

KeLP Reference Manual

Version 1.4

Scott B. Baden

Richard B. Frost

Daniel Shalit

February, 22, 2001

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114 USA

<http://www.cse.ucsd.edu/groups/hpcl/scg/kelp>

Copyright © 2000,2001 by Scott B. Baden, and The Regents of the University of California. All rights reserved. This software is offered on an "AS IS" basis. The copyright holders provide no warranty, expressed or implied, that the software will function properly or that it will be free of bugs. This software may be freely copied and distributed for research and educational purposes only, provided that the above copyright notice appear in all copies. A license is needed for commercial sale, in whole or in part, from The Regents of the University of California. Users of the software agree to acknowledge the authors. The development of the software was supported in part by the federal government and they may have certain rights to it. This work was supported by the DOE Computational Science Graduate Fellowship Program, NSF contract ASC-9520372, and the National Partnership for Advanced Computational Infrastructure (NPACI) under NSF contract ACI-9619020. Portions of this document are reproduced from the LPARX User's Guide, v2.0, and the KeLP v1.0 User's Guide, with the authors' permission. The current release of the KeLP programming system is based on version 1.0, parts of which are included by permission from the authors.

Table of Contents

1	Overview of the KeLP classes	3
2	Fundamental classes	4
3	Looping iterators and macros	15
4	Fortran Interface	17
5	Dock	18
6	DGrid	20
7	API-FD	22
8	MP++	24
A.	Appendix: The Patch Specification	31

1 Overview of the KeLP classes

The KeLP software is implemented as a C++ class library. There are seven main classes implemented by KeLP:

`Point`, `Region`, `FloorPlan`, `Grid`, `XArray`, `MotionPlan`, `Mover` A `Point` is an integer vector and a `Region` is a rectangular index set. A `Grid` is an array defined over a `Region` and a set of solution fields. `Grid` elements may be standard C++ types (double, int, float) or structures. An `XArray` is a one-dimensional array of `Grids` all of the same type. `XArrays` are parallel arrays over which KeLP provides a coarse-grained parallel looping construct. A `MotionPlan` is a distributed representation of a block communication pattern. A `Mover` carries out the communication pattern stored in a `MotionPlan`.

1.1 Dimensions of the Classes

The current implementation of KeLP supports dimensions one through four. All KeLP classes are typed according to dimension. The following static types are defined for efficiency:

```
1d:  Point1, Region1, Grid1, XArray1, Array1, FloorPlan1,
      MotionPlan1, Mover1
2d:  Point2, Region2, Grid2, XArray2, Array2, FloorPlan2,
      MotionPlan2, Mover2
3d:  Point3, Region3, Grid3, XArray3, Array3, FloorPlan3,
      MotionPlan3, Mover3
4d:  Point4, Region4, Grid4, XArray4, Array4, FloorPlan4,
      MotionPlan4, Mover4
```

1.2 Error checking

KeLP classes provide error checking, although this feature can be turned off during the compilation of KeLP. Errors are handled by simply printing out an error message and aborting the program (e.g. in the case of memory exhaustion), or, if possible, throwing an exception. Exceptions thrown by various KeLP classes are described in the subsequent pages of this document. The application program should try to catch these exceptions (in case of an uncaught exception, the program will simply abort). Note that all KeLP exceptions are derived from `KelpErr`. Consequently, the programmer can catch all of them by inserting the following try and catch phrase in the main program:

```
main () {
    MPI_Init(&argc,&argv);
    InitKeLP(argc,argv);

    try {
        // application code
    }
    catch (KelpErr & ke) {
// print out an error message and abort the program
        ke.abort();
    }

    MPI_Finalize();
    return(0);
};
```

In the above example, `KelpErr` or any other exception derived from `KelpErr` will be caught, and the appropriate `abort()` function will be called (`KelpErr::abort()` is a virtual function, redefined for each exception derived from `KelpErr`).

2 Fundamental classes

2.1 Class Point

Class `PointX` implements a simple X -dimensional integer vector.

Creating Points:

```
Point2 p(2,3);           // two dimensional vector (2,3)
Point4 q(1,2,3,4);      // four dimensional vector (1,2,3,4)
```

Operations on Points:

```
p(i)                    // extract the ith integer from p
*, +, -, /              // arithmetic operations on two points
*=, +=, -=, /=         // arithmetic operations on two points
==, !=                  // comparison of two points
min, max                // member functions for elementwise
                        // min/max
<<, >>                 // output and input points
```

Note that the arithmetic operations are also defined for a `Point` and an integer and return `Point`-valued results. For example, `q+1` and `p*2+1` return `[2,3,4,5]` and `[5,7]`, respectively.

`DimErr` exception can be thrown by `Point` member functions that take a dimension as a parameter. For example:

```
try {
    Point2 p(2,3);           // p is 2 dimensional
    int result = p(i);      // i is valid if it is either 0 or 1
}
catch (DimErr) {
    // error: the value of i is invalid!
}
```

For more information, consult `./array/PointX.h.m4`, `./array/PointX.C.m4`, and `./array/DimError.h` in the KeLP source code distribution.

2.2 Class Region

Class `RegionX` implements X -dimensional regions of integer index space. Regions are defined by lower and upper bounds. If the upper bound of the region has an element which is less than the corresponding element in the lower bound, then the region is considered "empty" and represents an empty set in index space.

Creating Regions:

```
Region3 empty;           // create an empty region

Point2 pl(1,2);          // create a region with lower bound pl
Point2 pu(2,3);          // and upper bound pu
Region2 r(pl, pu);

Region2 r(1,2,4,5);      // region [1,2] x [4,5]
Region2 r(6,7);         // region [0,0] x [5,6]
```

Note in the last case that the Region2 declaration is similar to how C/C++ performs array declarations. The following shows how we can manipulate Regions (assume R and S are Regions with the same number of dimensions):

```

    R.lower()           // lower bound of R (Point valued)
    R.lower(i)         // ith element of lower bound
(integer)
    R.upper()          // upper bound of R (Point valued)
    R.upper(i)         // ith element of upper bound
(integer)
    R.extents()        // upper()-lower()+1
    R.extents(i)       // upper(i)-lower(i)+1
    R.empty()          // TRUE if R is the empty set
    R.size()           // number of points in R
    R + S, R += S      // bounding box of the union of R and
S
    R * S, R *= S      // intersection of regions R and S
    R == S, R != S    // comparison of regions R and S
    <<, >>            // stream I/O output and input of
                        // regions
    S = shift(R,p)     // return region R shifted by point p

```

Regions can be ``grown" as follows:

```

    R.grow(1)           // grow R by 1 uniformly in all
                        // dimensions
    R.grow(p)           // grow R as specified by the point p
    S = grow(R,i)       // return R grown by integer i
    S = grow(R,p)       // return R grown by point p

```

For example,

```

Region3 R(1,3,4,5,6,8); // [1:5, 3:6, 4:8]
Region3 S = grow(R, 2); // [-1:7, 1:8, 2:10]
Point3  p(2,1,3);
Region3 T = grow(R, p); // [-1:7, 2:7, 1:11]
Region3 U = grow(R,-1); // [2:4, 4:5, 5:7]

```

Note that negative growth factor is also allowed.

DimErr exception can be thrown by Region member functions that take a dimension as a parameter. For example:

```

try {
    Region2 r(6,7); // r is 2 dimensional
    int result = r.lower(i); // i is valid if it is either 0 or 1
catch (DimErr) {
    // error: the value of i is invalid!
}

```

For more information, refer to ./array/RegionX.h.m4, ./array/RegionX.C.m4, and ./array/DimError.h.

2.3 class FloorPlan

Class FloorPlanX implements a one-dimensional array of X-dimensional Regions where each Region has a processor assignment. Thus, a FloorPlan represents an irregular block data decomposition.

Creating FloorPlans:

```
FloorPlan3 empty;           // create an empty Floorplan
FloorPlan2 F(5);           // create a FloorPlan with 5 entries
```

The following are some member functions to use to manipulate FloorPlans.

```
F.size()                   // number of regions in F (integer)
F.resize(i)                // change the size of F
F(i)                       // the ith element of F
F.setowner(i, proc)        // set processor owner of F(i)
F.setlower(i,P)            // set F(i).lower()
F.setupper(i,P)           // set F(i).upper()
F.setempty(i)              // set F(i) to empty Region
F.setregion(i,R)          // set F(i) = R
F.grow(i,P)                // apply Region grow operation to F(i)
F.GlobalBoundingBox()     // returns the bounding box of the
                          // Region elements of the FloorPlan
```

Two types of exceptions can be thrown by FloorPlan : DimErr and BndErr. DimErr exception can be thrown by FloorPlan member functions that take a dimension as a parameter. BndErr exception can be thrown by FloorPlan member functions that take an index as a parameter. For example:

```
try {
    FloorPlan2 F(5);           // F is 2 dimensional and has 5
                              // elements
    F.setowner(i,proc);       // i is valid if it is between 0 and 4
    F.setlower(j,k,value);    // k is valid if it is either 0 or 1
} catch (BndErr) {
    // error: the value of i or j is invalid!
}
} catch (DimErr) {
    // error: the value of k is invalid!
}
}
```

2.4 Class Grid

Class GridX was a fundamental class in earlier versions of KeLP. With the advent of Abstract KeLP and the Patch abstraction it has been “demoted” to a utility class. Nonetheless it is sufficiently useful that we describe it in detail here and provide this class as the default Patch implementation in this release.

GridX implements an X by Nfields dimensional array defined on a region of integer index space (a subset of Z^X). A GridX of type T has the class name:

```
GridX<T>
```

where X is 1, 2, 3, or 4, Nfields is the number of solution fields, and T is some C++ type such as double, int, float, or a user-defined class or structure. For example,

```
Grid1<int>                 // one dimensional grid of integer
```

```

Grid3<double>           // three dimensional grid of double
Grid4<UserClass>       // four dimensional grid of UserClass

```

Currently user-defined classes must be of fixed-size which can be queried by the `sizeof()` function. Grid definitions take the following form:

```

GridX<T> name(R, F);

```

where:

```

X is the dimension of the Grid
T is the Grid element type
name is the Grid name
R, F are region specifiers

```

For example,

```

Region1 F(1,3);

```

```

Grid3<double> Velocity(R,F);

```

is an array of doubles defined over some region R and having as its solution fields the three vector components of velocity.

The `Region1`, F, defaults to (1,1), in which case `GridX` is a simple X-dimensional array.

Note that in nearly all cases the programmer will not directly create Grids through the Grid constructor, but will instead rely on the XArray class to create Grids as needed.

The index set of the Grid is set by the region specifier R, which may can be either be a `RegionX`-valued expression or a comma-delimited list of X integers. If the latter form is specified, then the index set defaults to conform to standard C/C++ indexing:

```

Region2 r(1,1,6,6);           // region [1:6, 1:6]
Region1 fields(1,3)           // region [1:3]
Grid2<int> h(r, fields);      // h has index set [1:6, 1:6, 1:3]
    // The line above is by far the most common case.
Grid3<int> g(3,4,5, cs, ce);  // g has index set [0:2, 0:3, 0:4, cs:ce]

```

We can extract information about the index set of a Grid using the functions `lower()`, `upper()`, `extents()`, and `region()`. The functions `lower()`, `upper()`, and `extents()` are defined exactly as in the case of a Region. The `region()` member function returns the Region of a Grid. For example:

```

Grid2<double> g(Region2(i0,j0,i1,j1));
g.lower(0)    == i0
g.lower()    == Point2(i0,j0)
g.upper(1)   == j1
g.extents()  == Point2(i1-i0+1,j1-j0+1)
g.extents(1) == j1-j0+1
g.region()   == Region2(i0,j0,i1,j1)

```

Grid is supplied with an implicit cast to a `const Region`. Thus we may also apply any binary Region operation to Grids. For example, if G_1 and G_2 are Grids, then $G_1 * G_2$ will return $G_1.region() * G_2.region()$.

Data elements of a Grid may be accessed through the `data()` member function or the indexing operator `()`. Function `data()` is most often used when communicating with external functions, such as Fortran

numerical kernels, from C++. It returns a pointer to the first element of the array data, which is stored in Fortran standard column-major order:

```
double *p = g.data();           // pointer to first element of data
p[0] = ...                      // first data element of g
```

The subscripting operator () returns the specified element of a Grid:

```
Grid2<double> g(Region2(i0,j0,i1,j1), Region1(f0,f1));
double a = g(i0,j0,f0);         // first element of the grid
```

Five types of exceptions can be thrown by Grid: DimErr, BndErr, NonLocErr, RegionSubErr, and RegionSizeErr. DimErr exception can be thrown by Grid member functions that take a dimension as a parameter. BndErr exception can be thrown by Grid member functions that take an index as a parameter. For example:

```
try {
    Grid2<int> g(3,4);           // g is 2 dimensional with index set
                                // [0:2, 0:3]
    g(i, j) = 1;                // i is valid if it is between 0 and 2
                                // j is valid if it is between 0 and 3
    int result = g.lower(k);    // k is valid if it is either 0 or 1
catch (BndErr) {
    // error: the value of i or j is invalid or g is non-local!
}
catch (DimErr) {
    // error: the value of k is invalid!
}
```

NonLocErr, RegionSubErr, and RegionSizeErr exceptions can be thrown by the Copy(), SerializeOut(), and SerializeIn() member functions of Grid. For example:

```
try {
    target.CopyRegion (to, source, from);
}
catch (NonLocErr) {
    // error: non-local Grids!
}
catch (RegionSubErr) {
    // error: invalid Regions or Grids!
}
catch (RegionSizeErr) {
    // error: invalid Regions!
}
```

If target or source Grids are not local, NonLocErr will be thrown. If to is not a subregion of target.region(), or if from is not a subregion of source.region(), RegionSubErr exception will be thrown. Finally, if to and from Regions have different sizes, RegionSizeErr will be thrown.

For more information, consult ./kelp/GridX.h.m4, ./kelp/GridX.C.m4, ./tools/RegError.h, ./kelp/NonLocError.h, ./array/DimError.h, and ./array/BndError.h.

2.5 Class XArray

Class `XArrayX` implements a 1-dimensional parallel array of `X`-dimensional `Grids`¹, each of which may be owned by an arbitrary processor. The shape and processor owners for an `XArray` may be represented by a `FloorPlan`. KeLP defines coarse-grained looping constructs over `XArrays`. An `XArray` of `Grids` of dimension `X` of type `T` has the class name:

```
XArrayX<GridX<T> >
```

where `X` is 1, 2, 3, or 4 and `T` is some fixed-length type. For example:

```
XArray3<Grid3<double> >      // XArray of 3d Grids of type double
XArray2<Grid2<int> >         // XArray of 2d Grids of type int
```

`XArray` passes the necessary arguments to the `Grid` constructor. As with `Grid`, the `fields` argument is optional.

An `XArray` may be declared as follows:

```
XArray2<Grid2<double> > X;      // Empty XArray
XArray2<Grid2<double> > X(n);  // XArray of n Grids
XArray2<Grid2<double> > Y(FP, fields);
// XArray with decomposition stored by FloorPlan FP.
```

If an `XArray` is not declared with a `FloorPlan`, then at some future point the `FloorPlan` must be specified and `Grid` storage allocated. This is accomplished with the `instantiate` member function:

```
X.instantiate(FP);           // install decomposition of FloorPlan FP
```

An `XArray` provides a variety of member functions to access and manipulate its structure and data. Examples include:

```
X.floorplan(i)              // the ith member of X's FloorPlan
                             // (a Region with an owner)
X.size()                    // number of Grids
X.owner(i)                  // processor owner of ith Grid
X(i)                        // ith Grid in X
X.resize(i)                 // resize the whole XArray
X.ptr(i)                    // pointer to ith Grid
```

KeLP provides `nodeIterator` to iterate over `XArray` elements in parallel. The following code will execute the loop body once for each `Grid` in the `XArray X`. The body is executed on the processor that owns `Grid X(i)`.

```
for (nodeIterator iter(X); iter; ++iter) {
    int i = iter();           // query the iterator for current index
    do something
}
```

`nodeIterator` supersedes the `for_all` loop provided in the previous versions of KeLP for the same purpose:

```
for_all(i,X)
```

¹ For advanced users who wish to supply their own container class, simply replace `GridX<type>` with `myPatch` in this and the following discussions.

```

do something
end_for_all

```

Two types of exceptions can be thrown by `XArray`: `DimErr` and `BndErr`. `DimErr` exception can be thrown by `XArray` member functions that take a dimension as a parameter. `BndErr` exception can be thrown by `XArray` member functions that take an index as a parameter. For example:

```

try {
    XArray2<Grid2<double> > X(FP);
    int owner = X.owner(i);          // i is valid if it is between 0
                                     // and F.size()
    int lower = X.lower(j);         // j is valid if it is 0
    catch (BndErr) {
        // error: the value of i is invalid!
    }
    catch (DimErr) {
        // error: the value of j is invalid!
    }
}

```

2.6 Class DataFactory

The basic `XArray` constructor is

```

XArrayX(const FloorPlanX &D, const Region1 comps = Region1(1,1),
const DataFactoryX<UserPatch>& factory = DataFactoryX<UserPatch> ())

```

Internally, `XArray` uses the `DataFactory` to create `Grids`:

```

_data[k] = *factory.instantiate( D(nn), comps);

```

As described in the Users Guide, a default `DataFactory` is provided with the distribution and the `DataFactory` argument is optional. This convenience comes with a small price: since `XArray` expects an argument of type `DataFactory`, a user supplied `Factory` class must be a subclass of the default.

As a simple example, it is sometimes useful to create an array of `Grids` which have a definite region and fields but which do not allocate storage for data. Class `GridX` has an overloaded constructor which takes as an additional argument a flag which tells it whether or not to allocate storage. To accomplish this,

Define an appropriate `Factory` class:

```

template <class T>
class EmptyDataFactory2: public DataFactory2<T>
{
    const int _alloc;
public:
    EmptyDataFactory2(){}
    EmptyDataFactory2(const int alloc):_alloc(alloc){}
    virtual ~EmptyDataFactory2(){}

    virtual T* instantiate( Region2 FP, Region1 comps )
    {
        return new T(FP, _alloc, comps);
    }
};

```

In the application code:

```
EmptyDataFactory2<Grid2<complex> > Empty(FALSE);
XArray2<Grid2<complex > > Temp(F,Region1 (1,1),Empty);
```

2.7 Class MotionPlan

A `MotionPlanX` describes a data motion pattern for X-dimensional objects. KeLP supports communication between `Grids` over arbitrary `Regions`. A `MotionPlan` encodes a set of `Region` copy operations to be performed atomically. The set of data motion operations specified in a `MotionPlan` can then be carried out by a `Mover` object.

When created, a `MotionPlan` contains no `Region` copy operations.

```
MotionPlan3 M;          // declare empty MotionPlan for 3-dimensional
                        // Grids
```

A `MotionPlan` is built incrementally, one operation at a time, using the `Copy()` member function. For example, suppose `F` and `G` are `FloorPlans`, `F`, `Y` are `XArrays`, and `Q`, `R` are `Regions`. Then

```
M.Copy(F,i,Q,G,j,R);
```

represents the block copy operation "copy from any `XArray F(i)` distributed according to `FloorPlan F` over `Region Q` into some `XArray G(j)` distributed according to `Floorplan G` over `Region R`." `MotionPlan M` will be modified to store this operation. Thus, by having the processors issue a sequence of `Copy` operations to a `MotionPlan` a communication pattern may be built, one block copy operation at a time.

There are two forms of the `CopyOnIntersection` function provided for syntactic convenience.

```
M.CopyOnIntersection(F,i,G,j); // copy from F(i) into G(j) where
                                // their domains intersect
M.CopyOnIntersection(F,i,G,j,R); // copy from F(i) into G(j) where
                                // their domains intersect Region R
```

It is important to note that the family of `Copy()` functions do not actually copy anything. Rather, as the *class* name implies they encode a *plan* of block copy operations for the `Mover` to execute.

KeLP also provides several operations to directly modify existing `MotionPlans`. These operations are intended for the advanced KeLP user will be described in more detail in a future version of this document.

Exceptions raised. `BndErr` exception can be thrown by `MotionPlan Copy` and `CopyOnIntersection` functions. If `i` is not a valid index of `F`, or `j` is not a valid index of `G`, `BndErr` exception will be thrown. For example:

```
try {
    M.Copy (F,i,Q, G,j,R);
} catch (BndErr) {
    // error: the value of i or j is invalid!
}
```

For more information, refer to `./kelp/MotionPlanX.h.m4`, `./kelp/MotionPlanX.C.m4`, and `./array/BndError.h`.

2.8 Class Mover

A `Mover` is an object that will execute the set of block copy operations specified in a `MotionPlan`. Although a great deal of work is done internally by the `Mover`, from the simplest user's viewpoint (s)he has done all the work by the time the `Mover` is invoked. The next subsection will describe the `Mover` in enough detail to allow utilization of its built in capabilities. Later subsections will probe deeper for those users who may wish to go beyond this and custom build their own `Mover` object.

Mover Basics

A `MoverX` is built to perform communication over `XArrays` of `Grids` of dimension **X** and type **T** and has the class name

```
MoverX<GridX<T> >
```

where **X** is 1, 2, 3, or 4 and **T** is a basic data type, e.g. float, double, etc.

When a `Mover` is declared it is bound to a pair of `XArrays`, the *sender* and *receiver*, and to a `MotionPlan` *S*. Generically a `Mover` declaration looks like

```
MoverX<GridX<T> > Mov(Sender,Receiver,S);
```

For example, a `Mover` to communicate data between 3D `Grids` of doubles would be declared

```
Mover3<Grid3<double> > Mov(Sender,Receiver,S);
```

A call to the `execute()` member function,

```
Mover.execute();
```

will then perform all communications specified in the `MotionPlan` to which the `Mover` is bound. The call to `execute()` returns when all communication is complete.

```
/* *****  
 * MoverX<tGRID>::execute() *  
 * Execute a schedule *  
 *  
 * *****/  
template <class tGRID>  
void MoverX<tGRID>::execute()  
{  
#ifdef ERR_CHECK  
    try {  
        start();  
        finish();  
    }  
    catch (BndErr & be) {  
        throw;  
    }  
    catch (RegionSizeErr & se) {  
        throw;  
    }  
    catch (RegionSubErr & sue) {  
        throw;  
    }  
}
```

```

#else
    start();
    finish();
#endif
}

```

Note that modulo error checking (to be described shortly) `execute()` consists entirely of two function calls. The `start()` function begins asynchronous execution of the communication pattern, i.e. it begins a non blocking Send and posts a nonblocking Receive. The `finish()` function waits for communication to complete. If the programmer wishes to try to overlap computation with communication these functions can be called directly. Since KeLP is currently built on top of MPI and most existing MPI implementations do not provide support for such overlap there is little advantage to this at present.

At this time the `Mover` does not provide a facility for global reduction operations. See section 8, **MP++** for information on how to accomplish these operations.

Exceptions raised: Three types of exceptions can be thrown by `Mover`: `BndErr`, `RegionSubErr`, and `RegionSizeErr`. `BndErr` is thrown when an invalid index of `XArray` is involved in data motion. This might happen, for instance, if the `Mover` is instantiated with the wrong `XArray` or `MotionPlan`. `RegSubErr` is thrown when the source(dest) `Region` is not a subregion of the source(dest) `Grid's Region`. `RegionSizeErr` is thrown when the source and destination `Regions` have different sizes. Since this involves a certain amount of overhead the programmer has the option of activating or deactivating exception handling at compile time.

Examples are shown in the sample code above. For further information please refer to `./kelp/MoverX.h.m4`, `./kelp/MoverX.C.m4`, `./array/BndError.h`, and `./tools/RegError.h`.

Implementation Details

When a `Mover` is created, its constructor retrieves the information stored in the `MotionPlan`. For each `Grid` this includes three lists, one each for incoming (from off processor) messages, outgoing (to off processor) messages, and local (on processor) data movement. We now examine the functions `start()` and `finish()` in some detail.

```

{
(1)  ProcessOut();
(2)  ProcessIn();
(3)  if (KelpConfigStatus(CONTIG_MSG_IN_PLACE))
(4)      MarkContig();
(5)  AllocateBuffers();
(6)  Post();
(7)  Send();
(8)  Local();
}

```

Figure 2.9.2a: `start` function of class `MoverX`.

```

{
(1)  Deliver();
(2)  FreeOutBufs();
}

```

Figure 2.9.2b: `finish` function of class `MoverX`

Both `start` and `finish` consist entirely of function calls. In `start`, the calls to `ProcessOut` (1) and `ProcessIn` (2) set up arrays of message descriptors. By default, line (3) evaluates to true and KeLP checks if the data elements in a message are contiguous in memory line (4). If they are contiguous they are flagged in the member variable `contig` to avoid the overhead of allocating message buffers and packing

and unpacking. Line (5), `AllocateBuffers` allocates message buffers for non-contiguous messages and returns a pointer to existing memory for contiguous messages, both outgoing and incoming. These buffers will be deallocated when `wait()` is called.

Note that lines (1-5) are executed every time the `Mover` executes. This avoids tying up memory for any longer than necessary at the cost of imposing a fixed start-up overhead for messages. This overhead can be significant for small (relative to number of processors) problems but is amortized away for reasonable large problem sizes. As an alternative one could create an additional constructor which preallocated the message buffers.

The `Post` function, line (6), posts a non-blocking receive for each expected incoming message. Line (7), `Send`, loops over outgoing messages, packing the non-contiguous messages into the buffers created in `AllocateBuffers` and posting non-blocking sends. Finally `Local`, line (8), performs the local memory to memory copies for elements assigned to the same processor.

In `finish`, the call to `Deliver` loops over the array of incoming messages, detects if they have arrived, unpacks those that were non-contiguous, assigns them to the appropriate elements of the local `Grid`, and deallocates the incoming message buffer. `FreeOutBufs` loops over outgoing messages, waiting until the send is completed and deallocating the outgoing message buffer.

2.9 class `VectorMover`

A `VectorMover` presents the exact same interface as a `Mover`. However, a `VectorMover` performs message aggregation where possible. Message aggregation will provide superior performance for applications with many fairly small messages on architectures with a high message start cost. A `VectorMover` throws the exact same exceptions as a `Mover`.

3 Looping iterators and macros

KeLP provides a number of iterators and macros for controlling loop iterations. They may prove useful in writing dimension independent code.

3.1 indexIteratorX and for_X

indexIterator1 helps simple iteration over the domain of Grid1, Array1, FloorPlanX, or XArrayX. For example:

```
for (indexIterator1 iter(D); iter; ++iter) {
    Point1 p = iter();    // current point
    int i = iter(0);      // 0th element of the current point
    Do something to D(p);
}
```

where D one of the classes listed above, is logically equivalent to

```
for (int i=D.lower(); i<= D.upper(); i++) {
    S1;
}
```

Similarly, if E is a Grid2 or Array2

```
for (indexIterator2 iter(E); iter; ++iter) {
//According to taste either do this...
    Point2 p = iter();    // current point
    Do something to E(p);
//...or do this.
    int i = iter(0);      // 0th element of the current point
    int j = iter(1);      // 1st element of the current point
    Do something to E(I,j);
}
```

is equivalent to

```
for (int j=E.lower(1); j <= E.upper(1); j++)
    for (int i=E.lower(0); i <= E.upper(0); i++) {
        S1;
    }
```

The constructs for three and four-dimensional objects follow naturally.

IndexIteratorX supersedes the for_X and for_point_X macros provided in the previous versions of KeLP for the same purpose.

3.2 nodeIterator

nodeIterator is used for parallel iteration over elements of an XArray or FloorPlan. This iterator is similar to indexIterator1, but each processor only executes iterations that it ``owns":

```
for (nodeIterator iter(X); iter; ++iter) {
    int i = iter();      // current index
    if X(i) lives on this processor do something with X(i)...
}
```

In other words, each processor loops over the Grids or Regions which it owns. Note that the “something” will often involve iterating over the local index space of $X(i)$ as in the next example.

```
int i;
  Point2 p;

  for (nodeIterator ni(grid); ni; ++ni) {
    i = ni();

    for (indexIterator2 ii(grid(i)); ii; ++ii) {
      p = ii();
      grid(i)(p) = 1.0;
    }
  }
```

`nodeIterator` supersedes the `for_all` loop used for the same purpose in the previous versions of KeLP.

Any synchronization not enforced by data dependencies must be explicitly programmed, with mechanisms such as `mpBarrier`.

4 Fortran Interface

KeLP provides several macros to ease the process of calling Fortran numeric code on KeLP data structures. All KeLP arrays are laid out in Fortran-style column-major order. See the tutorial for an example of how to use the following constructs.

The `FORTTRAN_NAME` macro helps with Fortran name-mangling and external "C" linkage. To link with a Fortran subroutine `foo`, use the following:

```
#define foo_link FORTTRAN_NAME(foo_, FOO, foo)
extern "C" {
void foo_link( args );
}
```

The `FortranRegionX` class is a temporary class that serves as an intermediary for passing a `RegionX` to Fortran. Given a `RegionX`,

```
FortranRegionX FR(R);
```

creates a `FortranRegionX` from a `RegionX`.

The `FORTTRAN_REGIONX` macro will pass the integer values of the `FortranRegionX` to Fortran, by reference. So, the following code passes the `Region2 R` to four integer arguments of a Fortran routine.

```
Region2 R;
FortranRegion2 FR(R);
fortran_routine(FORTTRAN_REGION2(FR));
```

The Fortran routine should accept the arguments as follows.

```
subroutine fortran_routine(R_lower0, R_lower1, R_upper0, R_upper1)
int R_lower0, R_lower1, R_upper0, R_upper1
```

The `FORTTRAN_ARGSX` macro provides shorthand for describing integer reference arguments. The following:

```
extern "C" {
void fortran_routine(FORTTRAN_ARGS2);
}
```

is expanded to:

```
extern "C" {
void fortran_routine(const int *, const int *, const int *,
                    const int *);
}
```

These constructs extend naturally to other dimensionalities.

5 Dock

Dock (Decomposition Classes for KeLP) is a small class library that provides HPF-style BLOCK decompositions for regular Grid structures. A `GhostPlan` is a `FloorPlan` which also keeps track of the number of ghost cells in each dimension. A `GhostIterator` is a simple class that helps iterate over the ghost regions of a partition. The `GhostIterator` often proves helpful in setting up boundary conditions. A `Processors` object is a first-class representation of a logical processor array. A `Decomposition` is a `FloorPlan` which will automatically conform to various regular block decompositions.

5.1 class `GhostPlanX`

Class `GhostPlanX` is a `FloorPlanX`, along with a `Point` that represents the number of ghost cells in each dimension for the `Regions` in the `GhostPlanX`.

A `GhostPlan` is publicly derived from `FloorPlanX`, so all public `FloorPlan` operations are valid for `GhostPlans`. Other useful operations are:

```
GP.ghost()          // returns the ghost Point for GhostPlan GP
GP.AddGhost(P);    // add ghost cells to a GhostPlan
GP.setghost(P);    // set the ghost cell vector
```

For more information, consult `kelp1.3/apps/dock/GhostPlanX.h.m4` and `kelp1.3/apps/dock/GhostPlanX.C.m4`

5.2 class `GhostIteratorX`

Class `GhostIteratorX` provides iteration over the ghost regions of a partition with ghost cells. A `GhostIteratorX` will return, one at a time, the `Regions` representing ghost cells for a partition. Suppose `G` is an `Grid`, and the programmer requires the outermost 2 cells of `G` in each dimension to be ghost cells. The following code fragment will zero out all ghost cells of `G`:

```
for (GhostIteratorX iter(G,2);iter;++iter) {
    const RegionX R = iter(); // R = next ghost region of G
    G.fill(0.0,R);
}
```

For more information, consult `kelp1.3/apps/dock/GhostIteratorX.h` and `kelp1.3/apps/dock/GhostIteratorX.C` in the KeLP sample application code distribution.

5.3 class `ProcessorsX`

A `ProcessorsX` object represents an array of virtual processors. A `ProcessorsX` object may be declared as follows:

```
ProcessorsX P(Region2(1,1,3,4)) // declare a 3x4 virtual processor
                                array
ProcessorsX P;                  // use the system default virtual
                                processor array
ProcessorsX P(STRIP);           // declare the processor array
                                with a linear topology
```

The system default processor array tries to make the processor array as square as possible. The last two forms listed above are defined such that the number of virtual processors equals the number of physical processors. This rule may be violated using the first form, in which case virtual processors are mapped to physical processors round robin.

For more information, consult `kelp1.3/apps/dock/ProcessorsX.h` and `kelp1.3/apps/dock/ProcessorsX.C` in the KeLP sample application code distribution.

5.4 class DecompositionX

A `DecompositionX` is a `GhostPlan` that will automatically adjust its shape to conform to several HPF-style `BLOCK` decompositions.

A `DecompositionX` will partition a single n -dimensional index space:

```
Decomposition2 D(M,N);           // declare Decomposition D to partition
                                the Region [(1,1),(M,N)]
```

The space is then partitioned onto a virtual processor array via the `distribute` member function. Each dimension of the index space may be partitioned according to one of two rules. Suppose that the global index space has N elements along some dimension and the virtual processors array has P processors along the corresponding dimension. The rules are :

- `BLOCK1` : each virtual processor gets exactly $\text{ceil}(N/P)$ elements, until there are no more elements.
- `BLOCK2` : the first $N \bmod P$ processors get $\text{ceil}(N/P)$ elements, and the rest get $\text{floor}(N/P)$ elements.

Some examples of the `distribute` member function are:

```
D.distribute(BLOCK2,BLOCK2,P); // distribute both axes using the
                                BLOCK2 rule over the virtual
                                processor array P
D.distribute(BLOCK1,BLOCK1,BLOCK1); // distribute all three axes
                                    using the BLOCK1 rule and the system
                                    default Processors object
```

In addition to the methods of class `FloorPlan` , which provide *local* `Region` and ownership information, `DecompositionX` objects have additional functions to query or modify the *global* computational domain and the mapping of that domain onto the virtual processor array. For more information, consult `/dock/DecompositionX.h.m4` and `/dock/DecompositionX.C.m4` in the KeLP sample application code distribution.

6 DGrid

The `dGrid` library provides three classes, `dGridX`, `replicatedGridX`, and `CollectiveGroup`, each of which is implemented in two and three dimensions.

6.1 class `dGridX`:

The `dGrid` class, derived from `XArray`, represents a distributed array of `Grids` that has a regular `BLOCK` decomposition, represented by the `Decomposition` class of the `dock` library. So a `dGrid` is built from a `Decomposition` in exactly the same way as a `XArray` is built from a `FloorPlan`.

```
DecompositionX D;  
dGridX< type > DG(D);2
```

Class `dGrid` is also provided with methods which give it access to all the domain and processor array inquiry functions of the `Decomposition` class. A typical example would be

```
int domainExtents(const int dim) const  
{    return _decomp.domainExtents(dim); }
```

The primary utility of `dGrid` is that it provides a number of built in communications operations to support global matrix operations and linear algebra.

```
DG.copyOnIntersection(argtype X)    //copy values from argtype X where  
                                   //the domains intersect.
```

This function is overloaded so that `argtype` may be any one of `XArray`, `dGrid`, or `replicatedGrid`. The remaining specialized copy functions take only `dGrid` arguments

```
DG.aliasPSection(DG2,R)    // return an alias to a set of blocks from DG2  
                           // that fall in Region R of DG's virtual  
                           // processor array
```

```
DG.copy(DG2, Rsrc, Rdest)    // copy values from Region Rsrc of DG to  
                             // Region Rdest of DG2.
```

An important distinction between these copy functions and the copy functions of the `MotionPlan` class is that these perform the actual data motion. For more information see `/dgrid/dGridX.h.m4` and `/dgrid/dGridX.C.m4`

6.2 class `replicatedGridX`:

The `replicatedGrid` class is a specialized `XArray` whose members (each an individual `Grid`) are replicated over a collection of nodes, i.e. each node contains an identical copy of the same `Grid`. The key member function, `replicate`, broadcasts the values of the `Grid` from a source to all instances of the replicated `Grid`. For more information see `/dgrid/replicatedGridX.h.m4` and `/dgrid/replicatedGridX.C.m4`

² On some platforms use of this constructor leads to a run-time error for as yet undetermined causes. The workaround is to declare an *empty* `dGrid` and then use the `instantiate` member function to give it the desired properties.

```
dGridX< type > DG;  
DG.instantiate(D);
```

6.3 **class CollectiveGroup**

The `CollectiveGroup` class provides a KeLP analog of a MPI Communicator; a subset of the global node set. The `CollectiveGroup` is useful for operations that need be performed on only a subset of the nodes, including reductions and broadcasts. For more information see `/dgrid/CollectiveGrid.h`.

In order to implement the `dGrid` library efficiently, in a few cases the code drops below the KeLP abstractions and directly manipulates MPI Communicator objects.

7 API-FD

API-FD is a small set of classes which provides a very simple Application Programmer Interface for finite difference calculations. It is hoped that this API illustrates the proper use of KeLP mechanisms and can serve as a starting point for your own application-specific facilities.

The API provides the following classes: `mGrid`, `IrregularGrid`, and `Adder`. An `mGrid` is a `Grid` of `double`, with some simple numerical operators built in. An `IrregularGrid` is an `XArray<mGrid>`, with additional member functions that carry out certain numerical and communication operations. An `Adder` is a `Mover` that automatically adds the source to the destination without intermediate buffer-packing.

7.1 class `mGrid`

Class `mGrid` is a `Grid` of `double`, with some special functions built in that are useful for finite-difference methods.

An `mGrid` is publicly derived from `GridX<double>`, so all public `Grid` operations are valid for `mGrids`. Other useful operations are:

```
m.fill(d,R);      // over Region R, fill mGrid m with value d
m.fill(d);        // fill all cells of m with value d
m.maxDelta(G,R)  // returns max(|m(p)-G(p)|) over all points p in
                 // Region R
```

For more information, consult `./API/mGrid.h` and `./API/mGrid.C` in the KeLP sample application code distribution.

7.2 class `IrregularGrid`

Class `IrregularGrid` implements a simple single-level block-irregular grid structure.

Creating an `IrregularGrid`:

```
IrregularGrid I;           // empty IrregularGrid
IrregularGrid I(F);       // instantiate with shape of FloorPlan F
IrregularGrid I(F,FALSE); // as above, but do not build FillPatch
                          // MotionPlan
IrregularGrid I(G);       // as above, but G is a GhostPlan and
                          // holds ghost cell width
```

An `IrregularGrid` is derived publicly from `XArray` of `mGrid`, and all `XArray` operations are valid for `IrregularGrids`. Some other operations on `IrregularGrids` are:

```

IG.instantiate(G);           // instantiate with shape and ghost
                             // regions of GhostPlan G
IG.BuildFGSched();         // Build the fillpatch MotionPlan
IG.ghost();                // returns ghost region vector (a Point)
IG.interior(i);            // interior (non-ghost) Region of IG(i)
IG.fill(d);                // fill all cells with value d
IG.assignGhost(d);         // assign all ghost cells to value d
IG.fillGhost();            // execute fillpatch operation
IG.randomize();            // fill all cells with random values
IG.CopyOnIntersection(IG2) // Copy all values from IrregularGrid
IG2                          // where the domains intersect
IG.AddOnIntersection(IG2) // Add all values from IG2 to IG, where
                             // where the domains intersect
IG.maxDelta(IG2);          // returns max(|IG(p)-IG2(p)|) over all
                             // points p where the domains intersect

```

For more information, consult `./API/IrregularGrid.h` and `./API/IrregularGrid.C` in the KeLP sample application code distribution.

7.3 class Adder

An `Adder` is a specialized `Mover`. Instead of copying values from a source `Region` to a destination `Region`, and `Adder` adds values from a source `Region` to a destination `Region`.

An `Adder` is declared and used just like an `Mover`:

```

Adder A(Send,Recv,S);      // declare an Adder bound to two XArrays
                             // and a MotionPlan
A.execute();               // perform data motion and addition

```

For more information, consult `./API/Adder.h` in the KeLP sample application code distribution.

8 MP++

MP++ is a portable message passing library which provides facilities for asynchronous and synchronous message passing, barrier synchronization, and global reductions.

The MP++ header file "mp++.h" must be included in every source file which calls an MP++ function, and the executable must be linked with the library `-lmp++`.

You may exit a MP++ program by calling `mpExit()` and returning from `main()`. Under abnormal circumstances, MP++ function `Abort()` should be called to abort the program, print out an error message, and dump core (if supported).

8.1 Miscellaneous Functions

mpAbort

Force termination and dump core, if supported.

Synopsis:

```
void mpAbort()
```

Abort

Exit with an error message and dump core.

Synopsis:

```
void Abort(char *who, char *why)
```

Description:

Exits the program with an error message formed by the string in `who`, which typically contains the name of the function calling it, and in `why`, which usually contains the reason. It also prints out the name of the file and line number from where `Abort()` was called. `Abort()` is actually a macro.

mpExit

Clean up and exit the message-passing system.

Synopsis:

```
void mpExit()
```

Description:

This function may be called more than once.

mpInitialize

Initialize the message passing library

Synopsis:

```
void mpInitialize(int argc, char **argv)
```

Description:

This function must be called to initialize the message-passing library before any MP++ function may be called. The arguments to `mpInitialize` should be the command-line arguments passed to `main`.

mpNodes

Return the number of nodes available.

Synopsis:

```
int mpNodes()
```

mpMyID

Return the local processor identifier number, an integer between zero and the number of nodes minus one.

Synopsis:

```
int mpMyID()
```

mpWallClockSeconds

Return the elapsed wall clock time in seconds for a processor

Synopsis:

```
double mpWallClockSeconds()
```

Description:

Returns the elapsed wall clock time in seconds for a processor. Time is referenced to the beginning of the program, i.e. it returns 0 at the beginning of the program.

8.2 Communication Routines

mpBarrier

Perform a barrier synchronization

Synopsis:

```
void mpBarrier()
```

mpBroadcastOK

Verify if send supports broadcast on this hardware architecture

Synopsis:

```
int mpBroadcastOK()
```

mpLastBytes

Return the number of bytes received by the last message

Synopsis:

```
int mpLastBytes()
```

mpLastType

Return the type of the last message received

Synopsis:

```
int mpLastType()
```

mpLastProcessor

Return the source processor of the last message received

Synopsis:

```
int mpLastProcessor()
```

mpSend

Send a message (possibly blocking).

Synopsis:

```
void mpSend(int type, void *data, int nbytes, int dest)
```

Description:

Synchronous (blocking) send, where: `type` is the type of the message, and can be any value between `MP_FIRST_AVAILABLE_TYPE` and `MP_LAST_AVAILABLE_TYPE` (defined in `arch.h`); `data` is a pointer to the data to be sent; `nbytes` is the number of bytes to be sent; `dest` is the identification number of the target node, or the constant `mpAllNodes` (also defined in `arch.h`) if `mpBroadcastOK()` is `TRUE`, meaning that the message must be broadcasted to all nodes. `mpSend()` returns when either the message has been sent, or when the message has been buffered in the OS. There is no guarantee that the message has arrived at the receiving processor. In case of an error, an exception will be thrown:

```

try {
    mpSend(type, data, count, dest);
}
catch (mpMsgSizeErr) {
    // error: negative message size!
}
catch (mpRangeErr) {
    // error: destination processor out of range!
}
catch (mpBcastErr) {
    // error: cannot broadcast on this architecture!
}
catch (mpBcastPtPErr) {
    // error: cannot broadcast with MPI point-to-point!
}
catch (mpSendErr) {
    // error: problem in MPI_Send()!
}

```

9.2.7 mpRecv

Receive a message (blocking).

Synopsis:

```
void mpRecv(int type, void *data, int nbytes)
```

Description:

Synchronous (blocking) receive, where: `type` is the type of the message to be received, and can be any value between `MP_FIRST_AVAILABLE_TYPE` and `MP_LAST_AVAILABLE_TYPE`, or the constant `mpAnyType` (defined in `arch.h`); `data` is a pointer to the area where the received data should be placed; `nbytes` is the size of the area pointed to by `data`. `mpRecv()` returns when the message is received. Note that the received message can be less than the buffer size. In case of an error, an exception will be thrown:

```

try {
    mpRecv(type, data, nbytes);
}
catch (mpMsgSizeErr) {
    // error: negative message size!
}
catch (mpRecvErr) {
    // error: problem in MPI_Recv()!
}
catch (mpCountErr) {
    // error: received message doesn't fit in the message buffer!
}

```

mpTest

Wait for a message (blocking).

Synopsis:

```
void mpTest(int type)
```

Description:

A synchronous routine that blocks until a message of type `type` is received. If `type` is equal to the constant `mpAnyType`, `mpTest()` will return after any message is received. `mpTest()` returns immediately if, when it is called, there is already a message of type `type` (or any message, if `type` is equal to `mpAnyType`) being held by the OS. If any error occurs an exception will be thrown:

```
try {
    mpTest(type);
}
catch (mpTestErr) {
    // error: something is wrong in MPI!
}
```

mpASend

Asynchronous (nonblocking) send of a message.

Synopsis:

```
mpMessageID *mpASend(int type, void *data, int nbytes, int dest)
```

Description:

Asynchronous (nonblocking) send, where: `type` is the type of the message, and can be any value between `MP_FIRST_AVAILABLE_TYPE` and `MP_LAST_AVAILABLE_TYPE` (defined in `arch.h`); `data` is a pointer to the data to be sent; `nbytes` is the number of bytes to be sent; `dest` is the identification number of the target node, or the constant `mpAllNodes` (also defined in `arch.h`) if `mpBroadcastOK()` is `TRUE`, meaning that the message must be broadcasted to all nodes. `mpASend()` immediately returns a pointer to a `mpMessageID` identifying the buffer pointed to by `data`. The data may not be sent immediately, though. The data stays in the user's buffer until it is sent, and the user should not modify the data until he is sure that the message send has completed. This can be checked by the routines `mpADone()` and `mpAwait()`. If any error occurs, the program will be aborted (e.g. if there is memory exhaustion), or an exception will be thrown:

```
try {
    mpASend(type, data, count, dest);
}
catch (mpMsgSizeErr) {
    // error: negative message size!
}
catch (mpRangeErr) {
    // error: destination processor out of range!
}
```

(cont.)

```

catch (mpBcastErr) {
    // error: cannot broadcast on this architecture!
}
catch (mpBcastPtPErr) {
    // error: cannot broadcast with MPI point-to-point!
}
catch (mpSendErr) {
    // error: problem in MPI_Isend()!
}

```

mpARecv

Asynchronous (nonblocking) receive of a message.

Synopsis:

```
mpMessageID *mpARecv(int type, void *data, int nbytes)
```

Description:

Asynchronous (nonblocking) receive, where: `type` is the type of the message to be received, and can be any value between `MP_FIRST_AVAILABLE_TYPE` and `MP_LAST_AVAILABLE_TYPE`, or the constant `mpAnyType` (defined in `arch.h`); `data` is a pointer to the area where the received data should be placed; `nbytes` is the size of the area pointed to by `data`. `mpRecv()` returns, immediately, a pointer to a `mpMessageID` (defined in `mp++.h`) that identifies the buffer pointed to by `data`. When the data arrives, it is placed in this buffer, and made available for the user, who can check this using `mpAwait()` or `mpADone()`. If any error occurs, the program will be aborted (e.g. if there is memory exhaustion), or an exception will be thrown:

```

try {
    mpRecv(type, data, nbytes);
}
catch (mpMsgSizeErr) {
    // error: negative message size!
}
catch (mpRecvErr) {
    // error: problem in MPI_Irecv()!
}

```

mpATest

Check if a message of a certain type has arrived.

Synopsis:

```
int mpATest (int type)
```

Description:

`mpTest()` is asynchronous, and returns 1 if a message of type `type` has been received (and is still being held by the OS), and 0 otherwise. If `type` is equal to the constant `mpAnyType` (defined in `arch.h`), `mpATest()` will return `TRUE` if any message has been received, and `FALSE` if no message has been received.

mpADone

Check if an asynchronous message is done.

Synopsis:

```
int mpADone(mpMessageID *id)
```

Description:

`mpADone()` is asynchronous, and returns `TRUE` if the message corresponding to the buffer identified by `id` has already been sent or received, and `FALSE` otherwise. If the message is done, then the `mpMessageID` will be invalid. If any error occurs an exception will be thrown:

```
try {
    mpADone(id);
}
catch (mpTestErr) {
    // error: problem in MPI!
}
```

mpAwait

Wait until an asynchronous message has finished.

Synopsis:

```
void mpAwait(mpMessageID *id)
```

Description:

`mpAwait` is synchronous, and will block until the message corresponding to the buffer identified by `id` is actually sent or received. If any error occurs an exception will be thrown:

```
try {
    mpAwait(id);
}
catch (mpWaitErr) {
    // error: problem in MPI!
}
```

8.3 Reduction Functions

mpReduce Φ

Perform a reduction, across the nodes, executing the operation indicated by Φ . For Add, Max, and Min, $\langle T \rangle$ can be any of `char`, `int`, `unsigned int`, `long int`, `float`, `double`, or `long double`. For the logical operations `Or`, `And`, and `Xor`, $\langle T \rangle$ can be either `char` or `int`.

Synopsis:

```
void mpReduceAdd(<T> *data, int length = 1 )
void mpReduceOr (<T> *data, int length = 1 )
void mpReduceAnd(<T> *data, int length = 1 )
void mpReduceXor(<T> *data, int length = 1 )
void mpReduceMax(<T> *data, int length = 1 )
void mpReduceMin(<T> *data, int length = 1 )
```

Description:

The `mpReduce Φ` routine performs a reduction across all nodes, executing the operation indicated by Φ as shown in the prototypes. It returns only after all the active nodes have called it. `data` is a pointer to the buffer that contains the data to be reduced, and `length` is the number of elements. `mpReduce Φ` will execute `length` reductions, one for each element of data, and place each result in its corresponding position in the buffer pointed to by `data`. If any error occurs the program will be aborted.

8.4 Broadcast Functions

mpBroadcast

Perform a broadcast from the root to all processors. `<T>` can be either `char`, `int`, `double`, or `float`.

Synopsis:

```
void mpBroadcast(<T> *data, const int length, const int root)
```

Description:

The `mpBroadcast` routine performs a broadcast from `root` to all nodes. `data` is a pointer to the buffer that contains the data to be broadcast, and `length` is the number of elements.

A. Appendix: The Patch Specification

```
/******  
*                                     PatchX.h  
*  
* Defines the abstract public interface to a user defined container  
* class to be used in a KeLP application.  
* Conformance to this interface is enforced by the internal use, by  
* KeLP, of concrete member functions defined here or in an  
* implementation of a concrete Patch class.  
  
* Note that this file illustrates the minimum requirements for a KeLP  
* 1.4 compliant Patch class. For example, an application Patch may be  
* templated on the data type it holds, but this is not required. A more  
* practical example is provided in a separate file, UserPatchX.h.  
  
* A PatchX is a generalization of an X-dimensional array.  
* It has an associated rectangular bounding box, called the Region, but  
* the data do not necessarily form a uniform rectangular array. In  
* particular, it need not be defined over all the points of its Region.  
* A Patch may have one or more indexable solution fields, with a  
* separate index domain which is a KeLP Region1  
*  
* A Patch is not required to support random access, though it must  
* support copying over a regular section (including fields, if present).  
* A concrete patch class might include random access capability, but  
* such capability isn't required for the correct operation of the other  
* KeLP * classes: FloorPlan, XArray, MotionPlan, and Mover  
*  
*  
* Programmer interface and implementation  
*  
* The implementer of a concrete Patch should use this file as a  
* template. Patch class implementations which conform to this  
* specification are said to comply with KeLP v. 1.4.  
  
* This file implements various member functions that query Patch  
* attributes that are common to all concrete instances.  
* It is strongly advised that an application Patch class not override  
* these concrete members, since the correct operation of KeLP depends  
* on them.  
  
* The simplest kind of Patch might be an ordinary rectangular array.  
* The KeLP distribution includes such a definition, Grid, which *  
* had previously been supplied as part of KeLP. This definition may be  
* used as a starting point for implementing new Patch classes  
*  
*  
* PatchX.h also specifies various member functions to implement the  
* following functionality  
*  
* construction
```

```

*           copying and serialization
*
* The specific member functions are
*           Patch()
*           SerializeIn ()
*           SerializeOut ()
*           LinearSize()
*           isPreallocatable()
*
* Some of these members are abstract while others specify
* certain actions that must take place.
*
* Since some users may choose not to use indexible Patch fields,
* alternative forms of the above functionality is provided, that elides
* fields specifiers

* Scott B. Baden, 9/14/99
* Modified Daniel Shalit 01/11/01
*****/
#ifndef _included_PatchX_h
#define _included_PatchX_h

typedef int SIZE_T;

class PatchX
{
private:
    dTYPE* _data           //dTYPE may be intrinsic or user defined.
    RegionX    _region;   // The region
    Region1    _fields;   // The range of the fields.
                                // Note the internal use of the
                                // Region1 specifier for fields.

public:
    //
    // Constructors
    //
    PatchX( ) {}

    PatchX(const RegionX& region, Region1 fields ) :
        _region(region),
        _fields(fields){}
    //
    // Destructors
    //
    virtual ~PatchX(){}

    //
    // Functions for querying various patch attributes.  These must be
    implemented
    // as shown since KeLP class methods depend them.
    //
    const Region1& fields() const { return _fields; }
    const int cs() const { return _fields.lower(0) ; }

```

```

const int ce() const { return _fields.upper(0) ; }
const int nFields() const { return fields().size(); }
    //
    // Bounds
    //
const RegionX& region() const { return _region; }
const PointX& lower() const { return _region.lower(); }
const PointX& upper() const { return _region.upper(); }
PointX extents() const { return _region.extents(); }
const int lower(const int k) const { return _region.lower(k); }
const int upper(const int k) const { return _region.upper(k); }
const int extents(const int k) const { return _region.extents(k); }

// This tells the user if the size of a serialized region of data
// can be computed with local information only

virtual const int isPreallocatable() const ;
//
// Functions to move data from/to PatchX

/*****
    virtual void Copy(const RegionX& Rd,  const PatchX& Ps, const
RegionX& Rs,
        const int csd=1, const int ced=1, const int css=1,
        const int ces = 1 )

    Copy from (Ps on Rs over fields css through ces) into
        (*this on Rd over fields csd through ced)

    Various quantities in the source and destination must conform:
        Rs must lie wholly within Ps's Region
        Rd must lie wholly within *this's Region
        (css:ces) must be valid field indices for Ps
        (csd:ced) must be valid field indices for Pd
        ced - csd == ces - css

    Note that Rs and Rd need not necessarily have the same number of points
    Such a constraint might be required of an ordinary array class, and
    properly belongs in a concrete class definition rather than in Patch
*****/
    virtual void Copy(const RegionX& Rd,  const PatchX& Ps, const
RegionX& Rs,
        const int csd=1, const int ced=1, const int css=1,
        const int ces = 1 ) ;

    // Returns the number of bytes required to store this Patch
restricted
    // to the Region Rs over fields Rc.lower(0) through Rc.upper(0)

    virtual const SIZE_T LinearSize(const RegionX& Rs, const Region1&
myFields)
        const ;

    // Pack the subset of Patch restricted to Region Rd and fields
(cs:ce)
    // into previously allocated contiguous storage

```

```
    virtual void SerializeIn(void *Lptr, const RegionX& Rd, const int cs,
const
    int ce) const ;

    // Unpack data previously Serialized in contiguous storage into an
    // existing patch

    virtual void SerializeOut(void *Lptr, const RegionX& Rs, const int
cs, const
    int ce) ;
};

#endif
```