

LA-UR-98-292

*Approved for public release;
distribution is unlimited*

Title: PADRE User's Manual

Author(s): Kei Davis
Dan Quinlan

Submitted to: Electronic Distribution

**Los Alamos
National Laboratory**

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

PADRE User's Manual

Kei Davis and Dan Quinlan

Los Alamos National Laboratory
MS B265

Los Alamos, NM 87545

505-667-1749 (office)

505-667-1226 (fax)

{kei,dquinlan}@lanl.gov

<http://www.c3.lanl.gov/~kei/kei.html>

PADRE Web Page: <http://www.c3.lanl.gov/~kei/?????.html>

LACC Number: LA-CC-??-??

LAUR Number: LA-UR-??-??

PADRE User Manual (postscript version)

January 12, 1998

January 12, 1998

Copyright

Copyright, 1997. The Regents of the University of California.

This software was produced under a U.S. Government contract (?????????) by the Los Alamos National Laboratory, which is operated by the University of California for the U.S. Department of Energy. The U.S. Government is licensed to use, reproduce, and distributed this software. Permission is granted to the public to copy and use this software without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor the University makes any warranty, express or implied, or assumes any liability or responsibility for the use of this software.

The following notice is specific to the use by the United States Government.

NOTICE: The Government is granted for itself and others acting on its behalf a paid-up, non-exclusive, irrevocable worldwide license in this data to reproduce, prepare derivative works, and perform publicly and display publicly. Beginning five (5) years after (date permission to assert copyright was obtained), subject to two possible five year renewals, the Government is granted for itself and others acting on its behalf a paid-up, non- exclusive, irrevocable worldwide license in this data to reproduce, prepare derivative works, distribute copies to the public, perform publicly and display publicly, and to permit others to do so. NEITHER THE UNITED STATES NOR THE UNITED STATES DEPARTMENT OF ENERGY, NOR ANY OF THEIR EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

Acknowledgments

This work has been funded by DOE 2000, via Jim McGraw and Rod Oldehoeft, and forms a part of the ACTS ToolKit within DOE 2000.

IMPLEMENTATION NOTE: NEED MORE INFO ON DOE 2000 CONTRACT.

While PADRE itself is solely our own work, it makes essential use of various distribution libraries provided by Alan Sussman and Gagan Agrawal (Multiblock PARTI), Robert C. Ferrell, Douglas B. Kothe, and John A. Turner (PGSLib), the LANL POOMA team (Domain Layout), Scott Baden (KeLP); much of the higher-level interface to Multiblock PARTI comes from Dan Quinlan's P++ library.

Contents

1	Introduction	1
1.1	PADRE	1
1.1.1	Basic Terminology	2
1.2	Structure	3
1.2.1	Application	3
1.2.2	PADRE	4
1.2.3	Distribution Libraries	4
1.2.4	Communication Libraries	4
2	Abstractions	7
2.1	User Abstractions	7
2.1.1	Domains	7
2.1.2	Collection	7
2.1.3	LocalDescriptor	8
2.2	PADRE Abstractions	8
2.2.1	Distribution	8
2.2.2	Representation	9
2.2.3	Descriptor	9
3	The PADRE API	11
3.1	User Base Classes which simplify the PADRE API	11
3.2	The PADRE API	11
3.2.1	Distribution Library Names	11
3.2.2	PADRE_CommonInterface	12
3.2.3	PADRE_Distribution	15
3.2.4	PADRE_Representation	17
3.2.5	PADRE_Descriptor	18
3.2.6	Copy assignment, copy constructor	18
3.2.7	Constructors and destructors	18
4	PADRE Internals	21
4.1	File organization, building PADRE	21
4.2	PADRE_Communication	21
4.3	Distribution Library Classes	21
4.3.1	DL_Distribution	21
4.3.2	DL_Representation	21

4.3.3	DL_Descriptor	21
4.4	Primary PADRE classes	21
4.4.1	PADRE_CommonInterface	21
4.4.2	PADRE_Distribution	21
4.4.3	PADRE_Representation	21
4.4.4	PADRE_Descriptor	22
4.5	Conditional compilation directives	22
4.6	Installing the Distribution	22
4.6.1	Supported platforms	22
4.6.2	Building and installing the distribution	23
4.6.3	Comments, suggestions, bug reports	23
5	Near-term Future Development	25
5.1	Current Status	25
5.2	Communication Abstractions	25
5.2.1	Tulip	25
5.2.2	Optimization	25
5.3	Non-array-like Data	25

Chapter 1

Introduction

The increasing complexity of parallel scientific (numerical) software, just as in all other areas of software development, demands increasing modularity and reusability of software components. For parallel scientific applications, the management of data distribution and interprocess- and interprocessor communication is an issue of some complexity that is not typically a consideration in more mainstream areas of computing.

A natural evolution of solutions to the complexity of data distribution may be observed. Early parallel machines provided ‘native’ communication libraries, e.g. CMMD on the Thinking Machines CM-5. Next came attempts at standardization, e.g. PVM, MPI, and the currently emerging MPI-2, often built on top of native libraries. While providing facilities for communication, libraries such as these do not provide support for the management or organization of distributed data.

The lack of any general-purpose distributed-data management facilities motivated the development of libraries such as Multiblock PARTI [12], PGSLib [7, 6], and KeLP [8, 1]. At this level of abstraction (unlike at the PVM/MPI level) it appears unreasonable to hope for a single satisfactory solution because of widely differing distribution and communication requirements, sometimes even within a single application or parallel library.

The sophistication of distributed-data management libraries (or *distribution libraries*) is such that it behooves the parallel application or library designer to use existing libraries rather than build them from scratch. Unfortunately, their interfaces are of sufficient complexity that the use of more than one such library, or offering the user the choice of distribution library to be used, will almost certainly greatly complicate the parallel library code, particularly in the former case wherein the same data may be alternately or even simultaneously be ‘managed’ by more than one distribution library.

1.1 PADRE

PADRE seeks to provide a uniform distributed-data management interface in the presence of:

- a multiplicity of distribution libraries (Multiblock PARTI, PGSLib, Domain Layout, KeLP, or perhaps none in the degenerate case);
- a choice of communication libraries (PVM, MPI, MPI-2, or perhaps none, again in the degenerate case);

- a choice of execution models (SPMD or threaded);
- varied computational/communication patterns, and a variety of architectures (particularly MPP, SMP and DSM, and NOW).

In addition to providing a uniform interface, by supporting the simultaneous use of more than one distribution library, with transparent conversion of data between them, PADRE makes possible higher application performance than could be obtained when restricted (by reason of software complexity) to the use of a single distribution library.

In its current initial version PADRE is heavily oriented toward the distribution of array or array-like data. This in part reflects the fact that PADRE is not an academic experiment but rather a piece of software for which there is an immediate need which influences the initial specification. Nonetheless, it is fully intended that PADRE facilitate the distribution of other kinds of data; in particular particle-like data as used in [9], for example.

1.1.1 Basic Terminology

A set of related data is called a *collection*. Examples include all of the elements of a distributed array—a *global* collection, the portion of a single array on a particular processor—a *local* collection, and similarly for a set of points.

In general the modifiers *global* and *local* will be used to distinguish entities associated with whole collections from a distinct portion of a collection on a single processor, respectively.

Following Stroustrup [11], in C++ a *class* is a *type* defined such as *C* below.

```
class C { ... };
```

A *class template* is a class parameterized on one or more types (possibly classes) for example

```
template<class Q, class R>
class CT {...};
```

wherein *CT* is a class template, *Q* and *R* are the *formal parameters* or *parameters* of *CT*. In a specific class instance, such as

```
CT<int,C>
```

the types *int* and *C* are the *actual parameters* or *arguments* of *CT*. In practice the same name may be used as the name of a type, class, or class template as for objects or instances thereof when there is no ambiguity; typically when used as a proper noun the name refers to a type, class or class template, otherwise to an object. For example “a *UserCollection*” refers to an object of type *UserCollection*.

A *domain* is a specification of a cartesian space: typically a number of dimensions, and for each dimension a *base* (index of first element), a *bound* (index of last element), and a *stride* (the increment of index between elements—one in the simplest case). Domain objects may be *normalized*, in which case the base is zero and the base and bound may be subject to *translation*—having the same constant added to both, or *unnormalized* in which case the base and bound are absolute. (The usefulness of normalized domains will become clear with the description of the PADRE_Representation class.)

1.2 Structure

At the coarsest level is

1. A application or parallel library (Application) e.g. A++/P++;
2. PADRE;
3. A distribution library (DistLib), e.g. Multiblock PARTI, PGSLib, Domain Layout, etc.;
4. A communication library (CommLib), e.g. PVM, MPI.

Figure 1.2 depicts the structure of a program using PADRE. Access to entities (types, functions, classes, etc.) defined in each component is restricted to the arcs connecting the components.

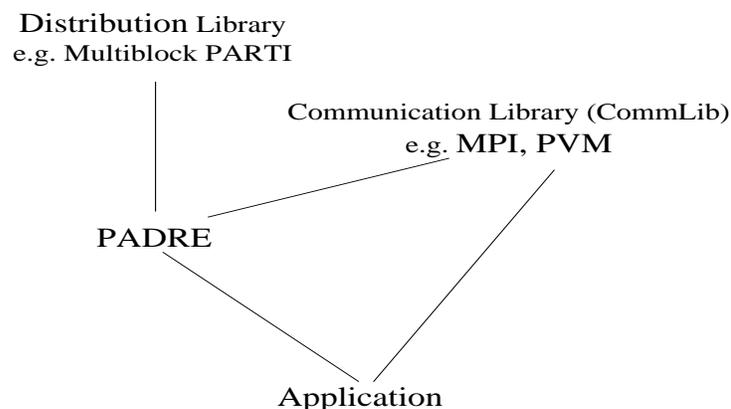


Figure 1.1: Gross Structure

Figure 1.2 shows the finer structure which is described following.

1.2.1 Application

The application will typically be a hierarchy, e.g. a low-level parallel library such as A++/P++ [?], a mid-level library such as OVERTURE [?], a high-level library such as AMR++ [?], and an application proper; in practice (e.g. in our case, with the integration of other DOE 2000 libraries) the hierarchy may be deeper. However, for the purpose of this manual such a hierarchy will collectively be called the application or *user* of PADRE.

At least conceptually, the application will have a *communication manager* (*CommMan*) with which it interfaces to the CommLib.

The major PADRE classes are actually class templates; the class arguments of these class templates are the *user classes*. PADRE provides abstract base classes for the user classes. This

effectively forces the application to provide the minimal functionality needed by PADRE to manage the application's data without overspecifying the user classes.

Typically both PADRE and the application must access CommLib.

1.2.2 PADRE

The fine structure and internals of PADRE comprise much of this manual and are described later.

The components of PADRE are the 'core' of PADRE (labelled "PADRE" in 1.2), PADRE's communication manager (*PADRE_CommMan*) which is PADRE's interface to the CommLib, and for each supported distribution library an interface to that library, for example *PADRE_Part* is the interface to Multiblock-PARTI.

1.2.3 Distribution Libraries

A distribution library such as Multiblock-PARTI provides the low-level support for the distribution of data. While it does not allocate the data, it does keep track of needed communication and generate *communication schedules*—specifications of needed communication—which it must be able to execute by external control.

1.2.4 Communication Libraries

Communication libraries such as MPI or PVM are well known and well documented [5, 4]. They provide a platform-independent protocol for interprocess communication.

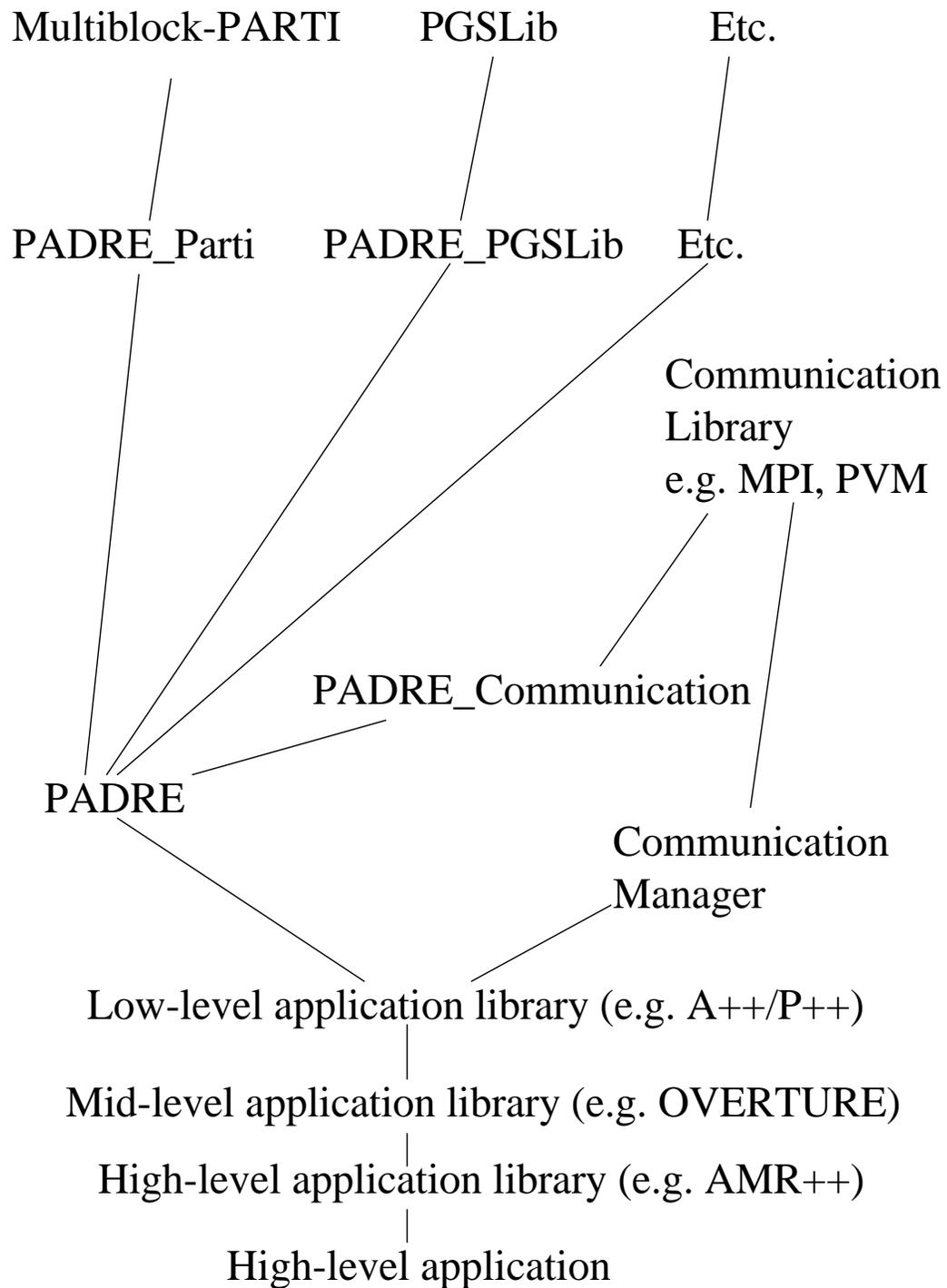


Figure 1.2: Fine Structure

Chapter 2

Abstractions

The power and usefulness of PADRE derives as much from the abstract framework that it provides as the manifest functionality of the code itself. These abstractions represent the distillation of considerable previous work spanning numerous iterations and refinement of embedded codes (that is, not distinct libraries such as PADRE) to solve much the same problems [10].

2.1 User Abstractions

The user abstractions—*domain*, *collection*, and *local descriptor*—are embodied by objects of type derived the abstract base classes *UserDomain*, *UserCollection*, and *UserLocalDescriptor*, respectively. These classes function as class template parameters for the PADRE abstractions. This mechanism forces the user to provide functionality required by PADRE whilst allowing the user the freedom to extend their functionality as needed.

2.1.1 Domains

Most abstractly, a *domain* is an index space; in the case of n -dimensional array-like objects n sets of integers. Typically, for each dimension there will be a base (index of first element), bound (index of last element), stride, and size. Thus a domain may be thought of as a *size and shape specification*.

IMPLEMENTATION NOTE: IS THIS REDUNDANT? IS ANY ONE OF BASE, BOUND, STRIDE, AND SIZE ALWAYS DETERMINED BY THE OTHER THREE?

Specific instances of *UserDomain* include:

- a *GlobalDomain*, which describes an entire array;
- a *LocalDomain*, which describes a piece of what the *GlobalDomain* describes, typically that piece residing on a single processor.

2.1.2 Collection

Objects of type *UserCollection* specify:

- the principle collection abstraction (e.g. an array, in the case the use of PADRE for an array class).

- the objects that will be represented by a single *PADRE_Distribution* object. The invocation of the member functions of the *PADRE_Distribution* then apply to all those *UserCollection* type objects where were associated with the *PADRE_Distribution*. All objects of type *UserCollection* which are associated with a specific *PADRE_Distribution* object.

2.1.3 LocalDescriptor

A *LocalDescriptor* is a descriptor for the piece of the data on a single processor. It contains

- a *UserDomain*;
- an optional pointer to the actual data (the pointer is optional because the use of PADRE permits the type of collections element be independent of the formal type of the *LocalDescriptor*. The point is that collections of many different types can be represented by a single *LocalDescriptor* and thus the distribution of many different types of collections (e.g. arrays of different type; double, float, int, etc.) can use the same PADRE objects. Alternatively it may be useful in the use of PADRE to have the raw data contained within the *LocalDescriptor* since this permits more powerful abstractions (PADRE could for example better manage the data if it had greater access to it). The reason why this is optional is that different users of PADRE have different ideas about what extent PADRE should include into the design of higher level libraries that use PADRE.

2.2 PADRE Abstractions

The major abstractions provided by PADRE are the *distribution*, the *representation*, and the *descriptor*, embodied by the class templates *PADRE_Distribution*, *PADRE_Representation*, and *PADRE_Descriptor*, respectively. The relations between these and other objects is described diagrammatically in Figure 2.2.2; details are discussed following.

2.2.1 Distribution

Roughly speaking, a distribution specifies *a set of processors, and how a collection is distributed over that set*. These processors may be virtual or physical, but the distinction is irrelevant at the level of abstraction of a distribution. It is useful to think of a distribution as a set of constraints; the implementation is free to distribute a collection in any way consistent with the constraints.

A *PADRE_Distribution* comprises a data-independent specification of a distribution, including:

- decomposition along each axis;
- specifications of ghost boundaries;
- decomposition onto subsets of processors;
- in the context of load-balancing, the fraction of a domain (collection?) on each processor.

Data-independent means, among other things, that *PADRE_Distributions* are not constrained by the particulars of a specific collection. Each *PADRE_Distribution* does, though, maintain a list of *GlobalCollection* objects (specifying size, shape, etc., as described later) using the specified distribution.

The *PADRE_Distribution* class template provides several constructors; the choice of constructor, via information about the intended distribution provided as arguments, constrains the set of distribution libraries that can support the distribution. Each *PADRE_Distribution* maintains a list of distribution library names that can support the distribution (according to the constraints it specifies); this list can change dynamically as the details of the distribution (the constraints) change.

There is a partial ordering on distribution libraries for which a greatest lower bound always exists. The ordering specifies compatibility: one distribution library is lesser than a second when the first can support any distribution that the second can. Thus for any two distributions that can be supported by at least one distribution library, there is a distribution library that can support both; this allows operations or translations between collections with differing distributions.

2.2.2 Representation

Conceptually a *representation* is a distribution plus a *GlobalDomain*.

A *PADRE_Representation*

- has a normalized global domain;
- provides a query interface for determining how data is actually distributed;
- handles abstract communication schedules.

More than one *PADRE_Representation* may be associated with a single *PADRE_Distribution*, as shown in Figure 2.2.2

2.2.3 Descriptor

A *descriptor* comprises

- a *representation*;
- a local size;
- an absolute domain;
- optionally, a pointer to data.

More than one *PADRE_Descriptor* may be associated with a single *PADRE_Representation*. *PADRE_Descriptors* are always in one-to-one correspondence with *UserLocalDescriptors*.

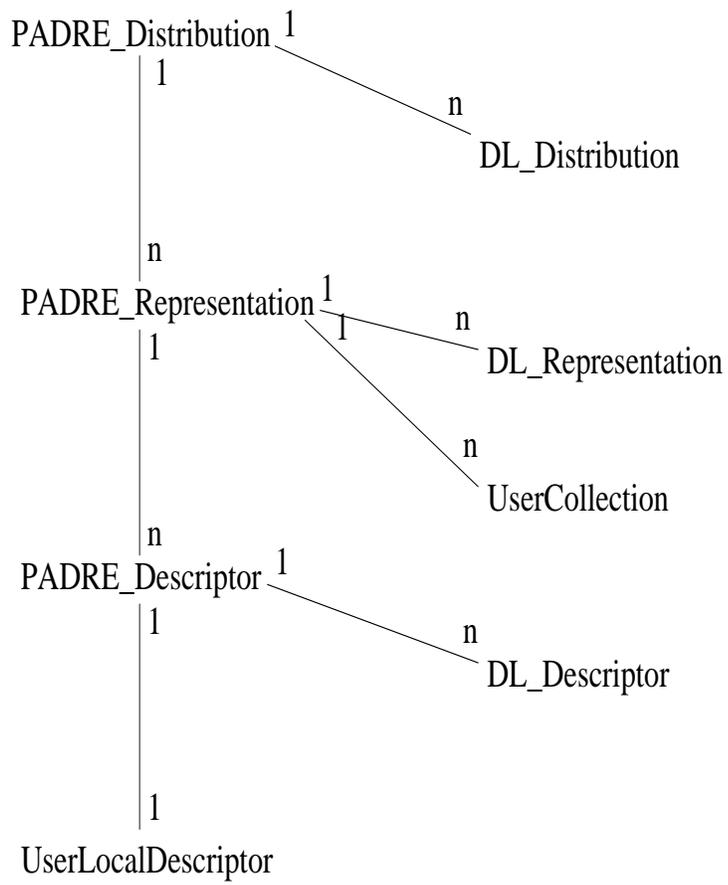


Figure 2.1: Gross Structure

Chapter 3

The PADRE API

This chapter presents a number of base classes that can be used to simplify the construction of classes used as template parameters within the PADRE objects. There are two parts to the PADRE API, the definition of the templated parameters used in teh PADRE objects and the definition of the PADRE objects themselves.

3.1 User Base Classes which simplify the PADRE API

The base classes are not available yet. The existing use of PADRE has implemented objects for the use with PADRE (as template parameters) without the benifit of these base classes (it is not significantly more difficult). The purpose of these base classes is to simplify the implementation of these classes which are used as template parameteres and required to use PADRE within a higher level library.

At a later point, the bases classes will be documented separately, for now we list the bases classes (one for each of the template parameteres that PADRE expects).

- *UserCollection*
- *UserLocalDescriptor*
- *UserDomain*

3.2 The PADRE API

The primary classes provided by PADRE are *PADRE_Distribution*, *PADRE_Representation*, and *PADRE_Descriptor*. These are actually class templates; the template parameters are the user base classes. Each of these is derived from a common base class *PADRE_CommonInterface*, but this is transparent to the user. Nonetheless, just as the base class is useful for factoring common functionality, so its description serves to similarly factor the documentation.

3.2.1 Distribution Library Names

The distribution library names are defined in a top-level class within PADRE. There purpose is to provide a way of naming the individual distribution libraries contained in PADRE.

With this naming mechanism for the distribution libraries in PADRE we can provide the following sorts of features in PADRE:

- a mechanism by which individual distribution libraries can be selected for use
- a mechanism by which individual distribution libraries can be selectively disabled
- lower level access to individual distribution libraries

```
class SubLibraryNames
{
public:
    enum libraryName { LibNONE,
                      LibTrivial,
                      LibPARTI,
                      LibDomainLayout,
                      LibPGS,
                      LibANY
                    };
    ...
};
```

This class also defines a corresponding ostream insertion operator.

As previously described, there is a partial order on distribution libraries in which a greatest lower bound exists for every set of libraries, that is, a library that can support the distributions of any given set. While interconversion is generally automatic, it is sometimes useful to be able to determine this directly.

```
SubLibraryNames::libraryName
subLibraryGreatestLowerBound( const SubLibraryNames::libraryName & X,
                              const SubLibraryNames::libraryName & Y );
```

3.2.2 PADRE_CommonInterface

While *PADRE_CommonInterface* objects are not created by the user, all of the primary PADRE classes inherit a number of public member functions from *PADRE_CommonInterface*. For completeness we document this PADRE class, however it is never seen or referenced by the user of PADRE. However, by documenting it we both specify the internal design and define the features of the different classes within PADRE in as simple a manner as possible.

```
class PADRE_CommonInterface{...};
```

Constructors, destructors, assignment

The copy assignment operator (*operator=*), copy constructor, other constructors, and destructors of (*PADRE_CommonInterface* are not intended to be accessible to the user.

Ids and reference counting

Each PADRE object is given an integer identifier at the time of creation. Each identifiers unique over all PADRE objects (e.g. a *PADRE_Distribution* cannot have the same identifier as a *PADRE_Representation*. Internally to PADRE these identifiers are used solely for internal run-time consistency checking; the user's use of them is unconstrained.

```
int getId () const { ... }    // return the unique integer id of this object
```

PADRE supports reference counting of PADRE objects. Reference counts may be incremented, decremented, and queried by the user. Each PADRE object is given a reference count of one at time of creation (ostensibly counting the reference by its own *this* pointer). From the user's point of view a reference count of zero indicates an error—internally a reference count of zero indicates that the object is in the process of destruction. The reference count should never be negative.

Upon initial construction the reference count is ZERO (or in a later implementation has the value returned by the member access function `referenceCountBase()`). Calls to the delete operator for any pointer to a PADRE object must be done only if the reference count is equal to the value of `referenceCountBase()` (or zero initially in the present implementation). Decrementing the reference count is considered part of the process of checking the reference count and deleting the object if it is equal to the `referenceCountBase()` value. Multiple pointer references to PADRE objects are expected to increment the reference count of the respective PADRE object.

```
// Example of multiple references to a PADRE object (with attention to reference counting)
Y = X;    // Copying a pointer to X should also increment the reference count
X->incrementReferenceCount();

// Example of deleting a PADRE object (with attention to reference counting)
X->decrementReferenceCount();
if (X->getReferenceCount() == X->referenceCountBase())
    delete X;
X = NULL;
```

The functions `incrementReferenceCount` and `decrementReferenceCount` are declared `const` as they may operate on *logically const* objects. (Discussions of logical versus physical *const*-ness may be found elsewhere [11, 3]).

```
void incrementReferenceCount () const;
void decrementReferenceCount () const;
int getReferenceCount () const;
```

Consistency checking

PADRE automatically performs internal consistency checking, the extent of which is determined by compile-time (of PADRE) options and user-controlled run-time settings, as described in Chapter ?? and Chapter ?. Additionally, the user may explicitly invoke a consistency check at runtime.

IMPLEMENTATION NOTE: LOTS OF DETAIL MISSING HERE.

```
void testConsistency( const char *Label = "" ) const;
```

I/O

Every class in PADRE has defined a `ostream` insertion operator `<<` to allow user-level debugging. When a PADRE object contains another PADRE object, or points to an object without an id, that object is also inserted via its insertion operator. When a PADRE object contains a pointer to a PADRE object with an id, only the id number is inserted.

Though it is not intended that objects of `PADRE_CommonInterface` be accessed directly, the pattern for the derived classes is as follows.

```
friend ostream & operator<< ( ostream & os, const PADRE\CommonInterface & X )
```

IMPLEMENTATION NOTE: FOR `PADRE_COMMONINTERFACE` THIS SHOULD BE PROTECTED RATHER THEN FRIEND?

More restricted diagnostic output is provided by the following functions.

```
static void displayDefaultValues ( const char *Label = "" );
```

```
void display( const char *Label = "" ) const;
```

IMPLEMENTATION NOTE: MUCH DETAIL MISSING HERE.

Distribution libraries

By default PADRE uses the Multiblock PARTI distribution library. The following are provided to add the name of a distribution library to the head of a list, and get the name of the library at the head of the list. These lists exist at the level of *PADRE_CommonInterface*, *PADRE_Distribution*, and *PADRE_Representation*. Each list is in priority order, with priority decreasing from the head of the list. The entire list at the *PADRE_CommonInterface* level is lower priority than the other two; it is not visible to the user and serves only to make Multiblock PARTI the default. The list at the *PADRE_Representation* level is at higher priority than the one at the *PADRE_Distribution* level.

IMPLEMENTATION NOTE: THIS SCHEME SHOULD BE UNAMBIGUOUSLY DESCRIBED—THERE’S MORE GOING ON HERE. IMPLEMENTATION NOTE: SHOULD BE ABLE TO GET THE ENTIRE LIST—IT’S AN STL LIST. IMPLEMENTATION NOTE: SHOULD BE RENAMED “DISTRIBUTIONLIBRARY” FROM “SUBLIBRARY.”

```
// add a distribution library to the head of the list
void setSubLibraryInUse( SubLibraryNames::libraryName X );

// get the sublibrary from the head of the list
SubLibraryNames::libraryName getSubLibraryInUse() const;

// adds a sublibrary to the head of the list
static void setSubLibraryInUse( SubLibraryNames::libraryName X );
\end{verbatim}
} }

{\indent
{\mySmallFontSize
\begin{verbatim}
// get the sublibrary from the head of the list
static SubLibraryNames::libraryName getSubLibraryInUse();
\end{verbatim}
} }

\subsection{Other members inherited by primary PADRE classes}
```

The functionality of the following should be clear.

```
{\indent
{\mySmallFontSize
\begin{verbatim}
// set and query ghost cell widths
virtual void setGhostCellWidth( unsigned axis, unsigned width ) = 0;
virtual int getGhostCellWidth( unsigned axis ) const = 0;

// set and get axes to be/that are distributed
virtual int getNumberOfAxesToDistribute() const = 0;
virtual void setDistributeAxis( int Axis ) = 0;
\end{verbatim}
} }

{\indent
{\mySmallFontSize
\begin{verbatim}
// get list of processors used in distribution

virtual void getProcessorSet( int* ProcessorArray,
                             int& Number_Of_Processors) const = 0;

// Update GhostBoundaries of all associated objects

virtual void updateGhostBoundaries() = 0;

static bool Has_Same_Ghost_Boundary_Widths
( const UserLocalDescriptor & Lhs_Domain,
  const UserLocalDescriptor & Rhs_Domain );

static bool Has_Same_Ghost_Boundary_Widths
( const UserLocalDescriptor & This_Domain,
  const UserLocalDescriptor & Lhs_Domain,
  const UserLocalDescriptor & Rhs_Domain );
```

3.2.3 PADRE_Distribution

The class template `PADRE_Distribution` is derived from `PADRE_CommonInterface`; the class's template arguments are the user defined classes `UserCollection`, `UserLocalDescriptor`, and `UserDomain`.

```
template<class UserCollection, class UserLocalDescriptor, class UserDomain>
class PADRE_Distribution :
public PADRE_CommonInterface{...};
```

Copy assignment, copy constructor

The semantics of copy assignment and the copy constructor are deep copy. By *deep copy* this means that all internal data is copied rather than multiply referenced internally. The later can be faster in some circumstances but makes for a much more complex semantics. Future versions of PADRE will provide for optional *shallow copy* operations, such optional behavior will reference the data internally rather than force complete recopying.

IMPLEMENTATION NOTE: THIS SHOULD BE MORE PRECISE.

Constructors and destructors

The `PADRE_Distribution` class provides several constructors; the choice of constructor dictates, for example, the default distribution library to be used and the default distribution of data.

```
// The following all default to Block Parti
PADRE_Distribution(); // distribute over all processors
PADRE_Distribution( int Number_Of_Processors );
PADRE_Distribution( int startingProcessor, int endingProcessor );
PADRE_Distribution( const int* Processor_Array, int NumberOfProcessors );
```

`PADRE_Distribution` currently has only a default destructor.

```
~PADRE_Distribution();
```

IMPLEMENTATION NOTE: NEED NEW AND DELETE OPERATORS DEFINED FOR THIS AND OTHER CLASSES.

Direct access to distribution libraries

A specific design criterion for PADRE is to avoid preventing the user from directly accessing the distribution libraries; on the other hand, direct access should never be necessary, and is intended only for those that know exactly what they are doing. Access is provided at two levels: directly through the interface of the libraries themselves (included by `PADRE.h`), and through the PADRE distribution-library layer. We describe only this PADRE layer.

```
// return reference to PADRE_PartI distribution object
PARTI_Distribution<UserCollection, UserLocalDescriptor, UserDomain> &
getpPARTI_Distribution() const
{
    PADRE_ASSERT(pPARTI_Distribution != NULL);
    return *pPARTI_Distribution;
}
```

Associating user objects with PADRE_Distributions

Various functions provide for making a breaking associations between `PADRE_Distributions` and user objects.

```
static void AssociateObjectWithDefaultDistribution( const UserCollection & X );

void AssociateObjectWithDistribution( const UserCollection & X );

static void UnassociateObjectWithDefaultDistribution( const UserCollection & X );

void UnassociateObjectWithDistribution( const UserCollection & X );
```

3.2.4 PADRE_Representation

The class template `PADRE_Representation` is derived from `PADRE_CommonInterface`; just as the class template `PADRE_Distribution` the class arguments are the user classes `UserCollection`, `UserLocalDescriptor`, and `UserDomain`.

```
template<class UserCollection, class UserLocalDescriptor, class UserDomain>
class PADRE_Representation :
public PADRE_CommonInterface{...};
```

Copy assignment, copy constructor

The semantics of copy assignment and the copy constructor are deep copy.

IMPLEMENTATION NOTE: THIS SHOULD BE MORE PRECISE.

Constructors and destructors

```
~PADRE_Representation();

PADRE_Representation(); // default is ??

PADRE_Representation( const PADRE_Representation & X );

// The following take UserDomain as a const & since we build a translation
// independent UserDomain internally (i.e we will not be referencing it
// so we don't have to pass it as a pointer) If we just pass in a
// UserDomain object then we assume that the defaults in the
// PADRE_Distribution object will be used (else we should have specified
// a specific distribution object to guide the partitioning onto the
// multiprocessor environment.

PADRE_Representation( const UserDomain & inputGlobalDomain );

PADRE_Representation( const UserDomain & inputGlobalDomain,
                      PADRE_Distribution<UserCollection,
                      UserLocalDescriptor,
                      UserDomain>
                      *distribution );
```

Misc.

```
bool isLeftPartition(int i) const;
bool isMiddlePartition(int i) const;
bool isRightPartition(int i) const;
bool isNonPartition(int i) const;

// return pointer to corresponding PARTI representation
PARTI_Representation<UserCollection,
                    UserLocalDescriptor,
                    UserDomain>*
getPARTI_Representation () const;
```

```
// But we also need a reference to the PADRE_Distribution object that is in use
// This has to be a dynamic link (so that the association can be broken) so we
// need a dynamic reference (i.e. a pointer rather than a C++ reference)
PADRE_Distribution<UserCollection, UserLocalDescriptor, UserDomain> *Distribution;

// list of all UserCollection objects that share this representation
list<UserCollection*> associatedUserCollections;
```

3.2.5 PADRE_Descriptor

This class is derived from PADRE_CommonInterface and is templated on the user classes UserCollection, UserLocalDescriptor, and UserDomain.

```
template<class UserCollection, class UserLocalDescriptor, class UserDomain>
class PADRE_Descriptor :
public PADRE_CommonInterface{...};
```

3.2.6 Copy assignment, copy constructor

The semantics of copy assignment and the copy constructor are deep copy [this should be more precise].

3.2.7 Constructors and destructors

For now, the effect of choice of constructor is given by the comments.

```
// Null Descriptor Constructor (no data, default distribution, representation pointer is NULL)
PADRE_Descriptor();

// Null Descriptor Constructor (i.e. no data, but a specific distribution,
// representation pointer is NULL)
PADRE_Descriptor( PADRE_Distribution<UserCollection, UserLocalDescriptor, UserDomain>
                  *User_Distribution );

// Null Descriptor Constructor (i.e. no data, but a specific distribution and representation)
// This might not make so much sense because no data is associated with the representation.
PADRE_Descriptor( UserDomain *GlobalDomain,
                  PADRE_Representation<UserCollection, UserLocalDescriptor, UserDomain>
                  *inputRepresentation );

// Valid Descriptor Constructor with specific size of data and distributin
PADRE_Descriptor( UserDomain *GlobalDomain,
                  PADRE_Distribution<UserCollection, UserLocalDescriptor, UserDomain>
                  *User_Distribution );

// Valid Constructor (builds distriptom for specific size of
// data and using default distribution)
PADRE_Descriptor( UserDomain *GlobalDomain );
```

[Where's the destructor???

Public Data

This MUST be changed to use access functions!

```
// We want the PADRE_Representation objects to be translation independent so
// we have to store the specific domain in the PADRE_Descriptor object
// this allows multiple PADRE_Descriptor objects to use the same
// PADRE_Representation object with different bases for the data
UserDomain *globalDomain;

// pointer to parent PADRE_Representation
PADRE_Representation<UserCollection, UserLocalDescriptor, UserDomain> *representation;

// Pointer to the local descriptor which would contain a pointer to the data
// many details of the local descriptor are accessible through access functions
UserLocalDescriptor *LocalDescriptor;
```

Access to LocalDescriptor

```
const UserLocalDescriptor* getUserLocalDescriptorPointer() const { return LocalDescriptor; }
const UserLocalDescriptor & getUserLocalDescriptor()      const { return *LocalDescriptor; }
```

Communication

```
PADRE_CommunicationSchedule buildCommunicationSchedule( const PADRE_Descriptor & X );
```


Chapter 4

PADRE Internals

The purpose of this chapter is describe the internal organization and coding conventions of the PADRE source code, primarily as an aid to adding a distribution library or communication library. If you are not adding a new distribution library then this chapter should be skipped.

The design of PADRE includes the ability to use existing libraries that describe specific distribution mechanisms. This chapter is a description of what is required to add a new distribution library to PADRE.

This chapter remains largely incomplete. Until we have users of PADRE who want to add specific distribution mechanism through the addition of new or existing distribution libraries this aspect of the documentation of PADRE will likely be of lower priority than the rest of PADRE and the implementation of PADRE itself.

4.1 File organization, building PADRE

4.2 PADRE_Communication

PADRE_Communication contains CommLib-specific part of PADRE.

- execute communication schedules.

4.3 Distribution Library Classes

PADRE_DistLib interfaces to the distribution libraries (DistLib), e.g. PARTI.

- Reports processor set, either stored internally or from the appropriate DistLib.

4.3.1 DL_Distribution

4.3.2 DL_Representation

4.3.3 DL_Descriptor

4.4 Primary PADRE classes

4.4.1 PADRE_CommonInterface

4.4.2 PADRE_Distribution

4.4.3 PADRE_Representation

A PADRE_Representation

- has a global domain that is translation independent
- query interface: what is where?
- handles abstract communication schedules

4.4.4 PADRE_Descriptor

4.5 Conditional compilation directives

This section details the options associated with compiling PADRE and generating the internal debugging that is implemented within PADRE.

Logic: NDEBUG is C(++) intrinsic--if defined, the intrinsic `assert(_)` has vacuous semantics, and `PADRE_ASSERT(_)` should also have vacuous semantics. If NDEBUG is not defined, `PADRE_NDEBUG` may be defined, in which case `PADRE_ASSERT(_)` should be vacuous, otherwise it should behave much like `assert(_)`.

Independently, if `COMPILE_DEBUG` is defined, debug code should be compiled with the level of debugging governed by the public variable `PADRE::debugLevel`, an integer with nominal range 0 (for no debugging) to 10.

Finally, the class PADRE defines a public function `setErrorChecking(boolean)` which if called with TRUE should cause fairly exhaustive run-time consistency checking. These checks SHOULD NOT USE `PADRE_ASSERT`, but should use `PADRE_ABORT`. This is to prevent the condition when NDEBUG or `PADRE_NDEBUG` is defined and `setErrorChecking` is on resulting in error checking with errors found not being reported.

The constants `NDEBUG`, `PADRE_NDEBUG`, and `COMPILE_DEBUG` should only be defined as a compiler argument, not in the source code.

[IM(mkd)HO having both `PADRE_ASSERT` and `setErrorChecking` is redundant, leading to e.g. the complication just described.]

4.6 Installing the Distribution

At present the installation mechanism does not use GNU autoconf. The present makefiles within the distribution are short, simple and quite general to a number of architectures.

4.6.1 Supported platforms

We support, or intend to support, the following hardware platform, OS, and compiler combinations:

- Sun SPARC, Solaris (SunOS 5.x)
 - KAI KCC [done]
 - Sun CC [done]
 - Gnu CC (gcc) [or the enhanced GCC, to do]
- SGI (the common workstation), IRIX ???
 - KAI KCC [to do]
 - SGI CC [to do]
 - Gnu CC (gcc) [or the enhanced GCC, to do]
- SGI Origin 2000 (specifically the ASCI Blue), IRIX ???
 - KAI KCC [to do]
 - SGI CC [to do]
 - Gnu CC (gcc) [or the enhanced GCC, to do]

- Intel Pentium and up, Linux (kernel 2.x)
 - KAI KCC [to do]
 - Gnu CC (gcc) [or the enhanced GCC, to do]

We intend to support a more general collection of architectures as work proceeds. This will be done on an as needed basis, in particular as it is requested. Please send suggestions as to which architectures are most important. At present the code is quite general and not specific to any architecture, we have had good success working with it on a number of architectures but we will provide specific makefile details to handle each of numerous architectures and C++ compilers in the future.

At present there is no multilanguage support to permit the use of PADRE with C or Fortran (77 or later). This will change in the future, see the section of the manual related to Future Plans.

4.6.2 Building and installing the distribution

The use and development of software as part of ACTS is an important part of our focus. We want to make these steps as simple as possible to permit the easy use of PADRE within ACTS and within other people's work. PADRE as a library is intended for use within other libraries and so it is expected that these libraries will likely simplify their distribution requirements by including PADRE directly into their distributions. However does not readily allow the latest versions of PADRE to be used, so this consideration should be weighed against the more general use and installation of PADRE separate from the libraries that use it.

Current plans for how PADRE is installed are incremental. Eventually the user will: get the tar file from the web, untar, run autoconf, run make all, run make install, etc. However for now until autoconf is in place and until we have the paperwork in place to distribute PADRE on the web we have a more simple procedure: If not at Los Alamos, send email keilanl.gov to arrange for an ftp site. Otherwise, if at Los Alamos, go to `/users/kei/padre/`, type `cp -r padre <wherever>`, `cd <wherever>`, `make clean`; `make all`. It's pretty obvious how to change the current Makefile to use either Sun CC or KCC C++ compilers. The Makefiles are currently short and simple.

4.6.3 Comments, suggestions, bug reports

Initial bug reports should be sent to: keilanl.gov. Later a more formal mechanism will be setup for comments, suggestions, bug reports.

Chapter 5

Near-term Future Development

Rapid and recent developments within the PADRE library have complicated the process of keeping the documentation up-to-date. This chapter outlines a few of the most recent features of PADRE that are still in progress or being started soon.

5.1 Current Status

At this point in time (the state of integration of distribution libraries into PADRE is as follows.

- *Multiblock PARTI*: fully integrated;
- *PGSLib*: being integrated;
- *Domain Layout*: awaiting release of Domain Layout;
- *KeLP*: KeLP is somewhat more than just a distribution library; we are working with the authors to appropriately restrict KeLP to be usable by PADRE.

Up-to-date status may be found at the PADRE home page (or send us email for details).

5.2 Communication Abstractions

There are several issues regarding communications that will be addressed.

5.2.1 Tulip

Tulip [2] is a C++ run-time library that supports communication via *global pointers*, in particular for distributed shared memory architectures. Like PADRE, it is a component of the DOE 2000 SciTL project. Tulip will be an optional communication layer for PADRE; referring to Figure ??, it will be interposed between *PADRE_CommMan* and the Communication Library.

5.2.2 Optimization

Independent of incorporation of Tulip there are enhancements to be made to the handling of communications. *Communication schedules*—specifications of communications to be performed—will be encoded in a form independent of the underlying communications libraries, and in such a manner as to be amenable to various optimizations. These optimizations will include:

- high-level caching;
- temporal and spatial optimization (e.g. staging communications, routing all-to-all communications).

5.3 Non-array-like Data

As mentioned in Chapter 1, PADRE is currently oriented toward the distribution of array or array-like data; as in other aspects of PADRE, this reflects the fact that it was developed to be deployed into real-world use as soon as possible. Nonetheless, PADRE is intended to handle the distribution of other kinds of data. In particular, functionality for the distribution of particle-like data will be added.

Bibliography

- [1] Scott Baden. The kelp programming system. <http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html>.
- [2] Pete Beckman, Dennis Gannon, and Elizabeth Johnson. Tulip—a parallel run-time system class library for c++ frameworks. WWW page. <http://www.acl.lanl.gov/beckman/rts/>.
- [3] James O. Coplien. *Advanced C++*. Addison-Wesley, 1992.
- [4] Al Geist et. al. *PVM—Parallel Virtual Machine*. MIT Press, 1994.
- [5] Marc Snir et. al. *MPI—The Complete Reference*. MIT Press, 1996.
- [6] Robert C. Ferrell. Cambridge power computing associates. <http://www.cpga.com/>.
- [7] Robert C. Ferrell, Douglas B. Kothe, and John A. Turner. Pgslib: A library for portable, parallel, unstructured mesh simulations. In *Proceedings of 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
- [8] Stephen J. Fink and Scott Baden. Run-time support for multi-tier programming of block-structured applications on smp clusters. In Ishikawa et al., editor, *International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 97*, volume 1343 of *LNCIS*. Springer, 1997.
- [9] Jeremiah P. Ostriker and Michael L. Norman. Cosmology of the early universe viewed through the new infrastructure. *Communications of the ACM*, 40(11):84–94, November 1997.
- [10] Dan Quinlan and Rebecca Parsons. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
- [11] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition edition, 1997.
- [12] Alan Sussman and Gagan Agrawal. A manual for the multiblock parti runtime primitives. ftp://hpsl.cs.umd.edu/pub/block_parti_distribution/doc.ps.gz, January 1994.