KeLP User Guide
Version 1.4

Scott B. Baden *Richard 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

Abstract

KeLP is a domain specific library for efficiently implementing block structured parallel computations on distributed memory MIMD architectures. KeLP supports dynamic irregular collections of structured data organized into blocks that may be organized into levels, including multiblock and structured adaptive mesh hierarchies [2]. Applications may be written in a mixture of languages and developed on a single processor workstation. Such applications are portable across a diversity of parallel architectures including the IBM SP, Cray T3E, workstation clusters, and single processor workstations. KeLP applications may interoperate with numerical code that has been parallelized with OpenMP annotations, providing mixed mode programming on multi-tier SMP-based computers.

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, the KeLP User's Guide, v1.0 , and "The Data Mover: A Machine-Independent Abstraction for Managing Customized Data Motion," which was presented at the Twelfth International Workshop on Languages and Compilers for Parallel Computing (LCPC 99), La Jolla, CA, and appear 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 Introduction 1

2 Software Architecture 2

3 KeLP Classes 6

4 Getting Started with KeLP 11

5 Tutorial I 12

6 Tutorial II 19

7 Advanced features 27

8 Acknowledgments 30

9 References 30

1 Introduction

Building a finely tuned parallel scientific application from a high level mathematical description entails a massive expansion of detail that is unacceptable to many programmers. Non-uniform applications, such as multi-block and structured adaptive grid methods, present particular challenges due to the use of irregular computational structures which may be dynamic. Parallelism compounds the difficulty by introducing the need to manage data motion and load balancing, which generally cannot be handled by a compiler.

We have developed the KeLP Infrastructure to assist the scientific application programmer in managing the complexity of blocked representations at run time on high performance computers The current release of KeLP is the result of several years of research, and traces its roots to earlier versions of KeLP, and the LPAR-X system developed by Baden and Kohn [2].

KeLP is a domain-specific, run time middleware library intended for platforms ranging from multicomputer mainframes to clusters and single processor workstations. KeLP supports blocked representations only; it does not apply to fine-grained irregular computations involving general sparse matrices, rapid summation algorithms such as Barnes-Hut or the Fast Multipole Method, nor so-called “unstructured” finite element meshes. However, the class of applications that KeLP does apply to is significant and includes many applications that would be difficult to write with low level message passing libraries.

The design goals of KeLP are as follows.

- Enable all decisions about data decomposition, assignment of data to processors, and the communication of data dependencies to be deferred to run-time.
- Eliminate tedious bookkeeping details.
- Provide an appropriate level of abstraction that does not penalize performance
- Permit the reuse of heavily optimized serial numeric kernels.

KeLP, implemented as a `C++` class library, hides many of the details in managing complicated dynamic data structures. For example, KeLP hides all calls to the underlying interprocessor communication substrate, e.g. MPI. Rather, it enables the user to express data dependencies in terms of higher-level geometric set operations. The KeLP run time system interprets the required data motion carrying out any installation dependent optimization transparently. Our users have found that KeLP’s high-level, machine-independent interface reduces software development costs significantly. These savings come without sacrificing performance. In direct comparisons with hand-coded MPI applications KeLP performs at a comparable level with MPI, sometimes exceeding MPI’s performance [4,5]. KeLP applications may often reuse existing single-processor numerical kernels with modest changes, and may invoke extrinsic procedures written in languages such as Fortran. Such kernels may be parallelized with OpenMP or pthreads to admit mixed mode parallel execution on Multi-tier architectures.

KeLP1.4 requires only basic message passing (MPI) support and is currently running on the IBM SP2 and SP3, Cray T3E and clusters of workstations running Solaris and Linux.

1.1 Abstract KeLP

With the present release we introduce a new level of flexibility into KeLP. It permits the user to define their own notions of storage layout, linearization, and data copying between containers, while retaining the ability to express data motion and decomposition in terms of geometric sets in Cartesian space. Thus, it is possible to define a container of irregularly distributed particles over a region of space, and to transmit data laying in selected sub regions of data between containers living on different processors. There may not be a literal grid that holds the particles; the container might employ a tree or a hash table to organize the data. These details are encapsulated in member functions that the user must define, which are part of the contract between the KeLP run time system and the user. To support this capability, KeLP version 1.4 defines the

notion of an abstract container called a `Patch`. Further discussion of Patch is deferred to section 7 **Advanced Features.**

2 Software Architecture

Before we introduce the KeLP abstractions, we give a brief overview of the KeLP philosophy and we discuss the KeLP methodology for application software design.

2.1 The KeLP Model

KeLP supports coarse-grained data parallelism arising in distributed collections of structured blocks of data. It provides the illusion of a single global index space and a single logical thread of control. On an MIMD parallel computer the KeLP run-time system will execute in Single Program Multiple Data (SPMD) mode. On a single-processor workstation, the KeLP system will spawn parallel processes to mimic parallel execution, allowing development of parallel applications on a serial workstation.

KeLP supports dynamic collections of block-structured objects. Each block of a distributed object is an indivisible data structure assigned to a single physical processing node[1]. Blocks communicate via high-level block copy operations; intuitive geometric operations are used in lieu of indexing to determine which data are communicated. The KeLP class interfaces hide the low-level details of communication from the user.

How the blocks of data are organized is left up to the user's imagination. For example, in the multiblock application depicted in Fig. 1a, an irregular region of space —in this case Lake Superior—is covered with rectangles of various sizes, and the equation is solved on this irregular region. In other cases (see Fig. 1b), multi-resolution data structures adaptively concentrate computational effort in certain interesting regions of the solution— in this case in the vicinity of atoms making up a molecule— that may change with time. In still other cases, the processing nodes may run at different speeds, for example, in a heterogeneous computing cluster.

Such applications are difficult to parallelize because they rely on dynamic, irregular data structures. The location, size, and shape of the subdivided work is generally not known at compile time, and traditional static decompositions will fail to balance the work across the nodes, or introduce unacceptable overheads. Decisions about data decomposition, assignment to processors, and the determination of data dependencies must be made at run-time

KeLP provides tools to assist the user in managing such distributed data structures. Notably, KeLP does not support automatic data decomposition[2]. This is an appropriate level of abstraction for middleware like KeLP: data partitioning and mapping is highly application-dependent in treating irregular problems. Users who build their own libraries can subsequently reuse the code, or publish it for the benefit of other KeLP users. KeLP provides a standard interface for implementing such partitioners, greatly enhancing reusability.

Two important aspects of KeLP are its provision for *run time meta-data* and the notion of *communication orchestration*. Meta-data capture some aspect of execution, be it data layout, storage allocation, processor assignment, or communication. There is no point-to-point communication as in MPI per se. Underlying all KeLP applications is a single global coordinate system, a rectangular subset of Z^d, also called the *Lattice*. Communication is restricted to rectangular subsets of the lattice and is expressed in terms of global patterns that reside within it. The sources and destinations within these patterns refer blocks of data rather than processors. The mapping of blocks to processors isn't stated, and the overall communication pattern is encoded as a data dependence descriptor with the help of geometric set operations. Once this descriptor has been formed, then KeLP may be invoked to satisfy the data dependencies. The process is similar to

[1] This includes SMP nodes with multiple processors.
[2] The DOCK library, which is supplied with the KeLP distribution, supports HPF BLOCK-style decompositions.

constructing a persistent communication object in MPI, but at a much higher level of detail, since data dependences are expressed in geometric terms.

Under communication orchestration, an entire pattern of communication is logically executed as an atomic operation. The KeLP run time system may perform retrospective analysis over the entire global communication pattern involving all the processors. Thus, the KeLP programmer does not specify exactly how many messages are to be sent, their ordering, nor the specific processors involved. The fact that messages might be broken up in a certain way, or sent in a certain order, is not known into the programmer, and is under control of the run time system. This capability is the key to KeLP expressiveness and its ability to achieve high performance under a variety of conditions. KeLP has also been used to optimize regularly decomposed dense numerical linear algebra.

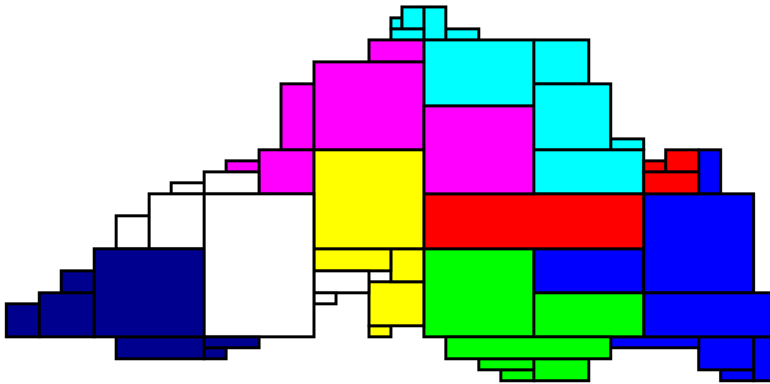


Figure 1a: A multiblock decomposition of Lake Superior (Courtesy of Yingxin Pang).

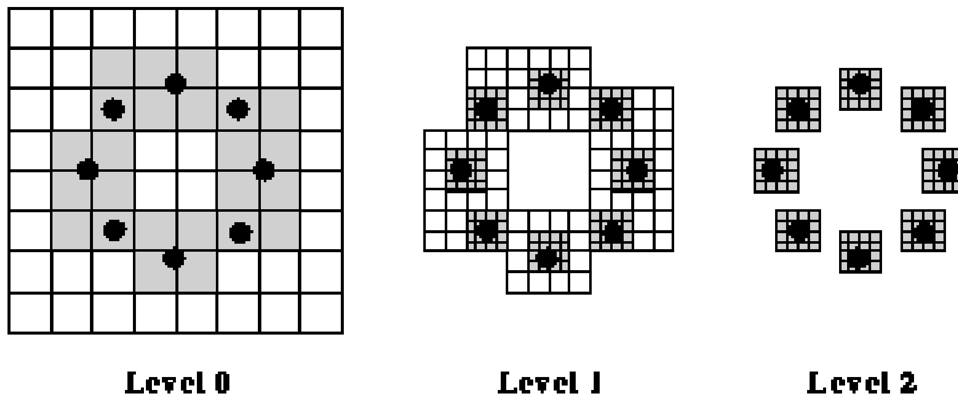


Figure 1b: Three levels of a structured adaptive mesh hierarchy. The eight dark circles represent regions of high error, such as atomic nuclei in materials design applications. The mesh spacing of each level is half of the previous coarser level (Courtesy of Scott Kohn.)

<table>
<tr><th colspan="3">Meta-Data Abstractions</th></tr>
<tr><th>Name</th><th>Definition</th><th>Interpretation</th></tr>
<tr><td>Point</td><td>< int i_0, int i_1, ... int i_{D-1} ></td><td>A point in Z^D</td></tr>
<tr><td>Map</td><td>f:{0, …, k-1} → Z</td><td>An integer mapping</td></tr>
<tr><td>Region</td><td>< Point *l*, Point *h*></td><td>A rectangular subset of Z^D</td></tr>
<tr><td>FloorPlan</td><td>Array of < Region, Map ></td><td>A set of Regions,
each with an integer owner</td></tr>
<tr><td>MotionPlan</td><td>List of < Region R_s, integer *i*, Region R_d, integer *j* ></td><td>A block-structured communication pattern between two FloorPlans</td></tr>
<tr><th colspan="3">Instantiators</th></tr>
<tr><th>Name</th><th colspan="2">Description</th></tr>
<tr><td>Grid</td><td colspan="2">A multidimensional array whose index space is a Region and set of solution fields</td></tr>
<tr><td>XArray</td><td colspan="2">An array of Grids; structure represented by a FloorPlan</td></tr>
<tr><td>Mover</td><td colspan="2">An object which executes the data motion pattern described by a MotionPlan as an atomic operation.
A Mover binds a MotionPlan to two XArrays, which are the source and sink for the data motion.</td></tr>
</table>

Table 1: A brief synopsis of the KeLP instantiator and meta-data abstractions.

2.2 DSL Construction

The purpose of KeLP is to curb the growth of detail in irregular applications. One way of achieving this goal is to implement a domain specific library (DSL) providing common facilities for a particular problem class, with a set of operations that map closely to the mathematics and physics of the problem. We encourage KeLP application programmers to view KeLP in this spirit; that is, as a tool for building DSLs. The KeLP distribution includes a very simple DSL, called *DOCK*, which provides facilities we find useful for solving elliptic differential equations with a finite-difference method. We invite our users to refer to this sample DSL as a starting point for writing their own DSL.

Because the DSL is written in KeLP, programmer has the full expressive power of KeLP at his or her disposal. Thus, the DSL user can “drop down” to the KeLP level in order to perform application-specific optimizations without having to work at the lower level communication layer, e.g. MPI. This approach has the advantage not only of hiding incidental detail, but it also enables heuristic knowledge captured at the application level to be transmitted via KeLP down to the hardware. Indeed, application-specific optimizations might otherwise be obscured.

The DSL and application code are typically written in two levels:

- top level KeLP code that manages the data structures and the parallelism;
- lower level code written in `C++`, `C`, or, most frequently, Fortran to handle numerical computation.

KeLP code may be freely mixed with calls to other `C++`, `C`, or Fortran functions. This particular style exploits the respective strengths of both `C++` and languages like Fortran; heavily optimized serial numerical kernels need not be massively reprogrammed in the course of parallelizing an application. The architecture of a KeLP application is shown in Fig. 2.

KeLP provides a set of *coordination abstractions* that provides “glue” between the underlying communication layer and the DSL. The current implementation of KeLP relies on MPI for message passing support, though other communication interfaces could be employed.

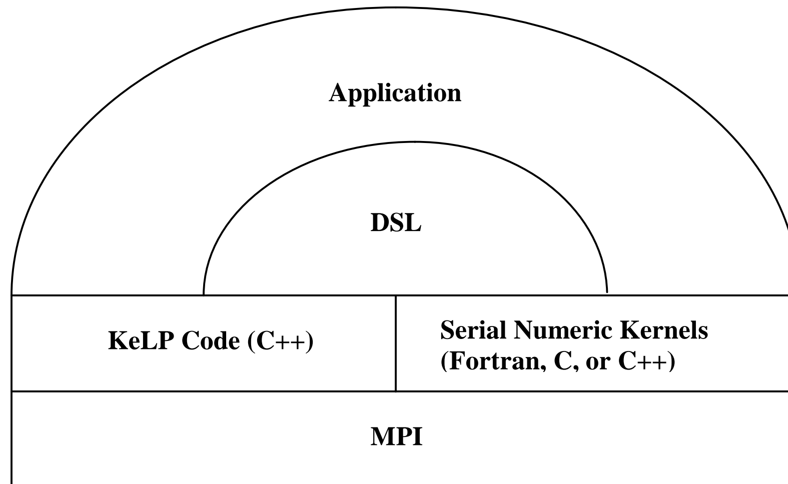


Figure 2: Architecture of a KeLP application

3 KeLP Classes

This section presents descriptive material about the four fundamental KeLP classes. Complete KeLP applications may be built using only these classes or they may form the basis for user developed DSL. The reader is referred to the tutorials later in this document for concrete examples of the use of these classes, and to the **Reference Manual** for implementation details.

3.1 Data Decomposition

KeLP provides three base classes to help manage irregular block-structured data decompositions at run time.

- `Region:` an abstract object representing an index space, a rectangular subset of Z^n.
- `FloorPlan:` an array of `Regions` with processor assignments, representing a blocked data decomposition.
- `XArray:` a dynamic array of aggregate data objects, such as a `Grid`, distributed over processors.

In addition, a utility class is provided.

- `Grid:` a uniform dynamic array instantiated over a `Region` and a set of solution fields.

In earlier versions of KeLP `Grid` was a fundamental class. With the introduction of the `Patch` concept, `Grid` has been moved to a separate library. It is linked into the KeLP build by default. This provides a simple default container class as well as back compatibility with existing KeLP applications. Users are free, however, to build their own `Patch` classes. The following discussion applies to more general `Patches` as well as to `Grids`.

The `Region` is useful for array section manipulation; it defines no data values or storage, only the indices themselves. In KeLP we may define a two dimensional region in Z^2 as follows:

```
Region2 r2(2,4,5,8)
```

The object `r2` is the bounding box over which the `Grid's` storage is to be defined. For example, the `C++` array `int X[M][N]` is an M x N array of integers, defined over the `Region (0,0,M-1,N-1)`, and the Fortran array `integer X(m0:m1,n0:n1)` is an `(m1-m0+1) x (n1-n0+1)` array of integers, defined over the `Region (m0,n0,m1,n1)`. A `Grid` defined on `r2`

```
MyGrid(r2)
```

is identical to the Fortran-90 array section

```
MyGrid(2:5,4:8)
```

A `Grid` is defined over a `Region` and may be manipulated with high-level block copy operations. `Grid` elements must all have the same type; allowable types include the `C++` built-in types `int, float, double, char`, `unsigned`, `long int, long double`, or any fixed-size user-defined type or class.[3] A `Grid` is not distributed; it exists in the address space of only one processor. Each `Grid` remembers its associated `Region`, which can be queried at run time, a convenience that greatly reduces bookkeeping for dynamically defined grids.[4]

In addition, a `Grid` can take an optional `Region1` argument specifying the number of solution fields, e. g. temperature and pressure:

```
Region1 f(1,2)
```

[3] KeLP currently does not permit arbitrary base types, i.e. user-defined classes with embedded pointers.

[4] Compare this with C, which requires that the programmer keep track of the bounds for a dynamic multidimensional array.

```
MyGrid(r2) -→   MyGrid(r2, f)
```

The `FloorPlan` defines a table based-data distribution. It is a table of `Regions` and their processor assignment. A `FloorPlan` may be constructed incrementally by the programmer or returned from a procedure that generates a data decomposition. The `FloorPlan` gives the programmer a common representation for representing an arbitrary irregular block-structured data decomposition.

The final class is the `XArray`, an array of `Grids` distributed across processors. The elements of an `XArray` must all be the same type; each `Grid` may have a different origin and size, but all must have the same number of dimensions and element type. Each `Grid` is assigned to exactly one processor. The KeLP system supplies a default mapping of `Grids` to processors, or the user may specify his or her own.

The programmer may describe arbitrary block distributions using the above classes. Typically, if there are *P* processors, the programmer declares a `FloorPlan` $P' \geq P$ `Regions` that represent the partitions. A subsequent call to a KeLP library routine instantiates a *P*' element `XArray` over the *P*' regions, allocating storage to processors according to the FloorPlan. The `XArray` serves as a container for these distributed `Grids`, which may be manipulated in parallel. The relationship between the `FloorPlan` and `XArray` are shown in Fig. 3a.

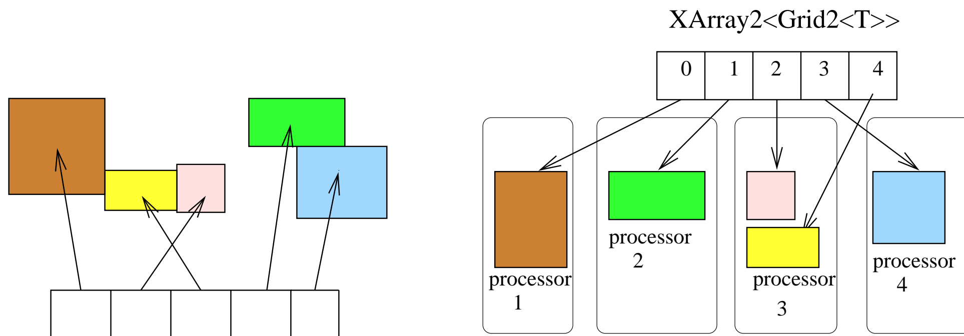


Figure 3a: **Example of a `FloorPlan` (left) and `XArray` (right). The arrows on the right denote processor assignments, not pointers. Indexing an `XArray` yields a `Grid`, not a pointer to a `Grid`.**

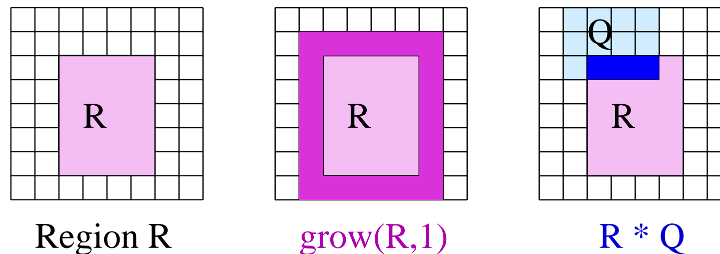


Figure 3b: **Examples of Region calculus operations grow and intersection *. `Q` and `R` are regions. `Grow(R,1)` expands the `Region R` by one unit in all directions, and is commonly used to grow a halo region for stencil computations. We can intersect this halo region with the other subdomain to determine data dependencies requiring communication**

3.1.1 The Region Calculus

To simplify `Region` manipulation, KeLP supports a *Region calculus.* The Region calculus provides geometric operators over `Regions` such as *grow* and *intersect* to help the programmer coordinate data motion among block-structured objects (see Fig. 3b) For example, a programmer can use `grow()` to define a ghost region for a finite difference calculation (see Fig. 3c). KeLP's data motion facilities may then be used to perform the data updates (see Fig. 3c). Data values will be copied over in the intersection of the ghost region and interacting blocks. KeLP hides all the bookkeeping, e.g. subscript calculations and manages any accompanying interprocessor communication.

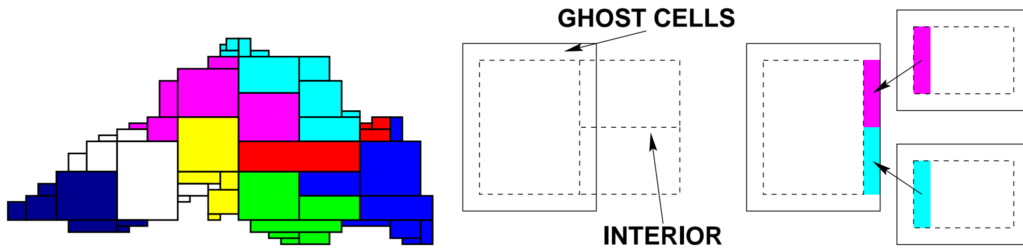


Figure 3c: An irregular multiblock grid (left). Ghost cells surround each grid (center). The copy operation shown in the right half of the figure updates the ghost cells. It is stored in a `MotionPlan` and carried out with a `Mover`.

3.2 Communication Orchestration

To realize efficient communication on message-passing architectures, KeLP provides data orchestration mechanisms to manage communication. These mechanisms are based on the inspector/executor paradigm [1]. They enable the programmer to pre-process the communication pattern at run-time, reducing synchronization requirements and realizing communication optimizations.

KeLP's provides two new data types to handle orchestration: `MotionPlan` and `Mover`. The `MotionPlan` encodes a set of block copy between two XArrays or among the elements of a single XArray (See Fig. 3d). This communication is treated as an atomic collective operation. The KeLP programmer typically builds a `MotionPlan` incrementally, adding block copy operations to the `MotionPlan` one at a time. The programmer can use KeLP's Region calculus and `FloorPlan` objects to write simple code that generates arbitrary collective block communication patterns. Alternatively, the MotionPlan generation code can be hidden in an application library or DSL.

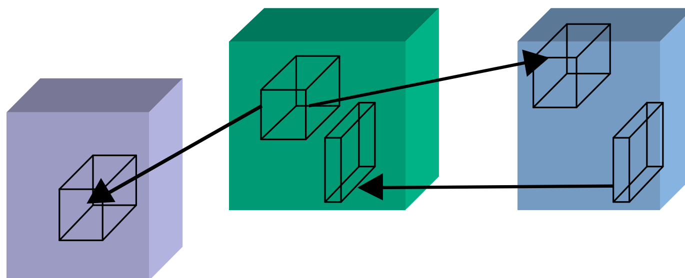


Figure 3d: Endogenous communication within a 3 element XArray. Exogenous communication between two XArrays is also possible, but isn't shown.

The `Mover` is a first-class object that performs the collective communication pattern represented by a `MotionPlan`. It performs block transfers over regular array sections between two XArrays. The advanced user may also tailor the `Mover` class to perform special optimizations or to associate numeric operations with a communication pattern, e.g. multigrid restriction and prolongation. This is described in section 7.4.

There are three steps to managing communication under KeLP's data orchestration model: 1) Build a `MotionPlan` describing the collective data dependence pattern, 2) Construct a `Mover` object by associating the MotionPlan with the XArrays to be moved, and 3) Carry out the communication to satisfy the dependencies by invoking Mover member functions.

To understand the operation of the MotionPlan and Mover, we use the following notation. Let D and S be two XArrays, $D(i)$ be element i of D, and $S(j)$ element j of S ($D(i)$ and $S(j)$ are Grids). Let R_s and R_d be two Regions. Then, the notation

$$(D(i) \text{ on } R_d) \Leftarrow (S(j) \text{ on } R_s)$$

denotes a dependence between two rectangular sections of XArray elements $D(i)$ and $S(j)$. Informally, we may think of this dependence as being met by executing the following block copy operation: copy the values from $S(j)$, over all the indices in R_s into $D(i)$, over the indices in R_d. We assume that R_d and R_s contain the same number of points, though their shapes may be different. The copy operation visits the points in R_d and R_s in a consistent systematic way, e.g., row-major order.

The MotionPlan encodes a set of dependencies of the above form. The MotionPlan M is a list containing n entries, denoted $M(k)$ for $1 \le k \le n$. Each entry is a 4-tuple of the following form:

$$\langle \text{Region } R_s, \text{ integer } i, \text{ Region } R_d, \text{ integer } j \rangle$$

The components of the 4-tuple describe dependence (and hence communication) in a manner consistent with the previous discussion. We use the `copy()` member function to augment the MotionPlan with new entries. An example of MotionPlan construction appears in Fig. 3e.

We instantiate a Mover object Mv by associating a MotionPlan M with two XArrays:

```
Mover Mv = Mover(M, S, D)
```

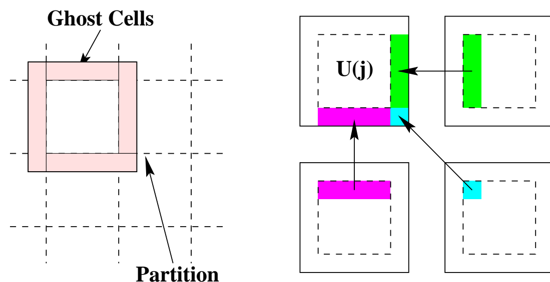
The result is an object `Mv` with two operations, `start()` and `wait()`. Informally, data motion commences with a call to the `start()` member function. This call is asynchronous and returns immediately. The `wait()` member function is used to detect when all communication has completed. In particular, when `wait()` returns, the Mover will have executed the following communication operation for each $M(k)$:

```
(D(M(k) # i)   on   M(k) # Rd ) ⇐  (S(M(k) # j)  on   M(k) # Rs),
```

where we select the entries of the `M(k)` 4-tuple with the # qualifier: `M(k) # Rd` , `M(k) # i`, and so on. Since the specific order of transfers is not defined, correctness must not depend on that ordering.

We build a Mover for each data motion pattern to be executed. In cases where source and destination are the same XArray, we have an endogenous copy, which is useful in halo updates.

```
FloorPlan2 U;
int NGHOST;
MotionPlan M;
for each U(i) ∈ U
   I = grow( U(i),-NGHOST );
   for each U(j) ∈ U, j ≠ i
     Region2 R = I ∩ U(j)
```



```
      M.Copy( U(i)∩ R,U(j)∩ R)
   end for
end for
```

Figure 3e: MotionPlan construction for a halo update (left). The grow( ) operation trims off the ghost cells (right), which are NGHOST cells deep. A graphical depiction of MotionPlan construction appears on the right.

3.3 Coarse-Grain Data Parallelism

KeLP defines a coarse-grained iterator—`nodeIterator`—that facilitates concurrent iteration over the `Grids` of an `XArray`. Each iteration of a `nodeIterator` loop executes independently in SPMD mode. Since an `XArray` is intended to treat irregular decompositions, each processor is may be assigned more than on XArray element.

KeLP treats the `nodeIterator` loop as an atomic operation and the result of executing such a loop is independent of the order in which the iterations are taken. The semantics of `nodeIterator` loop are similar to HPF's `INDEPENDENT forall` statement in that changes propagate to succeeding statements of a multiple statement `nodeIterator` loop. KeLP does not support nested parallelism; thus, `nodeIterator` loops may not be nested.

A typical KeLP program follows a simple model of coarse-grained parallel computation:

1. Generate the `FloorPlan`; (a) decompose the computational space, storing the Regions in a `FloorPlan`; (b) specify an assignment of each `Region` in (1a) to a processor.

2. Create an `XArray` of Grid corresponding to the `FloorPlan` of (1).

3. Generate a `MotionPlan` corresponding to the data dependencies between `Grids` in the XArray and instantiate a Mover.

4. Invoke a `Mover` to carry out the required communication.

5. Perform calculations on the Grids in the `XArray` in parallel using `nodeIterator` loop.

The decomposition in (1a) may be managed explicitly by the application, such as in the generation of refinement regions, or through partitioning routines such as recursive bisection [3] or uniform block partitioning. Because KeLP provides a uniform interface for representing and parallelizing irregular block decompositions, it provides flexibility and generality in treating a number of different problem classes.

The assignment of `Regions` to processors in (1b) provides the application flexibility in delegating work to processors. In general, this information will be returned by the routine that renders the partitions in (1a). This step may be omitted, in which case KeLP generates a default assignment.

Using the `FloorPlan`(1) we instantiate an `XArray` (2) corresponding to the data decomposition. KeLP creates `Grids` based on the supplied `Region` information and assigns them to the appropriate processors. After the decomposition and allocation of data, applications typically alternate between steps (4) and (5). After communication completes in (4), we compute in parallel on the `Grids` in the `XArray` using `nodeIterator` loop (5). In this step, we assume that the `Grids` are decoupled: they are processed independently and the computation proceeds asynchronously.

4 Getting Started with KeLP

If KeLP is not installed on your computer, you may download it from

`http://www-cse.ucsd.edu/groups/hpcl/scg/kelp/`.

Follow the installation instructions in the `README` file located in the top-level directory. KeLP requires that MPI be installed.

Once KeLP is installed on your system, set the shell environment variable `KELP_HOME` to the path of the installation directory. For example, if KeLP has been installed in `/usr/local/kelp` and you are using the C shell (`csh`) in the Unix environment, then execute the command

```
setenv KELP_HOME /usr/local/kelp
```

The `KELP_HOME` environment variable is used by the KeLP `makefile` system. To verify that it is set correctly, try copying one of the example directories to your working area and (re)compiling the code. For example, in the Unix C shell environment try:

```
cp -r $KELP_HOME/examples/region3_init
cd ~/region3_init
gmake
```

Notice that KeLP requires the use of `gmake` instead of `make` on most systems. `Gmake` is a popular package with system administrators and might already be installed on your system. It can be obtained at http://www.gnu.org/software/make/make.html.

Note also that on some systems (e.g. PCs running Linux), `gmake` *is* the default `make` and one must choose between these options at the top of the KeLP master Makefile:

```
#MAKE = gmake
MAKE = make
```

5 Tutorial I

For this tutorial we will examine three programs that progressively demonstrate the fundamental class components of the KeLP library. The source code for these programs can be viewed online at `http://www.cse.ucsd.edu/groups/hpcl/scg/kelp/KeLP1.3_src/examples/`, or in your installed KeLP software base in `$KELP_HOME/examples/`.

5.1 KeLP initialization

As shown in Fig. 2a, all KeLP programs currently depend upon MPI. To use the KeLP libraries in your program, first initialize MPI[5]:

```
MPI_Init(&argc,&argv);
```

Next, initialize the KeLP libraries:

```
InitKeLP (argc, argv);
```

During initialization, KeLP initializes its own message passing library `mp++` that hides the MPI implementation details. If your program needs to determine which processor is currently executing, or the total number of processors in the KeLP runtime environment, execute the calls:

```
int iproc = mpMyID() ;
int nproc = mpNodes() ;
```

to store the desired values in `iproc` and `nproc`, or the variable names of your choice.

5.2 region3_init : Region, FloorPlan, and XArray

As previously discussed in section 3, KeLP uses `Regions` and `FloorPlans` to describe how the coordinate space of a problem domain will be partitioned across parallel processing elements. In the case of a matrix, a `Region` corresponds to the bounds of a 2D array that encompasses all or a part of the matrix. The `FloorPlan` describes to the KeLP libraries how the `Regions` will be mapped across a processor space; i.e., all or some of the processes initiated with `MPI_Init`. Once a `FloorPlan` has been instantiated, an `XArray` may be created to store actual data values associated with the `Regions` distributed across the processor space.

Program `region3_init.C` demonstrates the instantiation of these objects for a simple 3D volume of size 8 x 8 x 8. The volume will be partitioned along the x-axis into 4 equal sized `Regions`.

After initializing KeLP, the program begins by defining the dimension of the problem coordinate space and number of regions in the macros Nx, `Ny`, `Nz`, and `NREGIONS`. The indexing variables `p` and `r` are also defined:

```
#define Nx (8)
#define Ny (8)
#define Nz (8)

// Number of Regions, must divide Nx (decomposition is along x).
#define NREGIONS (4)
```

---

[5] It might be argued that it would be cleaner and more logical to have `InitKeLP`, rather than the user program, call `MPI_Init.` However, by keeping the `MPI_Init` call outside of KeLP, we facilitate interoperation with applications that use a mix of KeLP and other libraries that call MPI.

```
int p ;
int r ;
```

The Regions `R[]` and `FloorPlan F` are then instantiated and the actual coordinate bounds of the regions and placement in `F` are declared. Note that Region `r` has origin `(1+(r*(Nx/NREGIONS)), 1, 1)` and extent `((r+1)*(Nx/NREGIONS), Ny, Nz)`

```
// Regions are coordinates of volumes.
Region3 R[NREGIONS] ;

// Declare FloorPlan to contain the regions.
FloorPlan3 F;
F.resize(NREGIONS);

// Declare our data decompositions and put them in the container.
for (r=0; r<NREGIONS; r++)
{
   R[r]=Region3(1+(r*(Nx/NREGIONS)),1,1,(r+1)*(Nx/NREGIONS),Ny,Nz)[6]
;
   F.setregion(r,R[r]);
}
```

After the regions have been placed in the `FloorPlan`, the processor assignments can be made. In this case we employ a BLOCK CYCLIC decomposition in which element `r` is assigned to processor `r` MOD `nproc` (here `nproc == 2`) are illustrated in Fig. 5a.

```
// Map the regions cyclically to processors until all regions are
// mapped.  Mapping is known on each processor
p = 0 ;
r = 0 ;
while (r < NREGIONS)
{
   F.setowner(r,p);
   r++ ;
   p++ ;
   p %= nproc ;
}
```

[6] The reader is reminded that the index *arguments* to a RegionX object correspond to X-dimensional array indices over that region sing Fortran column major ordering. This was described in detail in Section 3.1. This convention was adopted for convenience since most KeLP applications ultimately call Fortran numerical kernels.

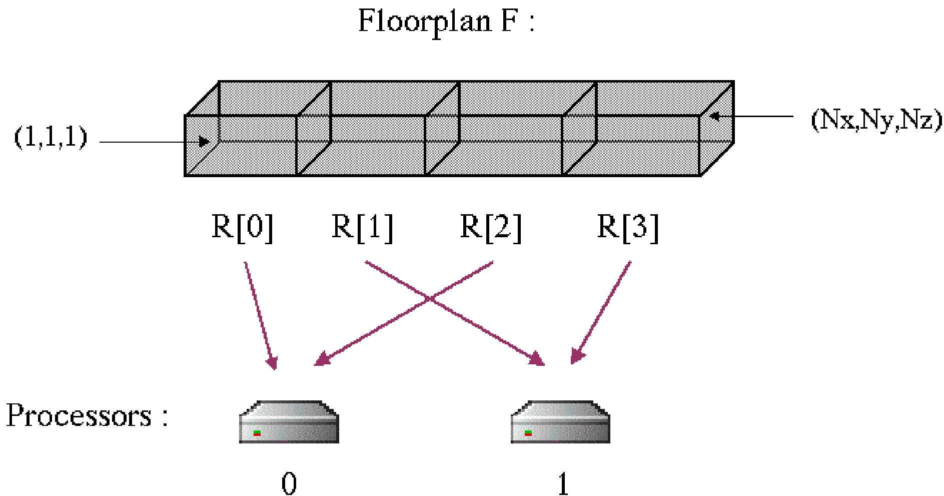


Figure 5a: Results of cyclic mapping of Regions `r = 0` to 3 to processors `p = 0` and `1`.

So far we have only created the description of a data structure. In order to actually store values at these locations we must allocate storage by creating an XArray from FloorPlan F:

```
// Create distributed array of 3D grids from the FloorPlan.
// Allocates storage.
XArray3<Grid3<double> > X(F);
```

Let us review what has been done up to now. The FloorPlan is a 1-Dimensional table, each entry consisting of a Region, i.e. a bounding box, and an owner, i.e. a processor ID. The XArray3 is a 1-Dimensional array each of whose elements is a Grid3, i.e. a 3-Dimensional array of, in this case, doubles.

This is also an appropriate point at which to note a KeLP indexing convention: KeLP objects Region, FloorPlan, and XArray are indexed in Fortran style, using parentheses. If Y is an indexible KeLP object, then Y(r) is the rth element of the `Y` object. If `Z` is an array of some KeLP class, then `Z[r]` is the rth element of `Z`.

The `XArray` can then be filled with values as desired. In this example, a constant value set by `initv` is used. The `nodeIterator` object `proc` is used to iterate through the `Regions` of the elements of `X`. Associated with each successive value of `proc` is a rank, which may be queried using the `( )` operator. We may use this rank to index the XArray `X` as shown below in the expression `X(r)`.

Each iteration of the `nodeIterator` may be executed in an arbitrary order. For many-to-one assignments of XArray elements to processors, there is an implied serial ordering on each processor, but the programmer may not take advantage of that ordering. Statements within the loop body of the nodeIterator are executed serially.

Within the loop body of the nodeIterator, we execute a serial `indexIterator` loop to sweep over all the 3-tuples corresponding to the points contained in the `Region of X(r)`. We create an `indexIterator3` object `ijk`. By an abuse of notation, we may construct an `indexIterator` using a Grid-valued object (recalling that we get a grid when we index the XArray X) as an argument to the constructor. The indexing operations on `ijk()` produce the `i`, `j`, and `k`, indices, respectively, corresponding to the three coordinate indices for each point. (Iterators are described in detail in Section 3 of the **KeLP Reference Manual).**

```
// Fill the data arrays with a constant value
double initv = 1.0 ;
for (nodeIterator proc(X); proc; ++proc)
{
   r = proc();
   cout << "Proc " << iproc << " getting region " << r << endl ;
   int i,j,k;
   for (indexIterator3 ijk(X(r)); ijk; ++ijk)
   {
   i=ijk(0);
   j=ijk(1);
   k=ijk(2);
   X(r)(i,j,k)=initv;
   cout << "X(" << r << ")(" << i << "," << j << "," << k << ") =
" << initv << endl ;
   }
}
```

The program can be compiled and run interactively on an SP2 system by executing the following commands in a copy of the `region3_init` directory:

```
gmake
poe region3_init -procs 2 -rmpool 1
```

Exercise for the reader:

Create a copy of the `region3_init` directory under a new name, perhaps `region3_init2` and modify the program to initialize the volume to non-trivial values.

A copy can be created in the Unix environment by:

```
cp –r $KELP_HOME/examples/region3_init/ region3_init2/
```

Change the program name and its reference in the `makefile :`

```
cd ~/region3_init2
mv region3_init.C region3_init2.C
```

In the makefile, the line

```
PROG = region3_init
```

should be changed to

```
PROG = region3_init2
```

Edit the source file `region3_init2.C` and make the desired changes to `X(r)(i,j,k)`.

As usual, compile with `gmake` and run the program using your local parallel scheduling environment. For example, on an IBM SP2 system this could be accomplished with

```
gmake
poe region3_init2 -procs 2 -rmpool 1
```

5.3 region3_xcut : the Region Calculus

Program `region3_xcut.C` demonstrates the use of the `Region` intersection operator, `*`. It continues where program `region3_init` left off, with a 3D `XArray` of size (1,1,1) x (Nx,Ny,Nz) partitioned into four Regions `R[r], r = 0…3` and a cyclic distribution across the processor space.

At this stage, each value of the `XArray` is initialized to `1.0`. Suppose we wish to increment the elements of the `XArray` on a sub-volume that runs parallel to the x-axis along the "core" of the global volume. To do so, we declare a `Region3` with the desired index set: one that happens to be one element wide in the `y` and `z` directions, and Nx elements in the x direction:

```
// Create a cross-cut region that intersects core along x-axis.
Region3 RC(1,Ny/2,Nz/2,Nx,Ny/2,Nz/2) ;
```

To complete the task, the newly created `Region3 RC` is intersected with the region of each XArray element, and the elements of the resulting intersection are incremented by a nominal value:

```
// Add some value to each value of the intersecting region.
      double incr = 2.5 ;
      for (nodeIterator xproc(X); xproc; ++xproc)
      {
         r = xproc();
         cout << "Proc " << iproc << " updating region " << r << endl ;
         // iterate across intersection
         int i,j,k;
         for (indexIterator3 ijk(X(r).region()*RC); ijk; ++ijk)[7]
         {
            i=ijk(0);
            j=ijk(1);
            k=ijk(2);
            X(r)(i,j,k) += incr ;
            cout << "X(" << r << ")(" << i << "," << j << "," << k <<
      ") = " << X(r)(i,j,k) << endl ;
         }
      }
```

The "magic" happens in the operation `X(r).region()*RC` which calculates the intersection (a new `Region`) of `RC` and the current local `Region` of `X`. The `iterator` object `ijk` is constructed from this new `Region`, and it generates a set of indices in X for use in the update.

Exercise for the reader:

> Create a copy of the `region3_xcut` directory under a new name and modify the program to initialize the volume to create non-trivial sub-volumes and `Region`*s*. Use the `Region intersection` operator.

[7] Remember that `X(r)` is a `Grid3`. Its `region()` member function returns its bounding box. So `X(r).region()*RC` is a `Region3`.

5.4 region3_move : the MotionPlan and Mover

Program `region3_move.C` completes the introduction to the core KeLP classes. Like the previous example, it also continues where program `region3_xcut` left off, but instead of updating values in a segment of the 3D volume, it moves (copies) values from one sub-volume to another. The source and target sub-volume are specified by two 3-dimensional regions, `RC1` and `RC2`:

```
// Create two regions that intersect the volume
// on either side of x-axis.
Region3 RC1(1,  (Ny/4)-1,     (Nz/4)-1,
            Nx, (Ny/4)+1,     (Nz/4)+1 );

Region3 RC2(1,  (3*(Ny/4))-1, (3*(Nz/4))-1,
            Nx, (3*(Ny/4))+1, (3*(Nz/4))+1 );
```

Recall that the values stored in the entire 3D volume were initialized to `1.0`. To make the example mildly interesting, we initialize each point of the sub-volume defined by `RC1` according to the rank of the intersecting XArray element, i.e. the value `r,` such that `X(r).region()`*RC1 is not empty:

```
// Update region RC1
for (nodeIterator xiter(X); xiter; ++xiter)
{
   r = xiter();
   for (indexIterator3 ijk(X(r).region()*RC1); ijk; ++ijk)
   {
      X(r)(ijk(0),ijk(1),ijk(2)) = 1.0*r ;
   }
}
```

We then create a `MotionPlan` to describe the desired data motion (copy) from RC1 to RC2. Note that the `MotionPlan` `Copy( )` member function does not actually copy any data but instead appends a record describing the source and target for data motion.

```
// Create a MotionPlan to shift-copy blocks from RC1 to RC2
MotionPlan3 Plan;

// Make the motion plan assignments
for (r=0; r<NREGIONS; r++)
{
   int s = (r + 1) % NREGIONS ;
   try {
      Plan.Copy(F,r,F(r)*RC1,F,s,F(s)*RC2);
   }
   catch (BndErr) {
      cout << "Plan.Copy(F," << r <<
                  ",RC1,F," << s << ",RC2) failed." << endl ;
      MPI_Finalize();
      return(0);
   }
}
```

Note that the `Copy` method of the `MotionPlan` carries out all the work in this as follows:

> Copy the data from the region defined by `F(r)*RC1,` which resides on processor `F(r).owner(),` into the region `F(s)*RC2` which resides on processor `F(s).owner().`

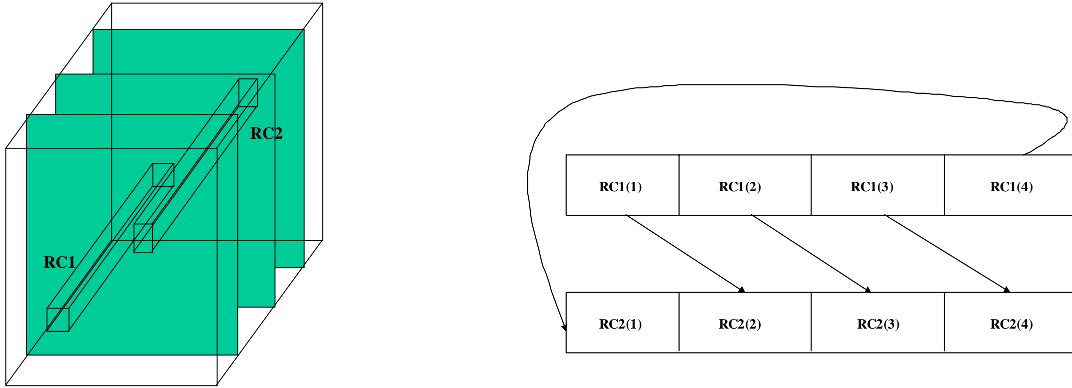


Figure 5c: Left figure shows the core regions described in the text. Right figure shows the pattern of data motion between the two core regions.

To execute this `MotionPlan`, the user needs to create an engine or `Mover` to perform the copy operation. Once the `Mover` has been created, a call the `execute()` member function will transfer the data:

```
Mover3<Grid3<double> > Move(X,X,Plan);
Move.execute();
```

Although this example only moves the data once, a more typical application would update the values in the source region during each iteration and execute the move after the update has finished. In such cases, the `XArrays`, `MotionPlan`, and `Mover` need only be declared once while the `execute` method is called each iteration.

In addition, since the `MotionPlan` describes a pattern of communication while the `Mover` is bound to particular `XArrays`, the mapping can be many to one. Suppose our simple example had declared a second `XArray` built on the same `FloorPlan`:

```
XArray3<Grid3<double> > Y(F);
```

Then to move data in a similar pattern among these arrays we might also have:

```
Mover3<Grid3<double> > MoveX(X,X,Plan);
MoveX.execute();
Mover3<Grid3<double> > MoveY(Y,Y,Plan);
MoveY.execute();
Mover3<Grid3<double> > MoveXY(X,Y,Plan);
MoveXY.execute();
```

Exercise for the reader:

> Modify the program to repeatedly update the source location and call `Move.execute()`, then print out contents of target location.

6 Tutorial II

In this tutorial we show how to implement a simple numerical application, `jacobi2D`, which solves Laplace's equation in two dimensions using 5-point Jacobi iteration. Jacobi's method requires two copies of the computational grid, one to compute updates and one to store the previous iteration. In this example we will utilize a single `Grid` with two solution fields.

6.1 Problem Description

We will use `jacobi2D` to solve Laplace's equation in two dimensions:

$$\Delta u = 0$$

for a real-valued function *u* of two variables. Ω is the computational box, a subset of R^2. We specify Dirichlet boundary conditions on $\partial\Omega$, the boundary of Ω:

$$u = g \quad \text{on} \quad \partial\Omega$$

where *g* is also a real-valued function of two variables.

We discretize the computation using the method of finite differences; we now solve a set of discrete equations defined on `W`, a regularly spaced (N+2) x (N+2) set of points in Z^2. We number the *interior* points of `W` from 1 to *N* in the *x*-coordinate and from 1 to *N* in the *y*-coordinate. The *boundary* points, which contain the Dirichlet boundary conditions for the problem, are located along *x*-coordinates of and *N*+1 and along *y*-coordinates of and *N*+1, as shown in Fig. 6a.

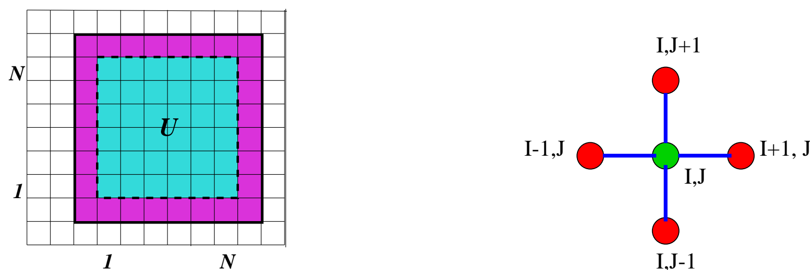


Figure 6a: A finite difference mesh (left) and the accompanying finite difference stencil (right). The Grid `U` is defined over the 2-dimensional `Region domain(0,0,N+1,N+1)`, i.e. the rectangle between lower bound `(0,0)` and upper bound `(N+1,N+1)`. The solution is computed on the inner portion of the mesh; the boundary conditions are supplied on the outer edge. This boundary region is one cell thick, as dictated by the finite difference stencil for the smoother, which in our case updates each cell as a function of its nearest neighbors along the Manhattan directions.

6.2 Main Routine

The main routine appears below as Fig. 6b. For clarity of exposition the version reproduced here has been stripped of comments and I/O statements. A complete version of following programming example is included in the source distribution in `kelp1.3/apps/jacobi2D/noAPI/`.

```
1) MPI_Init(&argc,&argv);
2) InitKeLP(argc,argv);
3) MotionPlan2 M;
4) try {
5) int N = ParseCommandLine(argc,argv);
6) Region2 domain(1,1,N,N);

7) Region1 alternates(1,2);
8) int cs = alternates.lower(0);
9) int ce = alternates.upper(0);

10)   FloorPlan2 T = UniformPartition(domain);
11)   XArray2<Grid2<double> > grid1(T,alternates);

/* Initialize the local grid */
12)   InitGrid(grid1,cs,ce);

/*******************************/
/* Do the Jacobi computation. */
/*******************************/

13)   XArray2<Grid2<double> > *oldgrid = &grid1;
14)   double stop = 1.0;
15)   initMotionPlan(T,M);
16)   Mover2<Grid2<double> > pDM1(grid1,grid1,M);
17)   for (int i= 0; i<NITERS; i++) {
/* Exchange boundary data with neighboring processors */
18)   pDM1.execute(Region1(cs,cs),Region1(cs,cs));
/* Perform the local jacobi computation */
19)   ComputeLocal(*oldgrid,cs,ce);
/* Swap the grids */
20)   SwapIndex(&cs, &ce);

21)   #ifdef COMPUTE_NORM
22)   /* Compute the stopping criterion */
23)   stop = L2Norm(*oldgrid, cs,ce);
24)   OUTPUT("L2 Norm : " << stop << endl);
25)   #endif
   }
}
26)   catch (KelpErr & ke) {
27)   ke.abort();
28)   }
29)   MPI_Finalize();
30)   return(0);
   }
```

Figure 6b: Main procedure for jacobi2D example.

The first two statements of any KeLP program should initialize MPI and the KeLP run-time system, (1) and (2):

```
(1)  MPI_Init(&argc,&argv);
(2)  InitKeLP(argc,argv);
```

Next (3), we declare (instantiate) the `MotionPlan` which will direct the communications.

Following this, we open the main `try` block (4) in order to catch any exceptions that may occur[8] and read in the problem size (5). Next (6), we define our computational box with an object of class `Region` (see Fig. 6a):

```
(6)  Region2 domain(1,1,N,N);
```

This defines a 2d `Region` valued variable called `domain`. Class `Region` is (strongly) typed by the number of spatial dimensions. `Region2` represents a 2-dimensional `Region`, `Region3` a 3-d region, and so on. Lines (7 – 9) then define the extents of the solution fields.

Next, on line (10), we call a user function to partition the computational domain across processors.

Now that we have defined the storage required by each processor, we are ready to instantiate storage to represent the distributed grids. An `XArray` is a distributed array of `Grids.` Each `Grid` may be assigned to an arbitrary processor. Statement (11) instantiates an `XArray` with the grid structure we have set up in `FloorPlan T`. Note that the dimensionality of the `XArray` (in this case, 2) must match the dimensionality of the component `Grid` type.

```
(11) XArray2<Grid2<double> > grid1(T,1,alternates);
```

Statement (12) is a function call which sets up initial boundary conditions and an initial guess for the unknowns. This function is simple and is not discussed here. (See the complete listing in the source distribution for more details). Statement (13) just defines a pointer we will use to refer to the distributed data structures.

```
(12) InitGrid(grid1);
(13) XArray2<Grid2<double> > *newgrid = &grid2
```

Statement `(15)` calls the function `initMotionPlan` which, given the `FloorPlan`, creates the `MotionPlan`.

After specifying the pattern of block copy operations in the `MotionPlan` we are now ready to carry out the communication expressed by the pattern. We instantiate a `Mover` object using the `MotionPlan M`. The Mover is responsible for moving data within a single XArray (16). Note that the use of solution fields has allowed us to instantiate only a single `XArrary` and a single `Mover`.

Statements (17-20) contain the main loop of the iterative solver. First the Mover is executed to exchange ghost cells (18). This operation updates off-processor values for each `Grid`. (Fig. 6d.). `ComputeLocal (20)` performs local smoothing on each `Grid component`. `SwapIndex(21)` just renames the storage for the next loop iteration.

```
(17)     for (int i= 0; i<NITERS; i++) {
```

[8] KeLP may be compiled in a special error checking mode to permit the KeLP to throw exceptions that result, for example, from illegally indexing an XArray, or attempting to move data outside the region of an XArray component. This mode is normally shut off as it imposes a run time overhead.

```
(18)        pDM1->execute(Region1(cs,cs),Region1(cs,cs));
(20)        ComputeLocal(*oldgrid,cs,ce);
(21)        SwapIndex((&cs, &ce);
         }
```

Statements (22 – 25) compute a residual and test for convergence.

Statements (26-27) catch any errors that arose during execution, and use the C++ exception mechanism. Statements (28) and (29) should be the last two lines of any KeLP program.

```
(28) MPI_Finalize();
(29) return (0);
```

6.3 UniformPartition

Function `UniformPartion` partitions the computational domain across processors. In general, the programmer must determine how to accomplish this for complicated domains. However, for this simple example, the KeLP distribution includes a library of classes—the `DOCK` library—to effect HPF style block decompositions. The `dock` library includes two classes—`Processor` and `Decomposition`—which enable us to define a logical processor array and to define a decomposition of the domain across the processors. Once we have determined a decomposition, then we will add ghost cells, and then finally instantiate the necessary `XArray`(s). The code for `UniformPartion` appears as Fig 6c.

```
FloorPlan2 UniformPartition(Region2& domain)
}
(1) Processors2 P;
(2)    Decomposition2 T(domain);
(3)    T.distribute(BLOCK1,BLOCK1,P);
(4)     int index;
(5)     for (indexIterator1 ii(T); ii; ++ii) {
(6)       index = ii(0);
(7)       T.setregion(index,grow(T(index),1));
(8)     }
(9)     return T;
}
```

Figure 6c: Function `UniformPartition` generates the uniform BLOCK partitioning which is shown in Fig. 6d.

Using the `DOCK` mapping classes, we first declare a virtual processor array corresponding to the physical processor set (1). A default `Processors2` object creates a 2D logical processor array.

```
(1)  Processors2 P;
```

Recall that a `FloorPlan` represents the `Regions` and processor assignments for a set of distributed `Grids.` A `Decomposition` is a class publicly derived from `FloorPlan` that will automatically create a block data distribution for a computational domain. Statements (2) and (3) thus create a `FloorPlan` `T` representing a block data decomposition, mapped to the physical processor array.

```
(2)  Decomposition2 T(domain);
(3)  T.distribute(BLOCK1,BLOCK1,P);
```

Such a partitioning is shown in Fig. 6d.

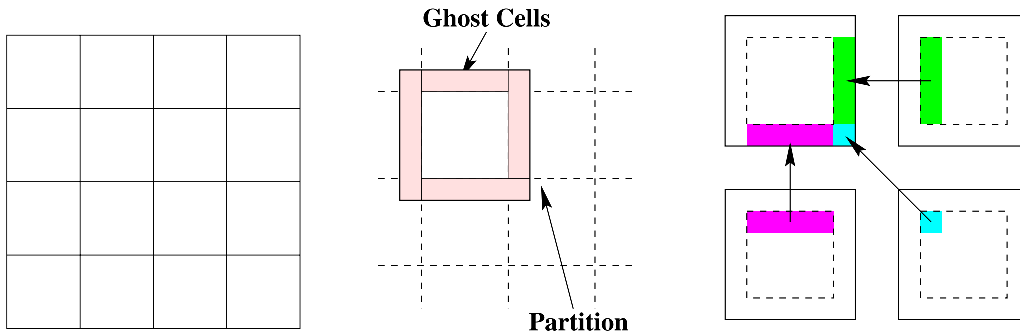


Figure 6d: Block partitioning into 16 regions (left), showing the ghost region for a typical partition (middle). The right figure gives the dependencies that must be satisfied to refresh the ghost cells.

Next, we pad each processor's partition with an additional ghost cell in each direction to hold Dirichlet boundary conditions and off-processor values. The loop at lines (4-7) accomplishes this. We use a serial `indexIterator1` to iterate over each element in `FloorPlan T`.[9] We get the index of the FloorPlan element for the current iteration with the ( ) indexing operator, and use the index to extract the region of the FloorPlan element.[10] For a `Region R`, `grow(R,1)` defines a `Region` padded with one cell in each direction. Thus, statement (7) grows the region of the current FloorPlan element as needed. The `setregion()` member function updates the element at the current index by the region-valued expression specified by the second argument.

```
(4)  int index;
(5)  for (indexIterator1 ii(T); ii; ++ii) {
(6)    index = ii(0);
(7)   T.setregion(index,grow(T(index),1));
(8)  }
```

6.4 initMotionPlan

The `initMotionPlan()` function (Fig. 6d) fills in the ghost cells for each processor's partition in an `XArray`. We now describe the KeLP code to effect this data motion, which appears as follows.

```
  void initMotionPlan(FloorPlan2& F,  MotionPlan2 &M ){

(1)  for (indexIterator1 ii(X); ii; ++ii) {
(2)    int i = ii(0);
(3)    Region2 inside = grow(F(i), -1);
(4)    for (indexIterator1 jj(F); jj; ++jj) {
(5)      int j = jj(0);
(6)      if (i != j) M.CopyOnIntersection(F,i,F,j,inside);
       }
  }
```

Figure 6e: initMotionPlan`()` function for jacobi2D example

[9] Remembering that KeLP iterators are strongly typed by dimensionality, we infer that a FloorPlan is implicitly 1-dimensional. We also recall that we may construct the iterator using a FloorPlan argument.
[10] By an abuse of notation, the indexing operator ( ) returns the region of the FloorPlan element. Alternatively, we could use the region( )member function: `T.region(index)`.

Now, we iterate serially over each Region in the FloorPlan (1-8), recalling that when we index a FloorPlan element, we get back a Region:

```
(1)  for (indexIterator1 ii(F); ii; ++ii) {
         ...
(6) }
```

For each FloorPlan element F(i), first we determine the Region of F(i) that corresponds to virtual processor i's portion of the solution; the grow() operation (3) is needed to trim off the ghost cells:

```
(3)    Region2 inside = grow(F(i).region(), -1);
```

We then loop again over each FloorPlan element F(j) (4-6). For each FloorPlan element other than F(i), we add to the MotionPlan a block copy operation meaning "copy from F(i) into F(j) where these FloorPlan elements intersect the Region inside" (6). As a consequence of how we previously allocated storage, this MotionPlan corresponds to copying "solution"' values from Grid X(i) into ghost cells on X(j).

```
(6)  for (indexIterator1 jj(F); jj; ++jj) {
(7)      j = jj(0);
(8)      if (i != j) M.CopyOnIntersection(F,i,F,j,inside);
(9) }
```

6.5 ComputeLocal

The ComputeLocal() function performs local Jacobi relaxation on each Grid in an XArray (Fig. 6e). We now show how to implement this operation by calling a serial Fortran 77 routine from KeLP.

```
1) #define f_j5relax FORTRAN_NAME(j5relax_, J5RELAX, j5relax)
2) extern "C" {
3) void f_j5relax(const double *const u, FORTRAN_ARGS2, const int* cs,
   const int* ce);
4) }
5) void ComputeLocal(XArray2<Grid2<double> >& oldgrid, const int cs,
   const int ce)
6) {
7) int i;
8) for (nodeIterator ni(oldgrid); ni; ++ni) {
9)    i = ni();
10)   Grid2<double>& OG = oldgrid(i);
11)   FortranRegion2 Foldgrid(OG.region());
12)   f_j5relax(OG.data(), FORTRAN_REGION2(Foldgrid), &cs, &ce);
13)   }
14)   };
```

Figure 6e: ComputeLocal() function for jacobi2D example.

Using the nodeIterator, we iterate in parallel over each Grid in the XArray (8 - 13).

We will pass each Grid OG (10) to be smoothed to a serial Fortran 77 smoother.

```
(10)      Grid2<double>& OG = oldgrid(i);
```

Because separately compiled Fortran modules do not understand class Grid, or even the notion of classes, KeLP provides Grid member functions to extract the data portion and the bounding box of a Grid in a

standard canonical form that can be interpreted by Fortran. The current implementation of KeLP is targeted to Fortran users; `Grid` data is stored in Fortran standard column major order.

Since Fortran name mangling conventions vary from compiler to compiler, we use C preprocessor macros to redefine Fortran names as appropriate (a compiler uses a name mangling convention to generate names used by the linker and loader). For example, many compilers, such as `f77` under Solaris, change all Fortran letters in the function name to lowercase and add a trailing underscore "_". The UNICOS compilers convert all letters to uppercase and omit the trailing underscore. The AIX compilers change all letters to lowercase and omit the training underscore. KeLP defines a preprocessor macro called `FORTRAN_NAME` that helps to convert Fortran names to their name-mangled equivalents (1).

```
(1) #define f_j5relax FORTRAN_NAME(j5relax_, J5RELAX, j5relax)
```

The Fortran 77 smoothing routine is shown in Fig. 6f. We use the `extern "C"` mechanism to declare this function interface in `C++` (2-4).

```
(2) extern "C" {
(3) void f_j5relax(const double *const u, FORTRAN_ARGS2,
           const double *const v);
(4) }
```

All arguments are passed by reference in Fortran, so we must pass `C++` pointers. `FORTRAN_ARGS2` is a macro that is provided to declare four Fortran integer arguments, which are intended to define a `Region`.

Next, we construct an object of class `FortranRegion2` which represents the `Region` object we wish to pass to the Fortran integer parameters(16).

```
(11)      FortranRegion2 Foldgrid(OG.region());
```

Finally, we are ready to invoke the Fortran routine. Since Fortran accepts parameters with call-by-reference, we must pass a `C++` pointer to Fortran to represent an array. For a `Grid<T> G`, `G.data()` returns a `T*` pointing to the first element of data in `G`. Also, we use the `FORTRAN_REGION2` macro to convert the `FortranRegion2` to four integer arguments, to be passed by reference to Fortran. With these facilities, statement (18) calls the serial Fortran smoother with the appropriate arguments.

```
(12)      f_j5relax(OG.data(), FORTRAN_REGION2(Foldgrid), &cs, &ce);
```

The Fortran 77 code appears as follows:

```
      subroutine j5relax(u,ul0,ul1,uh0,uh1,cs,ce)
      integer ul0, ul1,uh0, uh1, cs, ce
      double precision u(ul0:uh0,ul1:uh1,1:2)
      integer i, j
      do j = ul1+1, uh1-1
         do  i = ul0+1, uh0-1
           u(i,j,ce) = (1.0d0/4.0d0) *
   2                   (u(i-1,j,cs) + u(i+1,j,cs) + u(i,j-1,cs) +
   3                   u(i,j+1,cs))
         end do
      end do

      return
      end
```

Figure 6f: Fortran 77 serial smoother.

In this simple programming example we have introduced the concepts of a `Region`, `Grid`, `FloorPlan`, `XArray`, `MotionPlan`, and `Mover`. See the *KeLP Reference Manual* for a more complete description of facilities provided by these classes.

6.6 Reusability

The above code is completely portable. We may develop it on a workstation, and after the code appears to run correctly, we may then carry out performance tuning and production runs on parallel. In addition to being architecture independent, the logic of the code is also independent of the way the problem has been partitioned. We could have our partitioner return a 1-dimesional data decomposition without having to make any other changes whatsoever. We could even compute over an irregularly shaped domain, such as the Lake Superior geometry shown in Fig. 1a, without having to change the logic of the code. In addition, the logic of the code that generates the MotionPlan is nearly dimension independent. We may reuse our `initMotionPlan()` routine in a 3-dimensional problem with only modest changes: we change the dimensionality specifier in the classes Region, FloorPlan, and MotionPlan. In fact, with the help of a macro pre-processor we are able to maintain the core KeLP classes in dimension-independent form, and expand into the concrete typed forms automatically at build time.

7 Advanced features

This section provides limited documentation on some miscellaneous features of KeLP, which are recommended for the experienced KeLP user. These features may prove useful for building DSL classes. As always, feel free to contact us with questions and suggestions via the web page

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

7.1 Beyond GridX: The Patch Abstraction

Patch defines an abstract public interface to the user-defined container class ("user defined patch"), and is provided with the KeLP1.4 distribution as the file `Patch.h`. This file provides a specification or template for users to follow when implementing their own container class.

`Patch` must define certain member functions for use by the KeLP run time system. These functions are generally not called by user applications, but they must be defined appropriately to ensure that KeLP operates correctly. The correct operation of KeLP in effect verifies conformance to the `PatchX` specification.

A `Patch` may also have one or more indexible solution fields, with a separate index domain which is a KeLP `Region1`. Examples of solution fields are the vector components of velocity, or scalar quantities such as temperature, pressure, etc.

7.1.1 DataFactory

`Patches` are instantiated using a `DataFactory` class. A data factory is a design pattern [6]. It is needed when the region and field parameters to a Patch constructor are not sufficient to completely specify how much storage needs to be allocated. For example, in embedded boundary methods, the allocation of memory for the array depends on geometry information. A natural way to communicate this information is by passing the geometry object to the constructor, but then KeLP has to know about that detail of the application, which is undesirable. The Factory idiom is a way around that. It creates a generic wrapper that takes only Region plus number of fields, while it may have internal access to the additional state required to perform the construction.

Patches are not as a rule declared in application code. Rather the core KeLP classes `XArray` and `Mover` will refer to the appropriate member functions of a Patch class to handle data construction and communication.

Factory must supply an instantiate(RegionX, Region1) function which instantiates and returns a pointer to a Patch object. The XArray constructor then calls this function. For ease of use and backward compatibility a default Factory implementation, which simply passes the Region arguments to the Patch constructor, is included in the release. *Thus, providing a user defined `Factory` class is optional for any `Patch`, e.g. `Grid`, which can be completely specified by supplying a (RegionX, Region1) pair to its constructor.*

The reader is referred to the Patch specification, Patch.h, and the **Reference Manual** for further details.

7.2 Using command-line arguments

Some run-time systems, batch schedulers, and the like add command-line arguments for their own internal use. Unfortunately, some of these systems perturb the original positions of arguments so that the values of `argv` and `argc` passed to `main()`. For example, you might ask the scheduler to execute program `rb82` with two arguments:

```
rb82 100 12300
```

and expect that `argc` is 3, `argv[1]` is `100`, and `argv[2]` is `12300`. However, under some systems (e.g., the MPICH implementations of MPI installed with p4) it might occur that "your" command line

arguments have been shifted (in bulk) to the right or left by 1 or more positions. There are at least two ways to avoid this problem:

1. Use flags to identify your arguments and have your command-line parser look for flags instead of positions. For example:

```
rb82 -size 100 -iters 12300
```

2. In our experience with MPICH, process 0 in the application receives the correct command-line arguments, and then two additional arguments. The suggested procedure is to have process 0 parse the command-line arguments and then broadcast the values to other processes as needed. See the `jacobi2D` sample code in the source distribution which performs this operation

7.3 Instantiating user-defined classes

KeLP uses C++ templates to instantiate various classes. The following KeLP classes may be instantiated with a user-defined class as a parameterized type[11]:

- `XArray`
- `Mover`

`Grid` also uses templates. A `Grid` may be instantiated with any user-defined class `foo` with a fixed size, where `memcpy(sizeof(foo))` creates a valid copy of a `foo` object. An `XArray` may be instantiated from any class derived from `Grid`. So, the following is legal:

```
class myGrid: public Grid2<double> { . . .};
XArray2<myGrid> X;
```

The preceding paragraph applies just as well to a user defined template `Patch` class.

A `Mover` (or `VectorMover)` class must be instantiated with: a class derivable from `PatchX.`. Thus, the following is legal:

```
Mover2<myGrid> M;
```

Forcing a `C++` compiler to instantiate templates can be tricky, especially in a library. If you have problems, look through the KeLP libraries and DSL code to see how KeLP instantiates templates for the supported compilers.

7.4 Direct MotionPlan Modification

Currently the `MotionPlan` class is exposed to the programmer and may be manipulated as a first-class object. This process is recommended for the seasoned user. See the `MotionPlan` header file and the sample program in `kelp1.3/apps/jacobi3D/contig` for more information on manipulating the `MotionPlan` directly.

7.5 Customizing the Mover

A KeLP `Mover` usually packs data into a message buffer, ships the buffer to a remote processor, and then unpacks the message buffer into user data structures (`Grids`). (For `Grids` that live on the same physical processor, a copy operator performs the data transfer with no intermediate buffer.) In some cases, it may be desirable to associate numeric operations to accompany communication. To simplify this process, the KeLP `Mover` classes may be customized through inheritance. The simplest way to do this is to install user-defined pack, unpack, and copy operators.

---

[11] Additional classes—Array, List, and VectorMover—which are not employed by the casual KeLP user, are treated similarly.

The pack, unpack, and copy operators are virtual functions in the `Mover` class. Thus, you may derive specialized `Movers` that redefine these operators to perform some numeric function. For example, the finite difference DSL derives a `Mover` called `Adder` that adds the source data to the destination data, instead of simply copying. This is accomplished by re-defining the unpack and copy operators. See the DSL source code for more details.

Note: Normally the KeLP system automatically copies messages directly into user data structures where possible, bypassing calls to `pack()` and `unpack()`. If you wish to ensure that `pack()`, and `unpack()` are always called, you must tell the KeLP system to do this with

```
KeLPConfig(CONTIG_MSG_IN_PLACE,FALSE);
```

7.6 Using a DSL

While the previous tutorials illustrate the use of the core KeLP abstractions, the recommended way to use KeLP is to build a *Domain Specific Library (DSL)* that encapsulates common operations for a narrow class of applications. For illustrative purposes, the KeLP distribution contains a very simple DSL with some facilities that help solve elliptic PDEs with finite differences. The distribution also includes a version of the jacobi2D code using the DSL in `kelp1.3/apps/jacobi2D/useAPI`.

In this section we will examine some KeLP code used to implement the sample finite difference DSL. See the *KeLP Reference Manual* for more complete documentation of the DSL facilities. The DSL defines an `IrregularGrid` class, which implements a distributed array with a possibly irregular data decomposition. An `IrregularGrid` is basically an `XArray` with specialized added capabilities. An `IrregularGrid` understands its own ghost cells and provides member functions to help with initialization and convergence checking.

One powerful facility provided by `IrregularGrid` is `fillGhost()`, which fills in ghost regions as described above. The `fillGhost()` communication pattern is stored by a `MotionPlan` in the `IrregularGrid` object:

```
class IrregularGrid2: public XArray2<Grid2<double> > {
    MotionPlan2 fgPlan;            // motionPlan for ghost regions
public:
    IrregularGrid2();
    void fillGhost();
    void fill(const double& d);
    ...
};
```

The structure of an `IrregularGrid` is naturally represented by a `FloorPlan`, with an additional vector that holds the number of ghost cells in each dimension. The `dock` library defines the `GhostPlan` class to hold the structure of an `IrregularGrid`.

```
class GhostPlan2: public FloorPlan2 {
    Point2 _ghost;
public:
    ...
};
```

It is recommended that you peruse the DSL and `dock` source code and sample applications to get a better feel for how to build a DSL and use it along with KeLP facilities. It is hoped that the sample DSL provided will serve as a starting point for your application.

7.7 Other features

The KeLP and DSL source code is concise and reader-friendly. After becoming familiar with KeLP, you will probably find it fairly easy to go through the KeLP header files and source code to answer your

questions. In some cases, ambitious users have modified even the most complex KeLP classes (`Mover` and `VectorMover`) to realize operations the system developers did not anticipate. Feel free to tailor KeLP to your own needs, if you need some feature KeLP does not provide.

8 Acknowledgments

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, the KeLP User's Guide, v1.0, and “The Data Mover: A Machine-Independent Abstraction for Managing Customized Data Motion,” LCPC 99, La Jolla, CA, 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.

KeLP was the Ph. D. topic of Stephen J. Fink. KeLP is derived in part from the LPARX system, which was the Ph.D. thesis topic of Scott Kohn.

9 References

1. G. Agrawal, A. Sussman, and J. Saltz, “An integrated runtime and compile-time approach for parallelizing structured and block structured applications.” *IEEE Transactions on Parallel and Distributed Systems* 7(6) (1995), pp. 747-754.

2. S. R. Kohn and S. B. Baden, “Irregular Coarse-Grain Data Parallelism Under LPARX.” *Scientific Programming*, Vol. 5 (1996), p. 185-201.

3. M. J. Berger and S. H. Bokhari, “A partitioning strategy for nonuniform problems on multiprocessors.” *IEEE Transactions on Computers*C-36 (1987), pp. 570-580.

4. S. B. Baden and S. J. Fink, “The Data Mover: A Machine-Independent Abstraction for Managing Customized Data Motion,” *Twelfth International Workshop on Languages and Compilers for Parallel Computing (LCPC '99),* J. Ferrante et al. Eds., La Jolla, CA (1999). Springer-Verlag (To appear).

5. S. J. Fink, S. R. Kohn, and S. B. Baden, “Efficient Run-time Support for Irregular Block-Structured Applications.” *J. Parallel and Distributed Computing*, 50(1-2) (1998), pp. 61-82.

6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns, Addison Wesley, 1995