but it is frequently abbreviated as $[0,10,2]$. An `Interval` $[a,b]$ has unit stride: $[a:b:1]$, while a `Range` has specific stride s , e.g., $[a:b:s]$.

`Domains` can be constructed by combining `Domains` with smaller dimension. For example, since a two-dimensional `Interval` is the direct product of two one-dimensional `Intervals`, it can be specified using two one-dimensional `Intervals`. For example, `Interval` $\langle 2 \rangle (\text{Interval}\langle 1 \rangle (2,3), \text{Interval}\langle 1 \rangle (4,5))$ creates a $[2:3:1,4:5:1]$ `Domain`. The resulting dimensionality equals the sum of the components' dimensions. For example, a four-dimension `Loc` can be specified using three- and one-dimension `Locs` or using four one-dimension

`Locs`. If fewer dimensions than the created object's dimensionality, the last dimensions are unspecified and uninitialized. `Locs`, `Intervals`, `Ranges`, and `Grids` can all be composed from smaller similar components.

A `Domain` can be composed from smaller components with different types. A `Loc` object can be constructed from other `Loc` objects and integers. `Intervals`, `Ranges`, and `Grids` can be constructed using any of these types, `Locs`, and integers. For example, `Interval<3> a(Loc<2>(1,2), Interval<1>(3,5))` uses a two-dimensional `Loc` and a one-dimensional `Interval` to create a `[1:1:1,2:2:1,3:5:1]` `Domain`. During creation of a `Domain`, the type of each object is changed to the `Domain`'s type. In the example, `Loc<2>(1,2)` is first converted to an `Interval`.

`Domains` can participate in some arithmetic and comparison operations. For example, a `Domain`'s triplet can be shifted two units to the right by adding two. Multiplying a `Domain` by two multiplies its triplet's beginnings, endings, and strides by two. POOMA users rarely need to compare `Domains`, but we describe operating with the less-than operator on `Intervals`: `Interval d1 < Interval d2` if the length of `d1`'s interval is less than `d2`'s or, if equal, its beginning value is smaller. `Domain` arithmetic is frequently used with data-parallel statements and container views. These will be discussed in Chapter 7 and Chapter 8.

The current POOMA implementation supports `Domains` with dimensionality between one and seven, inclusive. Since most scientific computations use one, two, or three dimensions, this is usually sufficient. If more dimensions than seven are needed, they can be added to the source code.

5.3.1. Declaring Domains

Since `Domains` are mainly used to declare container domains, we focus on declaring `Domains`, deferring most discussion of their use. We subsequently describe a few `Domain` operations but most, including arithmetic operations with `Domains`, are described in Chapter 8.

All `Domain` declarations require a dimension template parameter `D`. This positive integer specifies the number of dimensions, i.e., rank, of the `Domain` and determines the length of the tuples for points in the `Domain`. For example, a three-dimensional `Domain` contains ordered triples, while a one-dimensional `Domain` contains singletons, or just integers. Multidimensional `Domains` are just the direct products of one-dimensional `Domains` so the techniques for declaring one-dimensional `Domains` carry over to multidimensional ones.

To declare a `Domain`, one must include the `Pooma/Domains.h` header file. However, most POOMA programs use `Domains` when constructing containers. The storage container header files automatically include `Pooma/Domains.h` so no explicit inclusion is usually necessary.

5.3.1.1. Locs

A `Loc<D>` is a `Domain` with just a single D-dimensional point. Although it is infrequently used as a container's domain, it is used to refer to a single point within another domain. Its beginning and ending points are the same, and its stride is one. One-dimensional `Locs` and integers are frequently interchanged.

Table 5-1. Declaring One-Dimensional Locs

constructor	result
<code>Loc<1>()</code>	indicates zero.
<code>Loc<1>(const Pooma::NoInit& no)</code>	creates an uninitialized <code>Loc<1></code> , to be assigned a value later.
<code>Loc<1>(const DT1& t1)</code>	creates a <code>Loc<1></code> with the integer converted from <code>t1</code> .
<code>Loc<1>(const DT1& t1, const DT2& t2)</code>	creates a <code>Loc<1></code> with the integer converted from <code>t1</code> . <code>t2</code> must equal <code>t1</code> .
<code>Loc<1>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates a <code>Loc<1></code> with the integer converted from <code>t1</code> . <code>t2</code> must equal <code>t1</code> , and <code>t3</code> is ignored.
DT1, DT2, and DT3 are template parameters.	

Constructors for one-dimensional `Locs` appear in Table 5-1. The empty constructor yields the zero point. The constructor taking a `Pooma::Init` object does not initialize the resulting `Loc` to any particular value. Presumably, the value will be assigned later. For small `Domains` such as `Locs`, the time savings from not initializing is small, but the functionality is still available. The constructor taking one argument with type `DT1` converts this argument to an integer to specify the point. The template type `DT1` may be any type that can be converted to an integer, e.g., `bool`, `char`, `int`, or `double`. The constructors taking two and three arguments of templated types facilitate converting an `Interval<1>` and a `Range<1>` into a `Loc<1>`. Since a `Loc` represents a single point, the `Interval`'s or `Range`'s first two arguments must be equal. The stride is ignored. Again, the templated types may be any type that can

be converted into an integer.

Table 5-2. Declaring Multidimensional Locs

constructor	result
<code>Loc<D>()</code>	indicates zero.
<code>Loc<D>(const Pooma::NoInit& no)</code>	creates an uninitialized Loc, to be assigned a value later.
<code>Loc<D>(const DT1& t1)</code>	creates a Loc using the given Domain object.
<code>Loc<D>(const DT1& t1, const DT2& t2)</code>	creates a Loc using the given Domain objects.
<code>Loc<D>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates a Loc using the given Domain objects.
<code>Loc<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4)</code>	creates a Loc using the given Domain objects.
<code>Loc<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5)</code>	creates a Loc using the given Domain objects.
<code>Loc<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6)</code>	creates a Loc using the given Domain objects.
<code>Loc<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6, const DT7& t7)</code>	creates a Loc using the given Domain objects.
D indicates the Loc's dimension. DT1, DT2, ... are template parameters.	

Constructors for multidimensional Locs appear in Table 5-2. D indicates the Loc's dimension. The first two constructors are similar to `Loc<1>`'s first two constructors, returning a representation of the zero point and returning an uninitialized point. The seven other constructors create a Loc using other Domain objects. These Domain objects, having types DT1, ..., DT7, can have any type that can be converted into

an integer, to a `Loc<1>`, or to a multidimensional `Domain` object that itself can be converted into a `Loc`. The total dimensionality of all the arguments' types should be at most `D`. For example, `Loc<5>(Range<1>(2,2,2), Loc<2>(2,3), Interval<1>(4,4))` creates a five-dimensional `Loc` [2,2,3,4,1] using a one-dimensional `Range`, a two-dimensional `Loc`, and a one-dimensional `Interval`. The final fifth dimension has an unspecified value, in this case 1. The one-dimensional `Range` is converted into the single integer two; its beginning and ending points must be the same. The two-dimensional `Loc` contributes values for the next two dimensions, while the `Interval` contributes its beginning point, which must be the same as its ending point. Note that the `Loc<1>` constructors taking two and three parameters ignore their second and third arguments, but this is not true for the multidimensional constructors.

5.3.1.2. Intervals

A one-dimensional `Interval` represents a set of integers within a mathematical *interval*. Multidimensional `Intervals` represent their multidimensional generalization, i.e., the direct product of one-dimensional intervals. `Intervals` are arguably the most commonly used POOMA `Domain`. A one-dimensional `Interval` has integral beginning and ending points and a unit stride.

Table 5-3. Declaring One-Dimensional Intervals

constructor	result
<code>Interval<1>()</code>	creates an empty, uninitialized interval.
<code>Interval<1>(const Pooma::NoInit& no)</code>	creates an uninitialized <code>Interval<1></code> , to be assigned a value later.
<code>Interval<1>(const DT1& t1)</code>	creates an <code>Interval<1></code> . See the text for an explanation.
<code>Interval<1>(const DT1& t1, const DT2& t2)</code>	creates an <code>Interval<1></code> with the integers converted from <code>t1</code> and <code>t2</code> .
<code>Interval<1>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates an <code>Interval<1></code> with the integers converted from <code>t1</code> and <code>t2</code> . <code>t3</code> must equal 1.
DT1, DT2, and DT3 are template parameters.	

`Interval<1>` constructors are patterned on `Loc<1>` constructors except that `Interval<1>`s can have differing beginning and ending points. See Table 5-3. The

default constructor creates an empty, uninitialized interval, which should not be used before assigning it values. If the one-parameter constructor's argument is a `Domain` object, it must be a one-dimensional `Domain` object which is converted into an `Interval` if possible; for example, it must have unit stride. If the one-parameter constructor's argument is not a `Domain` object, it must be convertible to an integer `e` and an interval `[0:e-1:1]` starting at zero is constructed. Note `e-1`, not `e`, is used so the `Interval<1>` has `e` indices. If two arguments are specified, they are assumed to be convertible to integers `b` and `e`, specifying the interval `[b:e:1]`. The three-parameter constructor is similar, with the third argument specifying a stride, which must be one.

Table 5-4. Declaring Multidimensional Intervals

constructor	result
<code>Interval<D>()</code>	creates an empty, uninitialized <code>Interval</code> , to be assigned a value later.
<code>Interval<D>(const Pooma::NoInit& no)</code>	creates an empty, uninitialized <code>Interval</code> , to be assigned a value later.
<code>Interval<D>(const DT1& t1)</code>	creates an <code>Interval</code> using the given <code>Domain</code> object.
<code>Interval<D>(const DT1& t1, const DT2& t2)</code>	creates an <code>Interval</code> using the given <code>Domain</code> objects.
<code>Interval<D>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates an <code>Interval</code> using the given <code>Domain</code> objects.
<code>Interval<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4)</code>	creates an <code>Interval</code> using the given <code>Domain</code> objects.
<code>Interval<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5)</code>	creates an <code>Interval</code> using the given <code>Domain</code> objects.
<code>Interval<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6)</code>	creates an <code>Interval</code> using the given <code>Domain</code> objects.

constructor

```
Interval<D>(const DT1&
t1, const DT2& t2, const
DT3& t3, const DT4& t4,
const DT5& t5, const DT6&
t6, const DT7& t7)
```

D indicates the Interval's dimension.
DT1, DT2, ... are template parameters.

result

creates an Interval using the given Domain objects.

Constructors for multidimensional Intervals closely follow constructors for multidimensional Locs. See Table 5-4. D indicates the Interval's dimension. The first two constructors both return empty, uninitialized intervals. The seven other constructors create an Interval using Domain objects. These Domain objects, having types DT1, ..., DT7, can have any type that can be converted into an integer, into a single-dimensional Domain object that can be converted into a single-dimensional Interval, or to a multidimensional Domain object that itself can be converted into an Interval. The total dimensionality of all the arguments' types should be at most D. One-dimensional Domain objects that can be converted into one-dimensional Intervals include Loc<1>s, Interval<1>s, and Range<1>s with unit strides. If the sum of the objects' dimensions is less than D, the intervals for the final dimensions are unspecified. See the last paragraph of Section 5.3.1.1 for an analogous example. Note that the Interval<1> constructors taking two and three parameters treat these arguments differently than the multidimensional constructors do.

5.3.1.3. Ranges

A one-dimensional Range generalizes an Interval by permitting a non-unit stride between integral members. A *range* is a set of integers in a mathematical interval [b,e] with a stride s between them: {a, a+s, a+2s, ..., b}. Ranges are generalized to D dimensions using the direct product of one-dimensional ranges.

Table 5-5. Declaring One-Dimensional Ranges**constructor**

```
Range<1>( )
Range<1>(const
Pooma::NoInit& no)
Range<1>(const DT1& t1)
```

result

creates an empty, uninitialized range.
creates an uninitialized Range<1>, to be assigned a value later.
creates a Range<1>. See the text for an explanation.

constructor

```
Range<1>(const DT1& t1,
const DT2& t2)
```

```
Range<1>(const DT1& t1,
const DT2& t2, const DT3&
t3)
```

DT1, DT2, and DT3 are template parameters.

result

creates a `Range<1>` with an interval specified by the integers converted from `t1` and `t2`.

creates a `Range<1>` by converting the arguments to integers `i1`, `i2`, and `i3` and then making a range `[i1:i2:i3]`.

`Range<1>` constructors are the same as `Interval<1>` constructors except they create ranges, not intervals. See Table 5-5. The default constructor creates an empty, uninitialized range, which should not be used before assigning it values. If the one-parameter constructor's argument is a `Domain` object, it must be a one-dimensional `Domain` object which is converted into a `Range` if possible. If the one-parameter constructor's argument is not a `Domain` object, it must be convertible to an integer `e` and a range `[0:e-1:1]` starting at zero is constructed. Note `e-1`, not `e`, is used so the `Interval<1>` has `e` indices. If two arguments are specified, they are assumed to be convertible to integers `b` and `e`, specifying the range `[b:e:1]`. The three-parameter constructor is similar, with the third argument specifying a stride.

Table 5-6. Declaring Multidimensional Ranges

constructor

```
Range<D>( )
```

```
Range<D>(const
Pooma::NoInit& no)
```

```
Range<D>(const DT1& t1)
```

```
Range<D>(const DT1& t1,
const DT2& t2)
```

```
Range<D>(const DT1& t1,
const DT2& t2, const DT3&
t3)
```

```
Range<D>(const DT1& t1,
const DT2& t2, const DT3&
t3, const DT4& t4)
```

result

creates an empty, uninitialized `Range`, to be assigned a value later.

creates an empty, uninitialized `Range`, to be assigned a value later.

creates a `Range` using the given `Domain` object.

creates a `Range` using the given `Domain` objects.

creates a `Range` using the given `Domain` objects.

creates a `Range` using the given `Domain` objects.

constructor	result
<code>Range<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5)</code>	creates a Range using the given Domain objects.
<code>Range<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6)</code>	creates a Range using the given Domain objects.
<code>Range<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6, const DT7& t7)</code>	creates a Range using the given Domain objects.
D indicates the Range's dimension.	
DT1, DT2, ... are template parameters.	

Constructors for multidimensional Ranges are the same as multidimensional Interval constructors except they create ranges, not intervals. See Table 5-6. D indicates the Range's dimension. The first two constructors return empty, uninitialized ranges. The seven other constructors create a Range using Domain objects. These Domain objects, having types DT1, ..., DT7, can have any type that can be converted into an integer, into a single-dimensional Domain object that can be converted into a single-dimensional Range, or to a multidimensional Domain object that itself can be converted into a Range. The total dimensionality of all the arguments' types should be at most D. One-dimensional Domain objects that can be converted into one-dimensional Ranges include `Loc<1>s`, `Interval<1>s`, and `Range<1>s`. If the sum of the objects' dimensions is less than D, the ranges for the final dimensions are unspecified. See the last paragraph of Section 5.3.1.1 for an analogous example. Note that the `Range<1>` constructors taking two and three parameters treat these arguments differently than the multidimensional constructors do.

5.3.1.4. Grids

`Locs`, `Intervals`, and `Ranges` all have regularly spaced integral values so they can be represented using *domain triplets*. One-dimensional `Grid` integral domains contain ascending or descending sequences of integers, with no fixed stride. For example, a `Grid<1>` may represent `{-13, 1, 4, 5, 34}`. `Grid<1>` is generalized to multidimensional `Grids` using the direct product of `Grid<1>` Domains.

Grids that can be represented using domain triplets can be constructed using techniques similar to other Domains, but irregularly spaced domains can be constructed using `IndirectionList<int>s`.

Table 5-7. Declaring One-Dimensional Grids

constructor	result
<code>Grid<1>()</code>	creates an empty, uninitialized grid.
<code>Grid<1>(const DT1& t1)</code>	creates a <code>Grid<1></code> . See the text for an explanation.
<code>Grid<1>(const DT1& t1, const DT2& t2)</code>	creates a <code>Grid<1></code> from the interval specified by the integers converted from <code>t1</code> and <code>t2</code> .
<code>Grid<1>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates a <code>Grid<1></code> from the domain triplet specified by the integers converted from <code>t1</code> , <code>t2</code> , and <code>t3</code> .
DT1, DT2, and DT3 are template parameters.	

To construct a `Grid<1>` that can also be represented by a domain triplet, use a `Grid<1>` constructor similar to those for `Interval<1>` and `Range<1>`. See Table 5-7 and the text explanations following Table 5-5 or Table 5-3.

`Grid<1>s` with irregularly spaced points can be constructed using `IndirectionList<int>s`. For example,

```
IndirectionList<int> list(4);
list(0) = 2;
list(1) = 5;
list(2) = 6;
list(3) = 9;
Grid<1> g(list);
```

constructs an empty `IndirectionList<int>`, fills it with ascending values, and then creates a `Grid<1>` containing {2, 5, 6, 9}. When creating a list, its size must be specified. Subsequently, its values can be assigned. `IndirectionLists` can also be initialized using one-dimensional Arrays:

```
Array<1,int,Brick> a1(Interval<1>(0,3));
a1(0) = 2; a1(1) = 5; a1(2) = 6; a1(3) = 9;
```

```

IndirectionList<int> il(a1);
Grid<1> g1(il);

```

The Array stores the integral points to include in the Grid<1> and is used to create the IndirectionList<int>, which itself is used to create the Grid<1>. Since the points are integers, the Array's type is `int`. Either a Brick or CompressibleBrick Engine should be used.

Table 5-8. Declaring Multidimensional Grids

constructor	result
<code>Grid<D>()</code>	creates an empty, uninitialized Grid, to be assigned a value later.
<code>Grid<D>(const DT1& t1)</code>	creates a Grid using the given Domain object.
<code>Grid<D>(const DT1& t1, const DT2& t2)</code>	creates a Grid using the given Domain objects.
<code>Grid<D>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates a Grid using the given Domain objects.
<code>Grid<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4)</code>	creates a Grid using the given Domain objects.
<code>Grid<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5)</code>	creates a Grid using the given Domain objects.
<code>Grid<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6)</code>	creates a Grid using the given Domain objects.
<code>Grid<D>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6, const DT7& t7)</code>	creates a Grid using the given Domain objects.
D indicates the Grid's dimension. DT1, DT2, ... are template parameters.	

Constructors for multidimensional `Grid`s are the same as multidimensional `Interval` constructors except they create `Grid`s, not intervals. See Table 5-8. `D` indicates the `Grid`'s dimension. The first constructor returns empty, uninitialized grids. The seven other constructors create an `Grid` using `Domain` objects. These `Domain` objects, having types `DT1`, ..., `DT7`, can have any type that can be converted into an integer, into a single-dimensional `Domain` object that can be converted into a single-dimensional `Grid`, or to a multidimensional `Domain` object that itself can be converted into an `Grid`. The total dimensionality of all the arguments' types should be at most `D`. One-dimensional `Domain` objects that can be converted into one-dimensional `Grid`s include `Loc<1>s`, `Interval<1>s`, `Range<1>s`, and `Grid<1>s`. If the sum of the objects' dimensions is less than `D`, the grids for the final dimensions are unspecified. See the last paragraph of Section 5.3.1.1 for an analogous example. Note that the `Grid<1>` constructors taking two and three parameters treat these arguments differently than the multidimensional constructors do.

5.3.2. Using Domains

Since an `Array` can be queried for its domain, we briefly describe some `Domain` operations. A fuller description, including arithmetic operations, occurs in Chapter 8. As we mentioned in Section 5.3.1, the `Pooma/Domains.h` header file declares `Domains`, but most storage container header files automatically include `Pooma/Domains.h` so no explicit inclusion is usually necessary.

Table 5-9. Some Domain Accessors

Domain member function	result
Multidimensional Domain Accessors	
<code>long size()</code>	returns the total number of indices.
<code>bool empty()</code>	returns <code>true</code> if and only if the <code>Domain</code> has no indices.
<code>D<1> operator[] (int dimension)</code>	returns the one-dimensional <code>Domain</code> for the specified dimension. The return type is a one-dimensional version of the <code>Domain</code> .
One-dimensional Domain Accessors	
<code>long length()</code>	returns the number of indices.
<code>int first()</code>	returns the beginning of the domain.

Domain member function	result
<code>int last()</code>	returns the ending of the domain.
<code>int min()</code>	returns the minimum index in the domain.
<code>int max()</code>	returns the maximum index in the domain.
 <code>D<1>::iterator begin()</code>	 returns a forward iterator pointing to the beginning domain index.
<code>D<1>::iterator end()</code>	returns a forward iterator pointing to the ending domain index.
<p>D abbreviates a particular Domain type, e.g., <code>Interval</code> or <code>Grid</code>. Other Domain accessors are described in Chapter 8.</p>	

Domain member functions are listed in Table 5-9. Functions applicable to both one-dimensional and multidimensional Domains are listed before functions that only applicable to one-dimensional Domains. The `size` member function yields the total number of indices in a given Domain. If and only if this number is zero, `empty` will yield `true`. A multidimensional domain `<D>` is the direct product of D one-dimensional Domains. The `operator[] (int dimension)` operator extracts the one-dimensional Domain corresponding to its parameter. For example, the three one-dimensional `Range<1>` Domains can be extracted from a `Range<3>` object `r` using `r[0]`, `r[1]`, and `r[2]`.

Domain accessors applicable only to one-dimensional Domains are listed in the second half of Table 5-9. The `length` member function, analogous to the multidimensional `size` function, returns the number of indices in the Domain. The `first` and `last` member functions return the domain's beginning and ending indices. The `begin` and `end` member functions return forward iterators pointing to these respective locations. They have type `D<1>::iterator`, where D abbreviates the Domain's type, e.g., `Interval` or `Grid`. The `min` and `max` member functions return the minimum and maximum indices in the Domain object, respectively. For `Loc<1>` and `Interval<1>`, these yield the same values as `first` and `last`, but `Range<1>` and `Grid<1>` can have their numerically largest index at the beginning of their Domains.

5.4. Declaring Arrays

A POOMA Array maps Domain indices to values. In this section, we describe how to declare Arrays. In the next section, we explain how to access individual values stored within an Array and how to copy Arrays.

Array values need not just be stored values, as C arrays have. The values can also be computed dynamically by the engine associated with the Array. We defer discussion of computing values to the next chapter discussing engines (Chapter 6). Therefore, when we mention “the values stored in an Array”, we implicitly mean “the values stored in or computed by the Array”.

Declaring an Array requires four arguments: the domain’s dimensionality, the type of values stored or computed, a specification how the values are stored or computed, and a Domain. The first three arguments are template parameters since few scientific programs need to (and no POOMA programs can) change these values while a program executes. For example, an Array cannot change the type of the values it stores, but an Array’s values can be copied into another Array having the desired type. Although scientific programs do not frequently change an array’s domain, they do frequently request a subset of the array’s values, i.e., a *view*. The subset is specified via a Domain so it is a run-time value. Views are presented in Chapter 8.

An Array’s first template parameter specifies its dimensionality. This positive integer *D* specifies its rank and has the same value as its domain’s dimensionality. Theoretically, an Array can have any positive integer, but the POOMA code currently supports a dimensionality of at most seven. For almost all scientific codes, a dimension of three or four is sufficient, but the POOMA code can be extended to support higher dimensions.

An Array’s second template parameter specifies the type of its stored or computed values. Common value types include `int`, `double`, `complex`, and `Vector`, but any type is permissible. For example, an Array’s values might be matrices or even other Arrays. The parameter’s default value is usually `double`, but it may be changed when the POOMA Toolkit is configured.

An Array’s third parameter specifies how its data is stored or computed by an Engine and its values accessed. The argument is a tag indicating a particular type of Engine. Permissible tags include `Brick`, `CompressibleBrick`, and `ConstantFunction`. The `Brick` tag indicates all Array values will be explicitly stored, just as built-in C arrays do. If an Array frequently stores exactly the same value in every position, a `CompressibleBrick` Engine, which reduces its space requirements to a constant whenever all its values are the same, is appropriate. A `ConstantFunction` Engine returns the same value for all indices. Some Engines compute values, e.g., applying a function to every value in another Engine. These

Engines are discussed in Chapter 6. To avoid being verbose in the rest of this chapter, we abbreviate “store or compute values” as “store values”. The engine parameter’s default value is usually `Brick`, but it may be changed when the POOMA Toolkit is configured.

Even though every `Array` container has an engine to store its values and permit access to individual values, the concept of an `Array` is conceptually separate from the concept of an engine. An engine’s role is low-level, storing values and permitting access to individual values. As we indicated above, the storage can be optimized to fit specific situations such as few nonzero values and computing values using a function applied to another engine’s values. An `Array`’s role is high-level, supporting access to groups of values. Arrays can be used in data-parallel expressions, e.g., adding all the values in one `Array` to all the values in another. (See Chapter 7 for more information.) Subsets of `Array` values, frequently used in data-parallel statements, can be obtained. (See Chapter 8 for more information.) Even though engines and Arrays are conceptually separate, higher-level Arrays provide access to lower-level Engines. Users usually have an `Array` create its `Engine(s)`, rarely explicitly creating `Engines` themselves. Also, Arrays support access to individual values. In short, POOMA users use Arrays, only dealing with how they are implemented (engines) when declaring them. For a description of Engines, see Chapter 6.

An `Array`’s one run-time argument is its domain. The domain specifies its extent and consequently how many values it can return. All the provided `Domain` objects are combined to yield an `Interval<D>`, where `D` matches the `Array`’s first template parameter. Since an `Interval` domain with its unit strides is used, there are no unaccessed “gaps” within the domain, wasting storage space. To use other domains to access an `Array`, first create it using an `Interval` domain and then take a view of it, as described in Chapter 8. As we mentioned above, the current POOMA code supports up to seven dimensions so at most seven `Domain` objects can be provided. If more dimensions are required, the POOMA code can be extended to the desired number of dimensions.

`Array` constructors are listed in Table 5-10. An `Array`’s three template parameters for dimensionality, value type, and engine type are abbreviated `D`, `T`, and `E`. Template parameters for domain types are named `DT1`, ..., `DT7`. The first constructor, with no domain arguments, creates an empty, uninitialized `Array` for which a domain must be specified before it is used. Specify the array’s domain using its `initialize` function. The next seven constructors combine their domain arguments to compute the resulting `Array`’s domain. These are combined in the same way that multidimensional `Intervals` are constructed. (See Table 5-4 and the following text.) The domain objects, having types `DT1`, ..., `DT7`, can have any type that can be converted into an integer, into a single-dimensional `Domain` object that can be converted into

a single-dimensional `Interval`, or to a multidimensional `Domain` object that itself can be converted into an `Interval`. The total dimensionality of all the arguments' types should *equal* `D`, unlike `Interval` construction which permits total dimensionality less than or equal to `D`. One-dimensional `Domain` objects that can be converted into one-dimensional `Intervals` include `Loc<1>s`, `Interval<1>s`, and `Range<1>s` with unit strides. To initialize all of an `Array`'s values to a specific value, use one of the final seven constructors, each taking a particular value, wrapped as a `ModelElement`. These constructors use the given domain objects the same way as the preceding constructors but assign `model` to every `Array` value. `model`'s type is `ModelElement<T>`, rather than `T`, to differentiate it from an `int`, which can also be used to specify a domain object. `ModelElement` just stores an element of any type `T`, which must match the `Array`'s value type `T`.

Table 5-10. Declaring Arrays

Array declaration	result
<code>Array<D,T,E>()</code>	creates an empty, uninitialized <code>Array</code> which must be <code>initialize()</code> d before use.
<code>Array<D,T,E>(const DT1& t1)</code>	creates an <code>Array</code> using the given <code>Domain</code> object or integer.
<code>Array<D,T,E>(const DT1& t1, const DT2& t2)</code>	creates an <code>Array</code> using the given <code>Domain</code> objects and integers.
<code>Array<D,T,E>(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates an <code>Array</code> using the given <code>Domain</code> objects and integers.
<code>Array<D,T,E>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4)</code>	creates an <code>Array</code> using the given <code>Domain</code> objects and integers.
<code>Array<D,T,E>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5)</code>	creates an <code>Array</code> using the given <code>Domain</code> objects and integers.
<code>Array<D,T,E>(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6)</code>	creates an <code>Array</code> using the given <code>Domain</code> objects and integers.

Array declaration

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
DT3& t3, const DT4& t4,
const DT5& t5, const DT6&
t6, const DT7& t7)
```

```
Array<D,T,E>(const DT1&
t1, const
ModelElement<T>& model)
```

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
ModelElement<T>& model)
```

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
DT3& t3, const
ModelElement<T>& model)
```

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
DT3& t3, const DT4& t4,
const ModelElement<T>&
model)
```

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
DT3& t3, const DT4& t4,
const DT5& t5, const
ModelElement<T>& model)
```

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
DT3& t3, const DT4& t4,
const DT5& t5, const DT6&
t6, const
```

```
ModelElement<T>& model)
```

```
Array<D,T,E>(const DT1&
t1, const DT2& t2, const
DT3& t3, const DT4& t4,
const DT5& t5, const DT6&
t6, const DT7& t7, const
ModelElement<T>& model)
```

result

creates an Array using the given Domain objects and integers.

creates an Array using the given Domain object or integer and then initializes all entries using model.

creates an Array using the given Domain objects and integers and then initializes all entries using model.

creates an Array using the given Domain objects and integers and then initializes all entries using model.

creates an Array using the given Domain objects and integers and then initializes all entries using model.

creates an Array using the given Domain objects and integers and then initializes all entries using model.

creates an Array using the given Domain objects and integers and then initializes all entries using model.

creates an Array using the given Domain objects and integers and then initializes all entries using model.

Array declaration**result**

Template parameters `D`, `T`, and `E` indicates the Array's dimension, value type, and Engine type, respectively. `DT1`, ..., `DT7` indicate domain types or integers.

We illustrate creating Arrays. To create a three-dimensional Array `a` explicitly storing double floating-point values, use

```
Interval<1> D(6);
Interval<3> I3(D,D,D);
Array<3,double,Brick> a(I3);
```

The template parameters specify its dimensionality, the type of its values, and a `Brick Engine` type, which explicitly stores values. Its domain, which must have three dimensions, is specified by an `Interval<3>` object which consists of `[0,5]` intervals for all its three dimensions. Since `double` and `Brick` are usually the default template parameters, they can be omitted so these declarations are equivalent:

```
Array<3,double> a_duplicate1(I3);
Array<3> a_duplicate2(I3);
```

To create a similar Array with a domain of `[0:1:1, 0:2:1, 0:0:1]`, use

```
Array<3> b(2,3,1);
```

since specifying an integer `i` indicates a one-dimensional zero-based `Interval [0:i-1:1]`. To store booleans, specify `bool` as the second template argument:

```
Array<2,bool> c(2,3);
```

To specify a default Array value of `true`, use `ModelElement<bool>(true)`:

```
Array<2,bool> c(2,3, ModelElement<bool>(true));
```

To create a one-dimensional Array containing seven doubles all equaling π , use

```
const double pi = 4.0*atan(1.0);
```

```
Array<1,double,CompressibleBrick>
  d(7, ModelElement<double>(pi));
```

We use a `CompressibleBrick Engine`, rather than a `Brick Engine`, so all seven values will be stored in one location rather than in seven separate locations when they are all the same.

An uninitialized `Array`, created using its parameter-less constructor, must have a specified domain before it can be used. For example, one must use the parameter-less `Array` constructor when creating an array of `Arrays` using `new` so their domains must be specified. (It would probably be better to create an `Array of Arrays` since memory allocation and deallocation would automatically be handled.) `Array`'s `initialize` functions accept the same set of domain object specifications and model elements that the `Array` constructors do, creating the specified domain. See Table 5-11. For example, both `a` and `b` are two-dimensional `Arrays` of `floats` with a `[2:7:1,-2:4:1]` domains:

```
// Create an Array and its domain.
Array<2,float,Brick> a(Interval<1>(2,7),
                      Interval<1>(-2,4));

// Create an Array without a domain and then specify
// its domain.
Array<2,float,Brick> b();
b.initialize(Interval<1>(2,7), Interval<1>(-2,4));
```

Invoking `initialize` on an `Array` with an existing domain yields unspecified behavior. All `Array` values may be lost and memory may be leaked.

Table 5-11. Initializing Arrays' Domains

An Array's `initialize` member functions sets its domain and should be invoked only for an array created without a domain. It returns nothing.

initialize declaration	result
<code>initialize(const DT1& t1)</code>	creates the Array's domain using the given <code>Domain</code> object or integer.
<code>initialize(const DT1& t1, const DT2& t2)</code>	creates the Array's domain using the given <code>Domain</code> objects and integers.

An Array's initialize member functions sets its domain and should be invoked only for an array created without a domain. It returns nothing.

initialize declaration	result
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3)</code>	creates the Array's domain using the given Domain objects and integers.
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4)</code>	creates the Array's domain using the given Domain objects and integers.
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5)</code>	creates the Array's domain using the given Domain objects and integers.
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6)</code>	creates the Array's domain using the given Domain objects and integers.
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6, const DT7& t7)</code>	creates the Array's domain using the given Domain objects and integers.
<code>initialize(const DT1& t1, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain object or integer and then initializes all entries using model.
<code>initialize(const DT1& t1, const DT2& t2, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain objects and integers and then initializes all entries using model.
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain objects and integers and then initializes all entries using model.
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain objects and integers and then initializes all entries using model.

An Array's `initialize` member functions sets its domain and should be invoked only for an array created without a domain. It returns nothing.

initialize declaration	result
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain objects and integers and then initializes all entries using <code>model</code> .
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain objects and integers and then initializes all entries using <code>model</code> .
<code>initialize(const DT1& t1, const DT2& t2, const DT3& t3, const DT4& t4, const DT5& t5, const DT6& t6, const DT7& t7, const ModelElement<T>& model)</code>	creates the Array's domain using the given Domain objects and integers and then initializes all entries using <code>model</code> .
Template parameters <code>DT1, ..., DT7</code> indicate domain types or integers.	

5.5. Using Arrays

In the previous section, we explained how to declare and initialize Arrays. In this section, we explain how to access individual values stored within an Array and how to copy Arrays. In Chapter 7, we explain how to use entire Arrays in data-parallel statements, including how to print them. In Chapter 8, we extend this capability to work on subsets.

In its simplest form, an Array stores individual values, permitting access to these values. For a C++ array, the desired index is specified within square brackets following the array's name. For POOMA Arrays, the desired index is specified within parentheses following the Array's name. The same notation is used to read and write values. For

example, the following code prints the initial value at index (2,-2) and increments its value, printing the new value:

```
Array<2,int,Brick> a(Interval<1>(0,3),
                    Interval<1>(-2,4),
                    ModelElement<int>(4));
std::cout << a(2,-2) << std::endl;
++a(2,-2);
std::cout << a(2,-2) << std::endl;
```

4 and then 5 are printed. An index specification for an `Array` usually has as many integers as dimensions, all separated by commas, but the `Array`'s engine may permit other notation such as using strings or floating-point numbers.

For read-only access to a value, use the `read` member function, which takes the same index notation as its nameless read-write counterpart:

```
std::cout << a.read(2,-2) << std::endl;
```

Using `read` sometimes permits the optimizer to produce faster executing code.

Copying Arrays requires little execution time because Arrays have *reference semantics*. That is, a copy of an `Array` and the `Array` itself share the same underlying data. Changing a value in one changes it in the other. Example 5-1 illustrates this behavior. Initially, all values in the array `a` are 4. The `b` array is initialized using `a` so it shares the same values as `a`. Thus, changing the former's value also changes the latter's value. Function arguments are also initialized so changing their underlying values also changes the calling function's values. For example, the `changeValue` function changes the value at index (0,0) for both its function argument and `a`.

Example 5-1. Copying Arrays

```
#include "Pooma/Pooma.h"
#include "Pooma/Arrays.h"
#include <iostream>

// Changes the Array value at index (0,0).
void changeValue(Array<2,int,Brick>& z)
{ z(0,0) = 6; }
```

```

int main(int argc, char *argv[])
{
    Pooma::initialize(argc,argv);

    Array<2,int,Brick> a(3,4, ModelElement<int>(4));
    std::cout << "Initial value:\n";
    std::cout << "a: " << a(0,0) << std::endl;

    // Array copies share the same underlying values.

    // Explicit initialization uses reference semantics
    // so changing the copy's value at (0,0) also
    // changes the original's value.
    Array<2,int,Brick> b(a);
    b(0,0) = 5;
    std::cout << "After explicit initialization.\n";
    std::cout << "a: " << a(0,0) << std::endl;
    std::cout << "b: " << b(0,0) << std::endl;

    // Initialization of function arguments also uses
    // reference semantics.
    std::cout << "After function call:\n";
    changeValue(a);
    std::cout << "a: " << a(0,0) << std::endl;
    std::cout << "b: " << b(0,0) << std::endl;

    Pooma::finalize();
    return 0;
}

```

The separation between a higher-level `Array` and its lower-level `Engine` storage permits fast copying. An `Array`'s only data member is its engine, which itself has reference semantics that increments a reference-counted pointer to its data. Thus, copying an `Array` requires creating a new object with one data member and incrementing a pointer's reference count. Destruction is similarly inexpensive.

Array assignment does not have reference semantics. Thus, the assignment `a = b` ensures that all of `a`'s values are the same as `b` at the time of assignment only. Subsequent

changes to `a`'s values do not change `b`'s values or vice versa. Assignment is more expensive than creating a reference. Creating a reference requires creating a very small object and incrementing a reference-counted pointer. An assignment requires storage for both the left-hand side and right-hand side operands and traversing all of the right-hand side's data.

The `Array` class has internal type definitions and constants useful for both compile-time and run-time computations. See Table 5-12. These may be accessed using the `Array`'s type and the scope resolution operator (`::`). The table begins with a list of internal type definitions, e.g., `Array<D, T, E>::This_t`. A *layout* maps a domain index to a particular processor and memory used to compute the associated value. The two internal enumerations `dimensions` and `rank` both record the `Array`'s dimension.

Table 5-12. Array Internal Type Definitions and Compile-Time Constants

internal type or compile-time constant	meaning
<code>This_t</code>	the <code>Array</code> 's type <code>Array<D, T, E></code> .
<code>Engine_t</code>	the <code>Array</code> 's <code>Engine</code> type <code>Engine<D, T, E></code> .
<code>EngineTag_t</code>	the <code>Array</code> 's <code>Engine</code> 's tag <code>E</code> .
<code>Element_t</code>	the type <code>T</code> of values stored in the <code>Array</code> .
<code>ElementRef_t</code>	the type of references to values stored in the <code>Array</code> (usually <code>T&</code>).
<code>Domain_t</code>	the type of the <code>Array</code> 's domain.
<code>Layout_t</code>	the type of the <code>Array</code> 's layout.
<code>const int dimensions</code>	the number <code>D</code> of dimensions of the <code>Array</code> .
<code>const int rank</code>	synonym for <code>dimensions</code> .

The `Array` class has several member functions easing access to its domain and engine. The first ten functions listed in Table 5-13 ease access to `Array` domains. The first three functions are synonyms all returning the `Array`'s domain, which has type `Array<D, T, E>::Domain_t` (abbreviated `Domain_t` in the table). The next seven functions query the domain. `first`, `last`, and `length` return the first index, last index, and number of indices for the specified dimension. The domain's dimensions are numbered `0, 1, ..., Array<D, T, E>::dimensions-1`. If these values are needed for all dimensions, use `firsts`, `lasts`, and `lengths`. The returned

`Loc<D>s` have `D` entries, one for each dimension. `size` returns the total number of indices in the entire domain. This is the product of all the dimensions' lengths. The `layout` member function returns the `Array`'s layout, which specifies the mapping of indices to processors and memory. The last two functions return the `Array`'s engine.

Table 5-13. Array Accessors

Array member function	result
<code>Domain_t domain()</code>	returns the <code>Array</code> 's domain.
<code>Domain_t physicalDomain()</code>	returns the <code>Array</code> 's domain.
<code>Domain_t totalDomain()</code>	returns the <code>Array</code> 's domain.
<code>int first(int dim)</code>	returns the first index value for the specified dimension.
<code>int last(int dim)</code>	returns the last index value for the specified dimension.
<code>int length(int dim)</code>	returns the number of indices (including endpoints) for the specified dimension.
<code>Loc<Dim> firsts()</code>	returns the first index values for all the dimensions.
<code>Loc<Dim> lasts()</code>	returns the last index values for all the specified dimensions.
<code>Loc<Dim> lengths()</code>	returns the numbers of indices (including endpoints) for all the specified dimensions.
<code>long size()</code>	returns the total number of indices in the domain.
<code>Layout_t layout()</code>	returns the <code>Array</code> 's layout.
<code>Engine_t engine()</code>	returns the <code>Array</code> 's engine.
<code>const Engine_t engine()</code>	returns the <code>Array</code> 's engine.
Internal type definitions, e.g., <code>Domain_t</code> , are listed here without the class type prefix <code>Array<D, T, E>::</code> .	

We illustrate using `Array` member functions in Example 5-2. The program computes the total number of `Array`'s indices, comparing the result with invoking its `size` method. Since the `Array`'s name is `a`, `a.size()` returns its size. The `computeArraySize` function also computes the `Array`'s size. This templated function uses its three template parameters to accept any `Array`, regardless of its dimension, value type, or `Engine` tag. It begins by obtaining the range of indices for all

dimensions and their lengths. Only the latter is necessary for the computation, but using the former further illustrates using member functions. The domain's size is the product of the length of each dimension. Since the lengths are stored in the `Loc<D> lens`, `lens[d]` is a `Loc<1>`, for which its `first` member function extracts the length. The `length` Array member function is used in the `PAssert`.

Example 5-2. Using Array Member Functions

```
#include "Pooma/Pooma.h"
#include "Pooma/Arrays.h"
#include <iostream>

// Print an Array's Size

// This program illustrates using the Array member
// functions.  computeArraySize's computation is
// redundant because Array's size() function computes
// the same value, but it illustrates using Array
// member functions.

template <int Dim,typename Type,typename EngineTag>  (1)
inline
long computeArraySize(const Array<Dim,Type,EngineTag>& a)
{
    const Loc<Dim> fs = a.firsts();  (2)
    const Loc<Dim> ls = a.lasts();
    const Loc<Dim> lens = a.lengths();
    long size = 1;
    for (int d = 0; d < Dim; ++d) {
        size *= lens[d].first();  (3)
        // Check that lengths() and our computed lengths agree.
        PAssert((ls[d]-fs[d]+1).first()==a.length(d));  (4)
    }
    return size;
}

int main(int argc, char *argv[])
{
    Pooma::initialize(argc,argv);
```

```

Array<3,int,Brick> a(3,4,5, ModelElement<int>(4));
PAssert(computeArraySize(a) == a.size());  (5)
std::cout <<
    "The array's size is " << a.size() << ".\n";

Pooma::finalize();
return 0;
}

```

- (1) These template parameters, used in the `Array` parameter's type, permit the function to work with any `Array`.
- (2) We invoke these three member functions using the `Array`'s name `a`, a period, and the functions' names. These functions return `Locs`.
- (3) `lens[d]` returns a `Loc<1>` for dimension `d`'s length. Invoking `Loc<1>`'s `first` method yields its value.
- (4) This comparison is unnecessary but further illustrates using member functions.
- (5) The `size` is invoked by prepending the `Array`'s name followed by a period. This assertion is unnecessary, but the `computeArraySize` function further illustrates using member functions.

5.6. DynamicArrays

Arrays have fixed domains so the set of valid indices remains fixed after creation. The *DynamicArray* class supports one-dimensional domains that can be resized even while the array is used.

`DynamicArray`'s interface extends the one-dimensional interface of an `Array` by adding member functions to change the domain's size. It is declared in `Pooma/DynamicArrays.h`. A `DynamicArray` has two, not three, template parameters, omitting the array's dimensionality which must be one. The first parameter `T` specifies the type of stored values. Its default value is usually `double`, but this may be changed when the POOMA Toolkit is configured. The second parameter specifies an `Engine` via an `Engine` tag. The engine must support a domain with dynamic resizing. For example, the `Dynamic Engine` is analogous to a one-dimensional `Brick Engine`.

supporting a dynamically-resizable domain. It is also usually the default value for this tag. For example, `DynamicArray<> d0(1);`, `DynamicArray<double> d1(1);`, and `DynamicArray<double, Dynamic> d2(1);` all declare the same `DynamicArrays` explicitly storing one `double` value. A `DynamicArray` automatically allocates its initial memory and deallocates its final memory, just as an `Array` does.

The `create` and `destroy` member functions permit changing a `DynamicArray`'s domain. Table 5-14 lists these member functions but omits functions exclusively used in distributed computation. When making the domain larger, new indices are added to the end of the one-dimensional domain and the corresponding values are initialized with the default value for `T`. Existing values are copied.

Table 5-14. Changing a `DynamicArray`'s Domain

<code>DynamicArray</code> member function	description
<code>void create(int num)</code>	extend the current domain by the requested number of elements.
<code>void destroy(const Dom& killList)</code>	remove the values specified by the indices in the given <code>Domain</code> argument. The “Backfill” method moves values from the end of the domain to replace the deleted values.
<code>void destroy(Iter killBegin, Iter killEnd)</code>	remove the values specified by the indices in the container range <code>[begin,end)</code> specified by the random-access iterators. The “Backfill” method moves values from the end of the domain to replace the deleted values.
<code>void destroy(const Dom& killList, const DeleteMethod& method)</code>	remove the values specified by the indices in the given <code>Domain</code> argument. Deleted values can be replaced by <code>BackFill</code> 'ing, i.e., moving data from the domain's end to fill removed values, or by <code>ShiftUp</code> 'ing, i.e., compacting all data but maintaining the relative ordering.

DynamicArray member function	description
<code>void destroy(Iter killBegin, Iter killEnd, const DeleteMethod& method)</code>	remove the values specified by the indices in the container range [begin,end) specified by the random-access iterators. Deleted values can be replaced by <code>BackFill</code> 'ing, i.e., moving data from the domain's end to fill removed values, or by <code>ShiftUp</code> 'ing, i.e., compacting all data but maintaining the relative ordering.

This table omits member functions designed for distributed computation.

The `destroy` member function deletes the specified indices. The indices may be specified using either a `Domain` object (`Interval<1>`, `Range<1>`, or `IndirectionList`) or by random-access iterators pointing into a container. For example, every other value from a ten-value array `d` might be removed using `Range<1>(0,9,2)`. Alternatively,

```
int killList[] = {0, 2, 4, 6, 8};
d.destroy(killList, killList+5);
```

performs the same deletions. As indices are removed, other indices are moved into their positions. Using the `BackFill` method moves the last index and its associated value into deleted index's position. Thus, the total number of indices is decreased by one, but the indices are reordered. Using the `ShiftUp` method ensures the order of the indices is preserved by "shifting" all values left (or up) so all "gaps" between indices disappear. For example, consider removing the first index from a domain.

original indices:	0 1 2 3
destroy using <code>BackFill</code> :	3 1 2
destroy using <code>ShiftUp</code> :	1 2 3

The `BackFill` moves the rightmost index 3 into the removed index 0's position. The `ShiftUp` moves all the indices one position to the left. This illustrates that `BackFill` moves exactly as many indices as are deleted, while `ShiftUp` can shift all indices in a domain. Thus, `BackFill` is the default method. When multiple indices are deleted, they are deleted from the last (largest) to the first (smallest). When using the `BackFill` method, some indices may be moved repeatedly. For example, consider removing indices 0 and 2 from original indices of 0 1 2 3. Removing 2 yields 0

1 3 because 3 is moved into 2's position. Removing 0 yields 3 1 because 3 is again moved. Use an object with the desired type to indicate which fill method is desired, i.e., `BackFill()` or `ShiftUp()`.

We illustrate `DynamicArray` resizing in Example 5-3. `DynamicArrays` are declared in `Pooma/DynamicArrays.h`, not `Pooma/Arrays.h`. Their declarations require two, not three, template arguments because the array must be one-dimensional. The three arrays, each having one `double` value, are equivalent. (The POOMA Toolkit can be configured to support different default template values.) Invoking `d0's create` with an argument of five increases its domain size from one to six. The additional indices are added to the end of the domain so the value at index 0 is not changed. To illustrate which indices are removed and which indices are reordered, the program first sets all values equal to their indices. This illustrates that `DynamicArray` values are accessed the same way as `Array` values. For example, `d0(i)` accesses the i^{th} value. The `destroy` member function removes every other index from the array because the one-dimensional `Range` specifies the domain's entire interval with a stride of 2. The `BackFill` function call creates a `BackFill` object indicating the `BackFill` method should be used. We illustrate the steps of this method:

original indices:	0 1 2 3 4 5
delete index 4:	0 1 2 3 5
delete index 2:	0 1 5 3
delete index 0:	3 1 5

Since multiple indices are specified, the rightmost one is removed first, i.e., index 4. The rightmost index 5 is moved into 4's position. When removing index 2, the index originally at 5 is again moved into 2's position. Finally, index 0 is replaced by index 3. The rest of the program repeats the computation, using the random-access iterator version of `destroy`. Since this `DynamicArray's` indices are specified using `ints`, the `killList` explicitly lists the indices to remove. The `destroy` call uses pointers to the beginning and end of the `killList` array to specify which of its indices to use. Since no replacement method is specified, the default `BackFill` method is used. All the `DynamicArrays's` unallocated memory is deallocated.

Example 5-3. Example Using `DynamicArrays`

```
#include "Pooma/Pooma.h"
#include "Pooma/DynamicArrays.h"    (1)
#include <iostream>
```

```

// Demonstrate using DynamicArrays.

int main(int argc, char *argv[])
{
    Pooma::initialize(argc,argv);

    // Create a DynamicArray with one element.  (2)
    DynamicArray<> d0(1);
    DynamicArray<double> d01(1);
    DynamicArray<double, Dynamic> d02(1);

    // Add five more elements.  (3)
    d0.create(5);
    // Store values in the array.
    for (int i = d0.domain().first(); i <= d0.domain().last(); ++i)
        d0(i) = i;  (4)

    // Delete every other element.  (5)
    d0.destroy(Range<1>(d0.domain().first(),d0.domain().last(),2),

    // Print the resulting array.
    std::cout << d0 << std::endl;

    // Use the iterator form of 'destroy.'
    DynamicArray<> d1(6);
    for (int i = d1.domain().first(); i <= d1.domain().last(); ++i)
        d1(i) = i;
    int killList[] = { 0, 2, 4 };  (6)
    d1.destroy(killList, killList+3);
    std::cout << d1 << std::endl;

    Pooma::finalize();
    return 0;
}

```

- (1) This header file declares `DynamicArrays`.
- (2) These three declarations yield equivalent `DynamicArrays`, storing one double value.

- (3) This `create` member function call adds five indices to the end of the domain.
- (4) `DynamicArray` values are accessed the same way as `Array` values.
- (5) The `Range` object specifies that every other index should be removed. The `Back-Fill ()` object is unnecessary since it is the default replacement method.
- (6) This `destroy` call is equivalent to the previous one but uses iterators.

Chapter 6. Engines

Each container has one or more `Engines` to store or compute its values. As we mentioned in Section 5.4, a container's role is high-level, supporting access to groups of values, and an engine's role is low-level, storing or computing values and supporting access to individual values. This separation permits optimizing space and computation requirements.

We begin this chapter by introducing the concept of an engine and how it is used. Then, we describe the various `Engines` that POOMA provides, separating them into engines that store values and engines that compute values.

6.1. The Concept

An engine performs the low-level value storage, computation, and element-wise access for a container. An engine has a domain and accessor functions returning individual elements. The `POOMA Engine` class and its specializations implement the engine concept. Given an index within the domain, an `Engine`'s `operator()` function returns the associated value, which can be used or changed. Its `read` member function returns the same value but permitting only use, not modification. The acceptable indices are determined by each `Engine`. Most accept indices specified using `int` and `Loc<D>` parameters, but an `Engine` might accept string or floating-point parameters. An `Engine`'s layout specifies maps its domain indices to the processors and memory used to store and compute the associated values.

Since an engine's main role is to return the individual values associated with specific domain indices, any implementation performing this task is an engine. POOMA `Engines` fall into three categories:

- `Engines` that store values.
- `Engines` that compute their values using other `Engines`' values.
- `Engines` that support distributed computation.

For example, the `Brick Engine` explicitly stores all its values, while the `CompressibleBrick` engine adds the feature of reducing its storage requirements if all these values are identical. A `UserFunction Engine` yields values by applying a *function object* to each value returned by another `Engine`. A `CompFwd Engine` projects components from another `Engine`. For example, `CompFwd` will use the second components of each `Vector` in an `Array` to form its own `Array`. Since

each container has at least one `Engine`, we can also describe the latter category as containers that compute their values using other containers' values. A `MultiPatch Engine` distributes its domain among various processors and memory spaces, each responsible for computing values associated with a portion, or patch, of the domain. The `Remote Engine` also supports distributed computation.

Just as multiple containers can use the same engine, multiple `Engines` can use the same underlying data. As we mentioned in Section 5.5, `Engines` have *reference semantics*. A copy of an `Engine` has a reference-counted pointer to the `Engine`'s data (if any exists). Thus, copying an `Engine` or a container requires little execution time. If an `Engine` has the same data as another `Engine` but it needs its own data to modify, the `makeOwnCopy` member function creates such a copy.

`Engines` are rarely explicitly declared. Instead a container is declared using an `Engine` tag, and the container creates the specified `Engine` to deal with its values. For example, a `Brick Engine` is explicitly declared as `Engine<D, T, Brick>`, but they are more frequently created by containers, e.g., `Array<D, T, Brick>`. An `Engine`'s first two template parameters specify the domain's dimensionality and the value type, as described in Section 5.4. Unlike container declarations, the third template parameter, the `Engine` tag, specifies which `Engine` specialization to use. For example, the `Brick Engine` tag indicates a `Brick Engine` should be used. Some `Engines`, such as `CompFwd`, are rarely declared even using `Engine` tags. Instead the `Array`'s `comp` and `readComp` member functions return views of containers using `CompFwd Engines`.

6.2. Types of Engines

In this section, we describe the different types of `Engines` and illustrate their creation, when appropriate. First, we describe `Engines` that explicitly store values and then `Engines` that compute values. See Table 6-1.

Table 6-1. Types of Engines

Engine tag	description
<hr/> Engines That Store	
<code>Brick</code>	explicitly stores all values; similar to C arrays.

Engine tag	description
<code>CompressibleBrick</code>	stores all values, reducing storage requirements when all values are identical.
<code>Dynamic</code>	is a one-dimensional <code>Brick</code> with dynamically resizable domain. This should be used with <code>DynamicArray</code> , not <code>Array</code> .
<hr/> Engines That Compute <hr/>	
<code>CompFwd</code>	extracts specified components of an engine's vectors, tensors, arrays, etc.; usually created using the <code>comp</code> container function.
<code>ConstantFunction</code>	makes a scalar value behave like a container.
<code>IndexFunction<FunctionObject></code>	makes the <code>FunctionObject</code> 's function of indices behave like a container.
<code>ExpressionTag<Expr></code>	evaluates an expression tree; usually created by data-parallel expressions.
<code>Stencil<Function, Expression></code>	applies a stencil computation (<code>Function</code>) to its input (<code>Expression</code>) which is usually a container; usually created by applying a <code>Stencil</code> object to a container. A stencil computation can use multiple neighboring input values.
<code>UserFunctionEngine<FunctionExpression></code>	applies the given function (or <i>function object</i>) to its input (<code>Expression</code>) which is usually a container; usually created by applying a <code>UserFunction</code> object to a container. The function implements a one-to-one mapping from its input to values.
<hr/> Engines for Distributed Computation <hr/>	
<code>MultiPatch<LayoutTag, EngineTag></code>	uses separate <code>EngineTag</code> <code>Engine</code> on each context (patch) specified by the given layout. This is the usual <code>Engine</code> for distributed computation.

Engine tag	description
<code>Remote<EngineTag></code>	runs the Engine specified by EngineTag on a specified context.
<code>Remote<Dynamic></code>	runs a Dynamic one-dimensional, resizable Engine on a specified context. This is a specialization of Remote.

Brick Engines explicitly store values just like C arrays. Compressible-Brick Engines optimize their storage requirements when all values are identical. Many Arrays use one of these two Engines. Bricks are the default Engines for Array and Field containers because they explicitly store each value. This explicit storage can require a large amount of space, particularly if all these values are the same. If all a compressible brick Engine's values are identical, the Engine stores that one value rather than many, many copies of the same value. These engines can both save time as well as space. Initializing a compressible engine requires setting only one value, not every value. Using less storage space may also permit more useful values to be stored in cache, improving cache performance. Reading a value in a compressed Engine using the `read` member function is as fast as reading a value in a Brick Engine, but writing a value always requires executing an additional `if` conditional. Thus, if an Engine infrequently has multiple different values during its life time, a CompressibleBrick Engine may be faster than a Brick Engine. If an Engine is created and its values are mostly read, not written, a CompressibleBrick Engine may also be faster. Otherwise, a Brick Engine may be preferable. Timing the same program using the two different Engine types will reveal which is faster for a particular situation. In distributed computing, many Engines may have few nonzero values so CompressibleBrick Engines may be preferable. For distributed computing, a container's domain is partitioned into regions each computed by a separate processor and Engine. If the computation is concentrated in sections of the domain, many Engines may have few, if any, nonzero values. Thus, CompressibleBrick Engines may be preferable for distributed computing.

Both Brick and CompressibleBrick Engines have `read` and `operator()` member functions taking `int` and `Loc` parameters. The parameters should match the Array's dimensionality. For example, if Array `a` has dimensionality 3, `a.read(int, int, int)` and `a(int, int, int)` should be used. The former returns a value that cannot be modified, while the latter can be changed. Using the `read` member function can lead to faster code. Alternatively, an index can be specified using a `Loc`. For example, `a.read(Loc<3>(1, -2, 5))` and `a(Loc<3>(1, -2, 5))` are equivalent to `a.read(1, -2, 5)` and `a(1, -2, 5)`.

The `Dynamic Engine` supports changing domain sizes while a program is executing. It is basically a one-dimensional `Brick`, explicitly storing values, but permitting the number and order of stored values to change. Thus, it supports the same interface as `Brick` except that all member functions are restricted to their one-dimensional versions. For example, `read` and `operator ()` take `Loc<1>` or one `int` parameter. In addition, the one-dimensional domain can be dynamically resized using `create` and `destroy`.

Chapter 7. Data-Parallel Expressions

In the previous chapters, we accessed container values one at a time. Accessing more than one value in a container required writing an explicit loop. Scientists and engineers commonly operate on sets of values, treated as an aggregate. For example, a vector is a one-dimensional collection of data and two vectors can be added together. A matrix is a two-dimensional collection of data, and a scalar and a matrix can be multiplied. A *data-parallel expression* simultaneously uses multiple container values. POOMA supports data-parallel expressions.

After introducing data-parallel expressions and statements, we present the corresponding POOMA syntax. Then we present its implementation, which uses expression-template technology. A naïve data-parallel implementation might generate temporary variables, cluttering a program's inner loops and slowing its execution. Instead, POOMA uses PETE, the Portable Expression Template Engine. Using expression templates, it constructs a parse tree of expressions and corresponding types, which is then quickly evaluated without the need for temporary variables.

7.1. Expressions with More Than One Container Value

Science and math is filled with aggregated values. A vector contains several components, and a matrix is a two-dimensional object. Operations on individual values are frequently extended to operations on these aggregated values. For example, two vectors having the same length are added by adding corresponding components. The product of two matrices is defined in terms of sums and products on its components. The sine of an array is an array containing the sine of every value in it.

A *data-parallel expression* simultaneously refers to multiple container values. Data-parallel statements, i.e., statements using data-parallel expressions, frequently occur in scientific programs. For example, the sum of two vectors v and w is written as $v+w$. Algorithms frequently use data-parallel syntax. Consider, for example, computing the total energy E as the sum of kinetic energy K and potential energy U . For a simple particle subject to the earth's gravity, the kinetic energy K equals $mv^2/2$, and the potential energy U equals mgh . These formulae apply to both an individual particle with a particular mass m and height h and to an entire field of particles with masses m and heights h . Our algorithm works with data-parallel syntax, and we would like to write the corresponding computer program using data-parallel syntax as well.

7.2. Using Data-Parallel Expressions

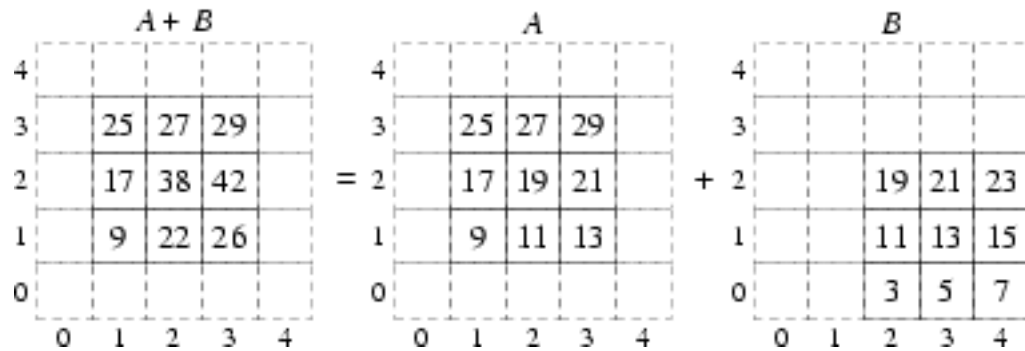
POOMA containers can be used in data-parallel expressions and statements. The basic guidelines are simple:

- The C++ built-in and mathematical operators operate on an entire container by operating element-wise on its values.
- Binary operators operate only on containers with the same domain types and by combining values with the same indices. If the result is a container, it has a domain equal to the left operand's domain.
- For assignment operators, the domains of the left operand and the right operand must have the same type and be conformable, i.e., have the “same shape”.

The data-parallel operators operate element-wise on containers' values. For example, if *A* is a one-dimensional array, $-A$ is a one-dimensional array with the same size such that the value at the i^{th} position equals $-A(i)$. If *A* and *B* are two-dimensional Arrays on the same domain, $A+B$ is an array on the same domain with values equaling the sum of corresponding values in *A* and *B*.

Binary operators operate on containers with the same domain types. The domain's indices need not be the same, but the result will have a domain equal to the left operand. For example, the sum of an Array *A* with a one-dimensional interval $[0,3]$ and an Array *B* with a one-dimensional interval $[1,2]$ is well-defined because both domains are one-dimensional intervals. The result is an Array with a one-dimensional interval $[0,3]$. Its first and last entries equal *A*'s first and last entries, while its middle two entries are the sums $A(1)+B(1)$ and $A(2)+B(2)$. We assume zero is the default value for the type of values stored in *B*. A more complicated example of adding two Arrays with different domains is illustrated in Figure 7-1. Code for these Arrays could be

```
Interval<1> H(0,2), I(1,3), J(2,4);
Array<2, double, Brick> A(I,I), B(J,H);
// ... fill A and B with values ...
... = A + B;
```

Figure 7-1. Adding Arrays with Different Domains

Adding Arrays with different domains is supported. Solid lines indicate the domains' extent. Values with the same indices are added.

Both A and B have domains of two-dimensional intervals so they may be added, but their domains' extent differ, as indicated by the solid lines in the figure. The sum has domain equal to the left operand's domain. Values with the same indices are added. For example, $A(2, 2)$ and $B(2, 2)$ are added. B's domain does not include index (1,1) so, when adding $A(1, 1)$ and $B(1, 1)$, the default value for B's value type is used. Usually this is 0. Thus, $A(1, 1) + B(1, 1)$ equals $9 + 0$.

Operations with both Arrays and scalar values are supported. Conceptually, a scalar value can be thought of as an Array with any desired domain and having the same value everywhere. For example, consider

```
Array<1, double, Brick> D(Interval<1>(7,10));
D += 2*D + 7;
```

$2*D$ obeys the guidelines because the scalar 2 can be thought of as an array with the same domain as D. It has the same value 2 everywhere. Likewise the conceptual domain for the scalar 7 is the same as $2*D$'s domain. Thus, $2*D(i) + 7$ is added to $D(i)$ wherever index i is in D's domain. In practice, the toolkit does not first convert scalar values to arrays but instead uses them directly in expressions.

Assignments to containers are also supported. The domain types of the assignment's left-hand side and its right-hand side must be the same. Their indices need not be the same, but they must correspond. That is, the domains must be *conformable*, or have the "same shape", i.e., have the same number of indices for each dimension. For example, the one-dimensional interval [0,3] is conformable to the one-dimensional interval [1,4] because they both have the same number of indices in each dimension. The domains of A and B, as declared

```
Interval<1> H(0,2), I(1,3), J(2,4), K(0,4);
Array<2, double, Brick> A(I,I), B(H,J), C(I,K);
```

are conformable because each dimension has the same number of indices. A and C are not conformable because, while their first dimensions are conformable, their second dimensions are not conformable. It has three indices while the other has five. We define *conformable containers* to be containers with conformable domains.

When assigning to a container, corresponding container values are assigned. (Since the left-hand side and the right-hand side are conformable, corresponding values exist.) In this code fragment,

```
Array<1, double, Brick> A(Interval<1>(0,1));
Array<1, double, Brick> B(Interval<1>(1,2));
A = B;
```

A(0) is assigned B(1) and A(1) is assigned B(2).

Assigning a scalar value to an Array also is supported, but assigning an Array to a scalar is not. A scalar value is conformable to any domain because, conceptually it can be viewed as an Array with any desired domain and having the same value everywhere. Thus, the assignment B = 3 ensures every value in B equals 3. Even though a scalar value is conformable to any Array, it is not an l-value so it cannot appear on the left-hand side of an assignment.

Data-parallel expressions can involve typical mathematical functions and output operations. For example, sin(A) yields an Array with values equal to the sine of each of Array A's values. dot(A,B) has values equaling the dot product of corresponding values in Arrays A and B. The contents of an entire Array can be easily printed to standard output. For example, the program

```
Array<1, double, Brick> A(Interval<1>(0,2));
Array<1, double, Brick> B(Interval<1>(1,3));
A = 1.0;
B = 2.0;
std::cout << A-B << std::endl;
```

yields (000:002:001) = 1 -1 -1. The initial (000:002:001) indicates the Array's domain ranges from 0 to 2 with a stride of 1. The three values in A-B follow.

The following four tables list the data-parallel operators that operate on Arrays. Table 7-1 lists standard C++ operators that can be applied to Arrays and also scalar values if appropriate. Each unary operator takes an Array parameter and returns an Array. The types of the two Arrays need not be the same. For example, ! can take an Array<bool>, Array<int>, Array<long>, or any other value type to which ! can be applied. The result is an Array<bool>. Each binary operator also returns an Array. When specifying two Arrays or an Array and a scalar value, a full set of operators is supported. When specifying an Array and a Tensor, TinyMatrix, or Vector, a more limited set of operators is supported. For example, == can take two Arrays, an Array and a scalar value, or a scalar value and an Array. If given two Arrays, corresponding values are used. If an argument is a scalar value, its same value is the used with each Array value. The + supports the same set of parameters but also supports adding an Array and a Tensor, an Array and a TinyMatrix, an Array and a Vector, a Tensor and an Array, a TinyMatrix and an Array, and a Vector and an Array. For these cases, the Array must have a value type that can be added to the other argument. For example, a Vector can be added to an Array of Vectors.

Table 7-1. Operators Permissible for Data-Parallel Expressions

	supported operators
unary operators	+, -, ~, !
binary operators with at least one Array and at most one scalar value	+, -, *, /, %, &, , ^, <, <=, >=, >, ==, !=, &&, , <<, >>
binary operators with at least one Array and at most one Tensor, TinyMatrix, or Vector	+, -, *, /, %, &, , ^, ==, !=

Mathematical functions that can be used in data-parallel expressions appear in Table 7-2. For example, applying cos to an Array of values with type T yields an Array with the same type. The functions are split into five sections:

- trigonometric and hyperbolic functions,
- functions computing absolute values, rounding functions, and modulus functions,
- functions computing powers, exponentiation, and logarithms,
- functions involving complex numbers, and
- functions for operating on matrices and tensors.

Several data-parallel functions require inclusion of header files declaring their underlying element-wise function. These header files are listed at the beginning of each section. For the data-parallel operator to be applicable, it must operate on the Array's type. For example, `cos` can be applied on Arrays of `int`, `double`, and even `bool`, but applying on Arrays of pointers is not supported because `cos` cannot be called with a pointer argument.

Two functions deserve special explanation. The `PETE_identity` function applies the identity operation to the array. That is, the returned array has values equaling the argument's values. `pow2`, `pow3`, and `pow4` provide fast ways to compute squares, cubes, and fourth powers of their arguments.

Table 7-2. Mathematical Functions Permissible for Data-Parallel Expressions

function	effect
Trigonometric and Hyperbolic Functions	<code>#include <math.h></code>
<code>Array<T> cos (const Array<T>& A)</code>	Returns the cosines of the Array's values.
<code>Array<T> sin (const Array<T>& A)</code>	Returns the sines of the Array's values.
<code>Array<T> tan (const Array<T>& A)</code>	Returns the tangents of the Array's values.
<code>Array<T> acos (const Array<T1>& A)</code>	Returns the arc cosines of the Array's values.
<code>Array<T> asin (const Array<T1>& A)</code>	Returns the arc sines of the Array's values.
<code>Array<T> atan (const Array<T1>& A)</code>	Returns the arc tangents of the Array's values.
<code>Array<T> atan2 (const Array<T1>& A, const Array<T2>& B)</code>	Computes the arc tangents of the values from the division of elements in B by the elements in A. The resulting values are the signed angles in the range $-\pi$ to π , inclusive.

function

```
Array<T> atan2 (const
Array<T1>& A, const T2&
r)
```

```
Array<T> atan2 (const T1&
l, const Array<T2>& B)
```

```
Array<T> cosh (const
Array<T>& A)
```

```
Array<T> sinh (const
Array<T>& A)
```

```
Array<T> tanh (const
Array<T>& A)
```

Absolute Value, Rounding, and Modulus
Functions

```
Array<T> fabs (const
Array<T1>& A)
```

```
Array<T> ceil (const
Array<T1>& A)
```

```
Array<T> floor (const
Array<T1>& A)
```

```
Array<T> fmod (const
Array<T1>& A, const
Array<T2>& B)
```

```
Array<T> fmod (const
Array<T1>& A, const T2&
r)
```

effect

Computes the arc tangents of the values from the division of r by the elements in A . The resulting values are the signed angles in the range $-\pi$ to π , inclusive.

Computes the arc tangents of the values from the division of elements in B by l . The resulting values are the signed angles in the range $-\pi$ to π , inclusive.

Returns the hyperbolic cosines of the Array's values.

Returns the hyperbolic sines of the Array's values.

Returns the hyperbolic tangents of the Array's values.

```
#include <math.h>
```

Returns the absolute values of the floating point numbers in the Array.

For each of the Array's values, return the integer larger than or equal to it (as a floating point number).

For each of the Array's values, return the integer smaller than or equal to it (as a floating point number).

Computes the floating-point modulus (remainder) of A 's values with the corresponding value in B . The results have the same signs as A and absolute values less than the absolute values of B .

Computes the floating-point modulus (remainder) of A 's values with r . The results have the same signs as A and absolute values less than the absolute value of r .

function

```
Array<T> fmod (const T1&
l, const Array<T2>& B)
```

Powers, Exponentiation, and Logarithmic
Functions

```
Array<T> PETE_identity
(const Array<T>& A)
```

```
Array<T> sqrt (const
Array<T>& A)
```

```
Array<T> pow (const
Array<T1>& A, const
Array<T2>& B)
```

```
Array<T> pow (const
Array<T1>& A, const T2&
r)
```

```
Array<T> pow (const T1&
l, const Array<T2>& B)
```

```
Array<T> pow2 (const
Array<T>& A)
```

```
Array<T> pow3 (const
Array<T>& A)
```

```
Array<T> pow4 (const
Array<T>& A)
```

```
Array<T> ldexp (const
Array<T1>& A, const
Array<int>& B)
```

```
Array<T> ldexp (const
Array<T1>& A, int r)
```

```
Array<T> ldexp (const T1&
l, const Array<int>& B)
```

```
Array<T> exp (const
Array<T>& A)
```

```
Array<T> log (const
Array<T>& A)
```

effect

Computes the floating-point modulus (remainder) of *l* with the values in *B*. The results have the same signs as *l* and absolute values less than the absolute values of *B*.

```
#include <math.h>
```

Returns the Array. That is, it applies the identity operation.

Returns the square roots of the Array's values.

Raises A's values by the corresponding power in B.

Raises A's values by the power *r*.

Raises *l* by the powers in B.

Returns the squares of A's values.

Returns the cubes of A's values.

Returns the fourth powers of A's values.

Multiplies A's values by two raised to the corresponding value in B.

Multiplies A's values by two raised to the *r*th power.

Multiplies *l* by two raised to the values in B.

Returns the exponentiations of the Array's values.

Returns the natural logarithms of the Array's values.

function	effect
<code>Array<T> log10 (const Array<T>& A)</code>	Returns the base-10 logarithms of the Array's values.
Functions Involving Complex Numbers	<code>#include <complex></code>
<code>Array<T> real (const Array<complex<T>>& A)</code>	Returns the real parts of A's complex numbers.
<code>Array<T> imag (const Array<complex<T>>& A)</code>	Returns the imaginary parts of A's complex numbers.
<code>Array<T> abs (const Array<complex<T>>& A)</code>	Returns the absolute values (magnitudes) of A's complex numbers.
<code>Array<T> abs (const Array<T>& A)</code>	Returns the absolute values of A's values.
<code>Array<T> arg (const Array<complex<T>>& A)</code>	Returns the angle representations (in radians) of the polar representations of A's complex numbers.
<code>Array<T> norm (const Array<complex<T>>& A)</code>	Returns the squared absolute values of A's complex numbers.
<code>Array<complex<T>> conj (const Array<complex<T>>& A)</code>	Returns the complex conjugates of A's complex numbers.
<code>Array<complex<T>> polar (const Array<T1>& A, const Array<T2>& B)</code>	Returns the complex numbers created from polar coordinates (magnitudes and phase angles) in corresponding Arrays.
<code>Array<complex<T>> polar (const T1& l, const Array<T2>& A)</code>	Returns the complex numbers created from polar coordinates with magnitude l and phase angles in the Array.
<code>Array<complex<T>> polar (const Array<T1>& A, const T2& r)</code>	Returns the complex numbers created from polar coordinates with magnitudes in the Array and phase angle r.
Functions Involving Matrices and Tensors	<code>#include "Pooma/Tiny.h"</code>
<code>T trace (const Array<T>& A)</code>	Returns the sum of the A's diagonal entries, viewed as a matrix.
<code>T det (const Array<T>& A)</code>	Returns the determinant of A, viewed as a matrix.
<code>Array<T> transpose (const Array<T>& A)</code>	Returns the transpose of A, viewed as a matrix.
<code>Array<T> symmetrize (const Array<T>& A)</code>	Returns the tensors of A with the requested output symmetry.

function

```

Array<T> dot (const
Array<T1>& A, const
Array<T2>& B)
Array<T> dot (const
Array<T1>& A, const T2&
r)
Array<T> dot (const T1&
l, const Array<T2>& A)

Array<Tensor<T>>
outerProduct (const
Array<T1>& A, const
Array<T2>& B)

Array<Tensor<T>>
outerProduct (const T1&
l, const Array<T2>& A)

Array<Tensor<T>>
outerProduct (const
Array<T1>& A, const T2&
r)

TinyMatrix<T>
outerProductAsTinyMatrix
(const Array<T1>& A,
const Array<T2>& B)

TinyMatrix<T>
outerProductAsTinyMatrix
(const T1& l, const
Array<T2>& A)

```

effect

Returns the dot products of values in the two Arrays. Value type T equals the type of the dot operating on T1 and T2.

Returns the dot products of values in the Array with r. Value type T equals the type of the dot operating on T1 and T2.

Returns the dot products of l with values in the Array. Value type T equals the type of the dot operating on T1 and T2.

Returns tensors created by computing the outer product of corresponding vectors in the two Arrays. Value type T equals the type of the product of T1 and T2. The vectors must have the same length.

Returns tensors created by computing the outer product of l with the vectors in the Array. Value type T equals the type of the product of T1 and T2. The vectors must have the same length.

Returns tensors created by computing the outer product of vectors in the Array with r. Value type T equals the type of the product of T1 and T2. The vectors must have the same length.

Returns matrices created by computing the outer product of corresponding vectors in the two Arrays. Value type T equals the type of the product of T1 and T2. The vectors must have the same length.

Returns matrices created by computing the outer product of l with the vectors in the Array. Value type T equals the type of the product of T1 and T2. The vectors must have the same length.

function

```
TinyMatrix<T>
outerProductAsTinyMatrix
(const Array<T1>& A,
const T2& r)
```

effect

Returns matrices created by computing the outer product of the vectors in the Array with *r*. Value type *T* equals the type of the product of *T1* and *T2*. The vectors must have the same length.

Type restrictions from how the underlying functions operate on individual elements may restrict permissible choices for the template type parameters.

Comparison functions appear in Table 7-3. `max` and `min` functions supplement named comparison functions. For example, `LT` and `LE` compute the same thing as the `<` and `<=` operators.

Table 7-3. Comparison Functions Permissible for Data-Parallel Expressions

function

```
Array<T> max (const
Array<T1>& A, const
Array<T2>& B)
Array<T> max (const T1&
l, const Array<T2>& A)
Array<T> max (const
Array<T1>& A, const T2&
r)
Array<T> min (const
Array<T1>& A, const
Array<T2>& B)
Array<T> min (const T1&
l, const Array<T2>& A)
Array<T> min (const
Array<T1>& A, const T2&
r)
Array<bool> LT (const
Array<T1>& A, const
Array<T2>& B)
```

effect

Returns the maximum of corresponding Array values.

Returns the maximums of *l* with the Array's values.

Returns the maximums of the Array's values with *r*.

Returns the minimum of corresponding Array values.

Returns the minimums of *l* with the Array's values.

Returns the minimums of the Array's values with *r*.

Returns booleans from using the less-than operator `<` to compare corresponding Array values in *A* and *B*.

function

```
Array<bool> LT (const T1&
r, const Array<T2>& A)
```

```
Array<bool> LT (const
Array<T1>& A, const T2&
r)
```

```
Array<bool> LE (const
Array<T1>& A, const
Array<T2>& B)
```

```
Array<bool> LE (const T1&
l, const Array<T2>& A)
```

```
Array<bool> LE (const
Array<T1>& A, const T2&
r)
```

```
Array<bool> GE (const
Array<T1>& A, const
Array<T2>& B)
```

```
Array<bool> GE (const T1&
l, const Array<T2>& A)
```

```
Array<bool> GE (const
Array<T1>& A, const T2&
r)
```

```
Array<bool> GT (const
Array<T1>& A, const
Array<T2>& B)
```

```
Array<bool> GT (const T1&
l, const Array<T2>& A)
```

```
Array<bool> GT (const
Array<T1>& A, const T2&
r)
```

```
Array<bool> EQ (const
Array<T1>& A, const
Array<T2>& B)
```

effect

Returns booleans from using the less-than operator < to compare l with the Array's values.

Returns booleans from using the less-than operator < to compare the Array's values with r.

Returns booleans from using the less-than-or-equal operator <= to compare Array values in A and B.

Returns booleans from using the less-than-or-equal operator <= to compare l with the Array's values.

Returns booleans from using the less-than-or-equal operator <= to compare the Array's values with r.

Returns booleans from using the greater-than-or-equal operator >= to compare Array values in A and B.

Returns booleans from using the greater-than-or-equal operator >= to compare l with the Array's values.

Returns booleans from using the greater-than-or-equal operator >= to compare the Array's values with r.

Returns booleans from using the greater-than operator > to compare Array values in A and B.

Returns booleans from using the greater-than operator > to compare l with the Array's values.

Returns booleans from using the greater-than operator > to compare the Array's values with r.

Returns booleans from determining whether corresponding Array values in A and B are equal.

function	effect
<code>Array<bool> EQ (const T1& l, const Array<T2>& A)</code>	Returns booleans from determining whether <code>l</code> equals the Array's values.
<code>Array<bool> EQ (const Array<T1>& A, const T2& r)</code>	Returns booleans from determining whether the Array's values equal <code>r</code> .
<code>Array<bool> NE (const Array<T1>& A, const Array<T2>& B)</code>	Returns booleans from determining whether corresponding Array values in <code>A</code> and <code>B</code> are not equal.
<code>Array<bool> NE (const T1& l, const Array<T2>& A)</code>	Returns booleans from determining whether <code>l</code> does not equal the Array's values.
<code>Array<bool> NE (const Array<T1>& A, const T2& r)</code>	Returns booleans from determining whether the Array's values are not equal to <code>r</code> .

The table of miscellaneous functions (Table 7-4) contains two functions. `peteCast` casts all values in an Array to the type specified by its first parameter. The `where` function generalizes the trinary `? :` operator. Using its first Array argument as boolean values, it returns an Array of just two values: `t` and `f`.

Table 7-4. Miscellaneous Functions Permissible for Data-Parallel Expressions

function	effect
<code>Array<T> peteCast (const T1&, const Array<T>& A)</code>	Returns the casting of the Array's values to type <code>T1</code> .
<code>Array<T> where (const Array<T1>& A, const T2& t, const T3& f)</code>	Generalizes the <code>? :</code> operator, returning an Array of <code>t</code> and <code>f</code> values depending on whether <code>A</code> 's values are true or false, respectively.

Throughout this chapter, we illustrate data-parallel expressions and statements operating on all of a container's values. Frequently, operating on a subset is useful. In POOMA, a subset of a container's values is called a view. Combining views and data-parallel expressions will enable us to more succinctly and more easily write the `Doof2d` diffusion program. Views are discussed in the next chapter.

7.3. Implementation of Data-Parallel Statements

Data-parallel statements involving containers occur frequently in the inner loops of scientific programs so their efficient execution is important. A naïve implementation for these statements may create and destroy containers holding intermediate values, slowing execution considerably. In 1995, Todd Veldhuizen and David Vandevoorde each developed an expression-template technique to transform arithmetic expressions involving array-like containers into efficient loops without using temporaries. Despite its perceived complexity, POOMA incorporated the technology. The framework called PETE, the Portable Expression Template Engine framework, is also available separately from POOMA at <http://www.acl.lanl.gov/pete/>.

In this chapter, we first describe how a naïve implementation may slow execution. Then, we describe PETE's faster implementation. PETE converts a data-parallel statement into a parse tree, rather than immediately evaluating it. The parse tree has two representations. Its run-time representation holds run-time values. Its compile-time representation records the types of the tree's values. After a parse tree for the entire statement is constructed, it is evaluated. Since it is a data-parallel statement, this evaluation involves at least one loop. At run time, for each loop iteration, the value of one container value is computed and assigned. At compile time, when the code for the loop iteration is produced, the parse tree's types are traversed and code is produced without the need for any intermediate values. We present the implementation in Section 7.3.2, but first we explain the difficulties caused by the naïve implementation.

7.3.1. Naïve Implementation

A conventional implementation to evaluate data-parallel expressions might overload arithmetic operator functions. Consider this program fragment:

```
Interval<1> I(0,3);
Array<1, double, Brick> A(I), B(I);
A = 1.0;
B = 2.0;
A += -A + 2*B;
std::cout << A << std::endl;
```

Our goal is to transform the data-parallel statement `A += -A + 2*B` into a single loop, preferably without using intermediary containers. To simplify notation, let `Ar` abbreviate the type `Array<1, double, Brick>`.

Using overloaded arithmetic operators would require using intermediate containers to evaluate the statement. For example, the sum's left operand $-A$ would be computed by the overloaded unary operator `Ar operator-(const Ar&)`, which would produce an intermediate Array. `Ar operator*(double, const Ar&)` would produce another intermediate Array holding $2*B$. Yet another intermediate container would hold their sum, all before performing the assignment. Thus, three intermediate containers would be created and destroyed. Below, we show these are unnecessary.

7.3.2. Portable Expression Template Engine

POOMA uses PETE, the Portable Expression Template Engine framework, to evaluate data-parallel statements using efficient loops without intermediate values. PETE uses expression-template technology. Instead of evaluating a data-parallel statement's subexpressions at solely at run time, it evaluates the code at both run time and at compile time. At compile time, it builds a parse tree of the required computations. The parse tree's type records the types of each of its subtrees. Then, the parse tree is evaluated at compile time using an evaluator determined by the left-hand side's type. This container type determines how to loop through its domain. Each loop iteration of the resulting run time code, the corresponding value of the right-hand side is evaluated. No intermediate loops or temporary values are needed.

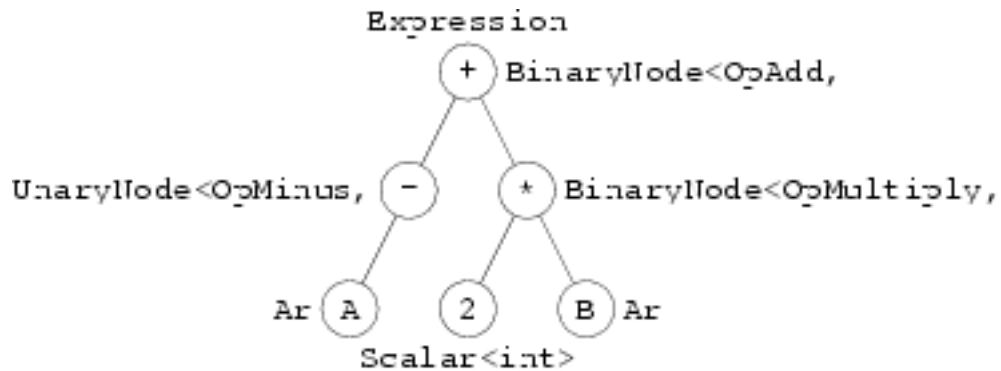
Before explaining the implementation, let us illustrate using our example statement $A += -A + 2*B$. Evaluating the right-hand side creates a parse tree similar to the one in Figure 7-2. For example, the overloaded unary minus operator yields a tree node representing $-A$, having a unary-minus function object, and having type

```
Expression<UnaryNode<OpMinus,Ar>>
```

The binary nodes continue the construction process yielding a parse tree object for the entire right-hand side and having type

```
Expression<
  BinaryNode<OpAdd,
    UnaryNode<OpMinus, Ar>,
    BinaryNode<OpMultiply<Scalar<int>,Ar>>>
```

Evaluating the left-hand side yields an object representing A .

Figure 7-2. Annotated Parse Tree for $-A + 2*B$ 

The parse tree for $-A + 2*B$ with type annotations. The complete type of a node equals the concatenation of the preorder traversal of annotated types.

Finally, the assignment operator `+=` calls the `evaluate` function corresponding to the left-hand side's type. At compile time, it produces the code for the computation. Since this templated function is specialized on the type of the left-hand side, it generates a loop iterating through the left-hand side's container. To produce the loop body, the `forEach` function produces code for the right-hand side expression at a specific position using a post-order parse-tree traversal. At a leaf, this evaluation queries the leaf's container for a specified value or extracts a scalar value. At an interior node, its children's results are combined using its function operator. One loop performs the entire assignment. It is important to note that the type of the entire right-hand side is known at compile time. Thus, all of these `evaluate`, `forEach`, and function operator function calls can be inlined at compile time to yield simple code without any temporary containers and hopefully as fast as hand-written loops!

To implement this scheme, we need POOMA (really PETE) code to both create the parse tree and to evaluate it. We describe parse tree creation first. Parse trees consist of leaves, `UnaryNodes`, `BinaryNodes`, and `TrinaryNodes`. Since `TrinaryNodes` are similar to `BinaryNodes`, we omit describing them. A `BinaryNode`'s three template parameters correspond to the three things it must store:

`Op`

the type of the node's operation. For example, the `OpAdd` type represents adding two operands together.

`Left`

the type of the left child.

Right

the type of the right child.

The node stores the left and right children's nodes.

BinaryNode does not need to store any representation of the node's operation. Instead the Op type is an empty structure defining a function object. For example, OpAdd's function object is declared as

```
template<class T1, class T2>
inline typename BinaryReturn<T1, T2, OpAdd>::Type_t
operator()(const T1 &a, const T2 &b) const
{
    return (a + b);
}
```

Since it has two template arguments, it can be applied to operands of any type. Because of C++ type conversions, the type of the result is determined using the BinaryReturn traits class. Consider adding an int and a double. BinaryReturn<int, double, OpAdd>::Type_t equals double. Inlining the function ensures all this syntax is eliminated, leaving behind just an addition.

UnaryNodes are similar but have only two template parameters and store only one child.

Parse tree leaves are created by the CreateLeaf class and its specializations. The default leaf is a scalar so it has the most general definition:

```
template<class T>
struct CreateLeaf
{
    typedef Scalar<T> Leaf_t;

    inline static
    Leaf_t make(const T &a)
    {
        return Scalar<T>(a);
    }
};
```

The `Scalar` class stores the scalar value. The `CreateLeaf`'s `Leaf_t` type indicates its type. The static `make` function is invoked by an overloaded operator function when creating its children.

The `CreateLeaf` class is specialized for Arrays:

```
template<int Dim, class T, class EngineTag>
struct CreateLeaf<Array<Dim, T, EngineTag> >
{
    typedef Array<Dim, T, EngineTag> Input_t;
    typedef Reference<Input_t> Leaf_t;
    typedef Leaf_t Return_t;
    inline static
    Return_t make(const Input_t &a)
    {
        return Leaf_t(a);
    }
};
```

The `Array` object is stored as a `Reference`, rather than directly as for scalars.

To simplify the next step of overloading arithmetic operators, a parse tree's topmost type is an `Expression`.

Now that we have defined the node classes, the C++ arithmetic operators must be overloaded to return the appropriate parse tree. For example, the unary minus operator `operator-` is overloaded to accept an `Array` argument. It should create a `UnaryNode` having an `Array` as its child, which will be a leaf:

```
template<int D1, class T1, class E1>
inline typename MakeReturn<UnaryNode<OpUnaryMinus,
    typename CreateLeaf<Array<D1, T1, E1>>::Leaf_t>::
    Expression_t
operator-(const Array<D1, T1, E1> & l)
{
    typedef UnaryNode<OpUnaryMinus,
        typename CreateLeaf<Array<D1, T1, E1> >::Leaf_t> Tree_t;
    return MakeReturn<Tree_t>::make(Tree_t(
        CreateLeaf<Array<D1, T1, E1> >::make(l)));
}
```

`Tree_t` specifies the node's unique type. Constructing the object first involves creating a leaf containing the `Array` reference through the call to

```
CreateLeaf<Array<D1,T1,E1>>::make
```

The call to `MakeReturn<Tree_t>::make` permits programmers to store trees in different formats. The POOMA implementation stores them as `Expressions`. The function's return type is similar to the `return` statement except it extracts the type from `Expression`'s internal `Expression_t` type.

Specializing all the operators for `Arrays` using such complicated functions is likely to be error-prone so PETE provides a way to automate their creation. Using its `MakeOperators` command with this input:

```
classes
-----
ARG    = "int D[n],class T[n],class E[n]"
CLASS  = "Array<D[n],T[n],E[n]>"
```

automatically generates code for all the needed operators. The “[n]” strings are used to number arguments for binary and ternary operators.

Assignment operators must also be specialized for `Array`. Inside the `Array` class definition, each such operator just invokes the `assign` function with a corresponding function object. For example, `operator+=` invokes `assign(*this, rhs, OpAddAssign())`. `rhs` is the parse tree object for the right-hand side. Calling this function invokes `evaluate`, which begins the evaluation.

Before we explain the evaluation, let us summarize the effect of the code so far described. If we are considering run time evaluation, parse trees for the left-hand and right-hand sides have been constructed. If we are considering compile time evaluation, the types of these parse trees are known. At compile time, the `evaluate` function described below will generate a loop iterating through the left-hand side container's domain. The loop's body will have code computing a container's value. At run time, this code will read values from containers, but the run-time parse tree object itself will not traversed!

We now explore the evaluation, concentrating on compile time, not run time. `evaluate` is an overloaded function specialized on the type of the left-hand side. In our example, the left-hand side is a one-dimensional `Array`, so `evaluate(const Ar& a, const Op& op, const RHS& rhs)` is inlined into a loop like

```
int end = a's domain[0].first() + a's domain[0].length();
for (int i = a's domain[0].first(); i < end; ++i)
    op(a(i), rhs.read(i));
```

`a` is the array, `op` is a function object representing the assignment operation, and `rhs` is the right-hand side's parse tree.

Evaluating `rhs.read(i)` inlines into a call to the `forEach` function. This function performs a *compile-time* post-order parse-tree traversal. Its general form is

```
forEach(const Expression& e, const LeafTag& f,
        const CombineTag& c).
```

That is, it traverses the nodes of the `Expression` object `e`. At leaves, it applies the operation specified by `LeafTag` `f`. At interior nodes, it combines the results using the `CombineTag` operator `c`. It inlines into a call to

```
ForEach<Expression, LeafTag, CombineTag>::apply(e, f, c)
```

The `apply` function continues the traversal through the tree. For our example, `LeafTag` equals `EvalLeaf<1>`, and `CombineTag` equals `OpCombine`. The former indicates that, when reaching a leaf, the leaf should be a one-dimensional container which should be evaluated at the position stored in the `EvalLeaf` object. The `OpCombine` class applies an interior node's `Op` to the results of its children.

`ForEach` structures are specialized for the various node types. For example, the specialization for `UnaryNode` is

```
template<class Op, class A, class FTag, class CTag>
struct ForEach<UnaryNode<Op, A>, FTag, CTag>
{
    typedef typename ForEach<A, FTag, CTag>::Type_t TypeA_t;
    typedef typename
        Combine1<TypeA_t, Op, CTag>::Type_t Type_t;
    inline static
    Type_t apply(const UnaryNode<Op, A>&expr, const FTag&f,
        const CTag& c)
    {
        return Combine1<TypeA_t, Op, CTag>::
```

```

        combine(ForEach<A, FTag, CTag>::
            apply(expr.child(), f, c), c);
    }
};

```

Since this structure is specialized for `UnaryNodes`, the first parameter of its `static apply` function is a `UnaryNode`. After recursively calling its child, it invokes the combination function indicated by the `Combine1` traits class. In our example, the `c` function object should be applied. Other combinators have different roles. For example, using the `NullCombine` tag indicates the child's result should not be combined but occurs just for side effects.

Leaves are treated as the default behavior so they are not specialized:

```

template<class Expr, class FTag, class CTag>
struct ForEach
{
    typedef typename
        LeafFunctor<Expr, FTag>::Type_t Type_t;
    inline static
        Type_t apply(const Expr&expr, const FTag&f, const CTag&)
        {
            return LeafFunctor<Expr, FTag>::apply(expr, f);
        }
};

```

Thus, `LeafFunctor`'s `apply` member is called. `Expr` represents the expression type, e.g., an `Array`, and `FTag` is the `LeafTag`, e.g., `EvalLeaf`. The `LeafFunctors` specialization for `Array` passes the index stored by the `EvalLeaf` object to the `Array`'s engine, which returns the corresponding value.

If one uses an aggressive optimizing compiler, code resulting from the `evaluate` function corresponds to this pseudocode:

```

int end = A.domain[0].first() + A.domain[0].length();
for (int i = A.domain[0].first(); i < end; ++i)
    A.engine(i) += -A.engine.read(i)+2*B.engine.read(i);

```

The loop iterates through `A`'s domain, using `Array`'s engines to obtain values and assigning values. Notice there is no use of the run-time parse tree so the optimizer can

eliminate the code to construct it. All the work to construct the parse tree by overloading operators is unimportant at run time, but it certainly helped the compiler produce improved code.

PETE's expression template technology may be complicated, using parse trees and their types, but the produced code is not. Using the technology is also easy. All data-parallel statements are automatically converted. In the next chapter, we explore views of containers, permitting use of container subsets and making data-parallel expressions even more useful.

Chapter 8. Container Views

A *view of a container* is a container accessing a subset of C's domain and values. The subset can include all of the container's domain. A “view” is so named because it is a different way to access, or view, another container's values. Both the container and its view share the same underlying engine so changing values in one also changes them in the other.

A view is created by following a container's name by parentheses containing a domain. For example, consider this code extracted from Example 3-3 in Section 3.4.

```
Interval<1> N(0, n-1);
Interval<2> vertDomain(N, N);
Interval<1> I(1,n-2);
Interval<1> J(1,n-2);
Array<2, double, Brick> a(vertDomain);
Array<2, double, Brick> b(vertDomain);
a(I,J) = (1.0/9.0) *
    (b(I+1,J+1) + b(I+1,J ) + b(I+1,J-1) +
     b(I ,J+1) + b(I ,J ) + b(I ,J-1) +
     b(I-1,J+1) + b(I-1,J ) + b(I-1,J-1));
```

The last statement creates ten views. For example, `a (I , J)` creates a view of `a` using the smaller domain specified by `I` and `J`. This omits the outermost rows of columns of `a`. The views of `b` illustrate the use of views in data-parallel statements. `b (I-1 , J-1)` has a subset shifted up one row and left one column compared with `b (I , J)`.

Appendix A. Obtaining and Installing POOMA

In Section 3.1, we described how to install POOMA. In the following section, we describe how to install POOMA to support distributed computation.

A.1. Supporting Distributed Computation

To use multiple processors with POOMA requires installing the Cheetah messaging library and an underlying messaging library such as the Message Passing Interface (MPI) Communications Library or the MM Shared Memory Library. In the following section, we first describe how to install MM. Read it only if using MM, not MPI. Then we describe how to install Cheetah and configure POOMA to use it.

A.1.1. Obtaining and Installing the MM Shared Memory Library

Cheetah, and thus POOMA, can use Ralf Engelschall's MM Shared Memory Library to pass messages between processors. For example, the author uses this library on a two-processor computer running Linux. The library, available at <http://www.engelschall.com/sw/mm/>, is available at no cost and has been successfully tested on a variety of Unix-like platforms.

We describe how to download and install the MM library.

1. Download the library from the POOMA Download page (<http://pooma.codesourcery.com/pooma/download>) available off the POOMA home page (<http://www.codesourcery.com/pooma/pooma/>).
2. Extract the source code using `tar xzvf mm-1.1.3.tar.gz`. Change directories into the resulting source code directory `mm-1.1.3`.
3. Prepare to compile the source code by configuring it using the `configure` command. To change the default installation directory `/usr/local`, specify `--prefix=directory` option. The other configuration options can be listed by specifying the `--help` option. Since the author prefers to keep all POOMA-related code in his `poomasubdirectory`, he uses

```
./configure --prefix=${HOME}/pooma/mm-1.1.3
```

4. Create the library by issuing the `make` command. This compiles the source code using a C compiler. To use a different compiler than the MM configuration chooses, set the `CC` environment variable to the desired compiler before configuring.
5. Optionally test the library by issuing the `make test` command. If successful, the penultimate line should be `OK - ALL TESTS SUCCESSFULLY PASSED`.
6. Install the MM Library by issuing the `make install` command. This copies the library files to the installation directory. The `mm-1.1.3` directory containing the source code may now be removed.

A.1.2. Obtaining and Installing the Cheetah Messaging Library

The Cheetah Library decouples communication from synchronization. Using asynchronous messaging rather than synchronous messaging permits a message sender to operate without the cooperation of the message recipient. Thus, implementing message sending is simpler and processing is more efficiently overlapped with it. Remote method invocation is also supported. The library was developed at the Los Alamos National Laboratory's Advanced Computing Laboratory.

Cheetah's messaging is implemented using an underlying messaging library such as the Message Passing Interface (MPI) Communications Library or the MM Shared Memory Library. MPI works on a wide variety of platforms and has achieved widespread usage. MM works under Unix-like operating systems on any computer with shared memory. Both libraries are available at no cost. The instructions below work for whichever library you choose.

We describe how to download and install Cheetah.

1. Download the library from the POOMA Download page (<http://pooma.codesourcery.com/pooma/download>) available off the POOMA home page (<http://www.codesourcery.com/pooma/pooma/>).
2. Extract the source code using `tar xzvf cheetah-1.0.tgz`. Change directories into the resulting source code directory `cheetah-1.0`.
3. Edit a configuration file corresponding to your operating system and compiler. These `.conf` files are located in the `config` directory. For example, to use `g++`

with the Linux operating system, use `config/LINUXGCC.conf`.

The configuration file usually does not need modification. However, if you are using MM, ensure `shmem_default_dir` specifies its location. For example, the author modified the value to `" /home/oldham/pooma/mm-1.1.3 "`.

4. Prepare to compile the source code by configuring it using the `configure` command. Specify the configuration file using the `--arch` option. Its argument should be the configuration file's name, omitting its `.conf` suffix. For example, `--arch LINUXGCC`. Some other options include

`--help`

lists all the available options

`--shmem --nompi`

indicates use of MM, not MPI

`--mpi --noshmem`

indicates use of MPI, not MM

`--opt`

causes the compiler to produce optimized source code

`--noex`

prevents use of C++ exceptions

`--static`

creates a static library, not a shared library

`--shared`

creates a shared library, not a static library. This is the default.

`--prefix directory`

specifies the installation directory where the library will be copied rather than the default.

For example, the author uses

```
./configure --arch LINUXGCC --shmem --nompi
--noex --static --prefix ${HOME}/pooma/cheetah-1.0
--opt
```

The `--arch LINUXGCC` indicates use of `g++` under a Linux operating system. The MM library is used, but C++ exceptions are not. The latter choice matches POOMA's default choice. A static library, not a shared library, is created. This is also POOMA's default choice. The library will be installed in the `${HOME}/pooma/cheetah-1.0`. Finally, the library code will be optimized, hopefully running faster than unoptimized code.

5. Follow the directions printed by `configure`: Change directories to the `lib` sub-directory named by the `--arch` argument and then type `make` to compile the source code and create the library.
6. Optionally ensure the library works correctly by issuing the `make tests` command.
7. Install the library by issuing the `make install` command. This copies the library files to the installation directory. The `cheetah-1.0` directory containing the source code may now be removed.

A.1.3. Configuring POOMA When Using Cheetah

To use POOMA with Cheetah, one must tell POOMA the location of the Cheetah library using the `--messaging` configuration option. To do this,

1. Set the Cheetah directory environment variable `CHEETAHDIR` to the directory containing the installed Cheetah library. For example,

```
declare -x CHEETAHDIR=${HOME}/pooma/cheetah-1.0
```

specifies the installation directory used in the previous section. If using the `csh` shell, use `setenv CHEETAHDIR ${HOME}/pooma/cheetah-1.0`.

2. When configuring POOMA, specify the `--messaging` option. For example, `./configure --arch LINUXgcc --opt --messaging` configures for Linux, `g++`, and an optimized library using Cheetah.

Glossary

A

architecture

particular hardware (processor) interface. Examples architectures include “linux”, “sgin32”, “sgi64”, and “sun”.

Array

a POOMA container generalizing C arrays and mapping indices to values. Constant-time access to values is supported, ignoring the time to compute the values if applicable. Arrays are first-class objects. DynamicArrays and Fields generalize Array.

See Also: container, DynamicArray, Field.

B

Brick Engine

an Engine explicitly storing each of its values. Its space requirements are at least the size of the Engine’s domain.

See Also: engine.

C

cell

a domain element of a Field. Both Array and Field domain elements are denoted by indices, but a cell exists in space. For example, it might be a rectangle or rectangular parallelepiped.

See Also: cell size, Field, mesh.

cell size

specifies a `Field` cell's dimensions e.g., its width, height, and depth, in \Re^d . This is frequently used to specify a mesh.

See Also: cell, mesh, corner position.

communication library

software library passing information among *contexts*, usually using messages.

See Also: distributed computing environment.

compilation time

See: compile time

compile time

in the process from writing a program to executing it, the time when the program is compiled by a compiler. This is also called *compilation time*.

See Also: programming time, run time.

computing environment

computer. More precisely, a computer with its arrangement of processors and associated memory, possibly shared among processors.

See Also: sequential computing environment, distributed computing environment.

conformable containers

containers with conformable domains.

See Also: conformable domains, data parallel.

conformable domains

domains with the “same shape” so that corresponding dimensions have the same number of elements. Scalars, deemed conformable with any domain, get “expanded” to the domain's shape. Assignment can operate on containers with conformable domains.

See Also: conformable containers, data parallel.

container

an object that stores other objects, controlling their allocation, deallocation, and access. Similar to C++ containers, the most important POOMA containers are `Arrays` and `Fields`.

See Also: `Array`, `DynamicArray`, `Field`, `Tensor`, `TinyMatrix`, `Vector`.

container value

object stored within a container and usually addressable via an index. Synonyms include “element” and “value”.

context

a collection of shared memory and processors that can execute a program or a portion of a program. It can have one or more processors, but all these processors must access the same shared memory. Usually the computer and its operating system, not the programmer, determine the available contexts.

See Also: distributed computing environment, layout.

context mapper

indicates how a container’s patches are mapped to processors and shared memory. Two common choices are distribution among the various processors and replication.

See Also: context, patch.

corner position

specifies the \mathfrak{R}^d point corresponding to a `Field` domain’s lower, left corner.

See Also: mesh, cell size.

D**data parallel**

describes an expression involving a (non-singleton) subset of a container’s values.

For example, `sin(C)` is an expression indicating that the `sin` is applied to each value in container `C`.

See Also: element wise, relation, stencil.

distributed computing environment

computing environment with one or more processors each having associated memory, possibly shared. In some contexts, it refers to strictly multiprocessor computation.

See Also: computing environment, sequential computing environment.

domain

a set of points on which a container can define values. For example, a set of discrete integral n -tuples in n -dimensional space frequently serve as container domains.

See Also: container, interval, stride, range.

domain triplet notation

notation `[begin:end:stride]` representing the mathematical set $\{\text{begin}, \text{begin} + \text{stride}, \text{begin} + 2\text{stride}, \dots, \text{end}\}$. `end` is in the set only if it equals `begin` plus some integral multiple of `stride`. This notation can abbreviate many domains. It is extended to multiple dimensions by separating the dimensions' sets with commas: `[begin0:end0:stride0,begin1:end1:stride1]`.

See Also: domain.

DynamicArray

a POOMA container generalizing one-dimensional `Arrays` by supporting domain resizing at run-time. It maps indices to values in constant time, ignoring the time to compute the values if applicable. `DynamicArrays` are first-class objects.

See Also: container, `Array`, `Field`.

E

element

See: container value

element wise

describes accesses to individual values within a container. For example, `C (-4 , 3)` represents one particular value in the container `C`.

See Also: data parallel, relation, stencil.

engine

stores or computes a container's values. These can be specialized, e.g., to minimize storage when a domain has few distinct values. Separating a container and its storage also permits views of a container.

See Also: Brick Engine, container, view of a container.

enumeration

C++ integral type with named constants. These are frequently used in template programming because they can be used as template arguments.

execution time

See: run time

external guard layer

guard layer surrounding a container's domain used to ease computation along the domain's edges by permitting the same computations as for more internal computations. It is an optimization, not required for program correctness.

See Also: guard layer, internal guard layer, patch.

F**Field**

a POOMA container representing an `Array` with spatial extent. It also supports multiple values and multiple materials having the same index. It maps indices to values in constant time, ignoring the time to compute the values if applicable. It also supports geometric computations such as the distance between two cells and normals to a cell. `Fields` are first-class objects.

See Also: container, cell, mesh, `Array`, `DynamicArray`.

first-class type

a type of object with all the capabilities of the built-in type having the most capabilities. For example, `char` and `int` are first-class types in C++ because they may be declared anywhere, stored in automatic variables, accessed anywhere, copied, and passed by both value and reference. `POOMA Array` and `Field` are first-class types.

function object

object that can behave as a function. The object can store values that the function uses. If its function is called `operator ()`, the object can be invoked as a function.

function template

a definition of an unbounded set of related functions, all having the same name but whose types can depend on template parameters. They are particularly useful when overloading *operator functions* to accept parameters that themselves depend on templates.

G**guard layer**

domain surrounding each patch of a container's domain. It contains read-only values. External guard layers ease programming, while internal guard layers permit each patch's computation to occur without copying values from adjacent patches. They are optimizations, not required for program correctness.

See Also: external guard layer, internal guard layer, partition, patch, domain.

I**index**

a position in a domain usually denoted by an ordered tuple. More than one index are called indices.

See Also: domain.

instantiation

See: template instantiation

indices

More than one index.

See Also: index.

internal guard layer

guard layer containing copies of adjacent patches' values. These copies can permit an individual patch's computation to occur without asking adjacent patches for values. This can speed computation but are not required for program correctness.

See Also: guard layer, external guard layer, patch.

interval

a set of integral points between two endpoints. This domain is frequently represented using mathematical interval notation $[a,b]$ even though it contains only the integral points, e.g., $a, a+1, a+2, \dots, b$. It is also generalized to an n -dimensional interval as the direct product of one-dimensional intervals. Many containers' domains consist of these sets of ordered tuples.

See Also: domain, stride, range.

L**layout**

a map from an index to processor(s) and memory used to compute the container's associated value. For a uniprocessor implementation, a container's layout always consists of its domain, the processor, and its memory. For a multiprocessor imple-

mentation, the layout maps portions of the domain to (possibly different) processors and memory.

See Also: container, domain.

M

matrix

See: `TinyMatrix`

mesh

a `Field`'s map from indices to geometric values such as cell size, edge length, and cell normals. In other words, it specifies a `Field`'s "spatial extent".

See Also: `Field`, cell, cell size, corner position, layout.

O

operator function

function defining a function invoked using a C++ operator. For example, the `operator+` function defines the result of using the `+`.

P

partition

a specification how to divide a container's domain into patches for distributed computation. It can be independent of the domain's size. For example, it divide each domain into halves, yielding a total of eight patches in three dimensions. See Figure 3-4 for an illustration.

See Also: guard layer, patch, domain.

patch

subset of a container's domain with values computed by a particular context. A partition splits a domain into patches. It may be surrounded by external and internal guard layers.

See Also: partition, guard layer, domain.

point

a location in multidimensional space \mathfrak{R}^d . In contrast, indices specify positions in container domains.

See Also: Field, mesh, index.

programming time

in the process from writing a program to executing it, the time when the program is being written by a programmer.

See Also: compile time, run time.

R**range**

a set of integral points between two endpoints and separated by a stride. This domain, frequently represented by domain triplets [b:e:s], can also be represented mathematically as an integral interval [b,e] with stride s, i.e., {a, a+s, a+2s, ..., b}. It is generalized to an n-dimensional range as the direct product of one-dimensional ranges.

See Also: stride, interval, domain.

reference semantics

a copy of an object \circ refers to the object \circ such that changing either one also changes the other. This is the opposite of value semantics.

relation

dependence between a dependent container and one or more independent contain-

ers and an associated function. If a dependent container's values are needed and one or more of the independent containers' values have changed, the dependent container's values are computed using the function and the independent containers' values. Relations implement "lazy evaluation".

See Also: data parallel, element wise, stencil.

run time

in the process from writing a program to executing it, the time when the program is executed. This is also called *execution time*.

See Also: compile time, programming time.

S

sequential computing environment

a computing environment with one processor and associated memory. Only one processor executes a program even if the computer itself has multiple processors.

See Also: computing environment, distributed computing environment.

stencil

set of values neighboring a container index and a function using those values to compute it. For example, the stencil in a two-dimensional Conway game of life consists of an index's eight neighbors and a function that sets its value to "live" if it is already live and it has two neighbors or it has exactly three live neighbors.

See Also: data parallel, element wise, relation.

stride

spacing between regularly-spaced points in a domain. For example, the set of points $a, a+2, a+4, \dots, b-2, b$ is specified by $[a, b]$ with stride 2. It is a domain.

See Also: range, interval, domain.

suite name

an arbitrary string denoting a particular toolkit configuration. For example, the string "SUNKCC-debug" might indicate a configuration for the SunTM Solaris op-

erating system and the KCC C++ compiler with debugging support. By default, the suite name it is equal to the configuration's architecture name.

T

template

class or function definition having template parameters. These parameters' values are used at compile time, not run time, so they may include types and other compile-time values.

See Also: template instantiation, template specialization.

template instantiation

applying a template class to template parameter arguments to create a type. For example, `foo<double, 3>` instantiates template `<typename T, int n> class foo` with the type `double` and the constant integer 3. Template instantiation is analogous to applying a function to function arguments.

See Also: template.

template specialization

class or function definition for a particular (special) subset of template arguments.

See Also: template.

Tensor

a POOMA container implementing multidimensional mathematical tensors as first-class objects.

See Also: `TinyMatrix`, `Vector`.

TinyMatrix

a POOMA container implementing two-dimensional mathematical matrices as first-class objects.

See Also: `Tensor`, `Vector`.

trait

a characteristic of a type.

See Also: traits class.

traits class

a class containing one or more traits all describing a particular type's characteristics.

See Also: trait.

Turing complete

describes a language that can compute anything that can be computed. That is, the language for computation is as powerful as it can be. Most wide-spread programming languages are Turing-complete, including C++, C, and Fortran.

V**value**

See: container value

Vector

a POOMA container implementing multidimensional mathematical vectors, i.e., an ordered tuple of components, as first-class objects.

See Also: Tensor, TinyMatrix.

view of a container

a container derived from another. The view's domain is a subset of the latter's, but, where the domains intersect, accessing a value through the view is the same as accessing it through the original container. In Fortran 90, these are called array sections. Only Arrays, DynamicArrays, and Fields support views.

See Also: container.