Efficient Run-time Support for Irregular
Block-Structured Applications *

Stephen J. Fink, Scott B. Baden
Department of Computer Science and Engineering
University of California, San Diego

Scott R. Kohn
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

*This work was supported in part by ONR contract N00014-93-1-0152 and by the DOE Computational Science Graduate Fellowship Program. Computer time on the Intel Paragon at the San Diego Supercomputer Center was provided by a UCSD School of Engineering block grant. Access to the IBM SP2 was provided by the Cornell Theory Center.
Abstract

Parallel implementations of scientific applications often rely on elaborate dynamic data structures with complicated communication patterns. We describe a set of intuitive geometric programming abstractions that simplify coordination of irregular block-structured scientific calculations without sacrificing performance. We have implemented these abstractions in KeLP, a C++ run-time library. KeLP’s abstractions enable the programmer to express complicated communication patterns for dynamic applications, and to tune communication activity with a high-level, abstract interface. We show that KeLP’s flexible communication model effectively manages elaborate data motion patterns arising in structured adaptive mesh refinement, and achieves performance comparable to hand-coded message-passing on several structured numerical kernels.
1 Introduction

Many scientific numerical methods employ structured irregular representations to improve accuracy. Irregular representations can resolve fine physical structures in complicated geometries, but are difficult to implement due to elaborate, dynamic data structures. These structures give rise to unpredictable, irregular communication patterns, making parallel programming of irregular calculations especially challenging. In order to get good performance on today’s non-uniform parallel memory hierarchies [2], the programmer must judiciously exploit parallelism and locality in the application to match the hardware capabilities.

To ease the programmer’s burden, programming languages and libraries can hide many low-level details of a parallel implementation [20, 24, 16, 1, 12, 10, 35, 15, 11, 23, 28, 4]. We present Kernel Lattice Parallelism (KeLP), a C++ class library that provides high-level abstractions to manage data layout and data motion for dynamic block-structured applications. Block structures arise in many scientific applications ranging from finite difference methods for partial differential equations [8] to blocked algorithms for numerical linear algebra [13]. In these applications, computational structures arise as uniform rectangular arrays of data, which communicate in potentially irregular patterns.

KeLP supports a small set of geometric programming abstractions to represent data layout and data motion patterns. KeLP’s data orchestration model separates the description of communication patterns from the interpretation of these patterns. The programmer uses intuitive geometric constructs to express dependence patterns among dynamic collections of arrays. Additionally, KeLP utilizes structural abstraction, introduced in the LPARX programming system[24, 26], to separate the description of data decompositions from the underlying storage implementation.

KeLP’s innovation lies in first-class communication schedule objects, called MotionPlans, which the programmer can define and manipulate with intuitive geometric operations. KeLP provides first-class objects called Movers to interpret communication schedules according to the needs of the application and hardware. With these primitives, the programmer may customize the data motion pattern and interpretation of the pattern by separately optimizing the MotionPlan and the Mover.

Such flexibility is vital for block-structured applications with complex or irregular data motion patterns based on run-time characteristics of the application. Consider, for example, an adaptive multigrid method which has been employed in ab-initio materials design[9, 25]. This application contains dynamically adapting collections of 3D blocks of data, arising in response to error in the computed numerical solution. The error in the solution varies non-uniformly in space according to
to appear in *J. Parallel and Distributed Computing*

evolving conditions in the data. The number of blocks per level will vary at run time, along with their shape, size, and physical location with respect to one another. (See Fig. 1.) The collections of data communicate in five distinct patterns to satisfy five different types of data dependencies. Naturally, each scientific application uses a distinct set of data motion patterns. For example, a shock hydrodynamics code will demand a set of data motion patterns which match the conservative differencing schemes used for hyperbolic PDEs [5].

KeLP has been employed in several projects demanding run-time partitioning and communication patterns, including structured adaptive mesh refinement applications [27, 33] and studies of load-balancing in heterogeneous computing environments[6, 18]. In heterogeneous computing environments, even uniform problems can present tricky data layout and data motion problems. For example, on a heterogeneous collection of processors, load balancing concerns will dictate an irregular work decomposition even for a uniform problem [22]. KeLP’s mechanisms simplify the expression of such irregular block data decompositions and communication patterns at run-time.

KeLP requires only basic message passing capabilities (MPI[30]) and has been ported to the IBM SP2, Cray T3E, Intel Paragon, SGI Power Challenge Array, SGI Origin 2000, clusters of workstations and single processor workstations. In this paper, we compare KeLP performance against hand-coded MPI implementations of the NAS Multigrid (MG) and Fourier Transform (FT) benchmarks [3], as well as the SUMMA matrix multiply algorithm [40]. Running on the IBM SP2 with up to 64 processors, KeLP codes achieves performance comparable to hand-coded versions using message-passing directly. For a uniform finite difference calculation, we show how high-level KeLP optimizations can reduce communication costs by a factor of two.

The remainder of this paper proceeds as follows. Section 2 describes KeLP’s programming model. Section 3 applies KeLP to a simple example problem. Section 4 discusses the implementation. Section 5 presents performance results for various applications. Section 6 discusses related work and Section 7 summarizes the results.

## 2 Programming Model

KeLP’s programming model relies on two levels of control flow, resembling the X and Y program levels in Snyder’s Phase Abstractions model [39]. Programs begin with a single logical thread of control. Periodically, the KeLP program enters a *forall* loop in which each loop iteration executes independently on exactly one SPMD process (Figure 2).
KeLP’s storage model consists of $P$ distinct address spaces, as commonly provided by distributed memory parallel computers. KeLP supports distributed collections of block-structured objects, where each block lives in a single address space. Each address space represents a single logical processor. In this paper, we only consider the case where a logical processor corresponds to a single physical processor. However, the KeLP model does not exclude multiprocessor nodes; a single address space could correspond to a shared-memory multiprocessor or a processor subset.

In order to manage potentially irregular block data decompositions at run-time, KeLP provides first-class language objects to represent data decompositions. The programmer must explicitly describe data decompositions. KeLP does not provide automatic data decomposition, since the partitioning strategy depends on application-specific knowledge. However, the KeLP distribution includes libraries that generate common partitioning strategies, and the programmer can implement application-specific partitioning policies with KeLP’s geometric abstractions.

The programmer specifies explicit data motion using block copy operations between distributed objects. KeLP does not implement implicit data motion; there is no compiler analysis of inter-block data dependencies. The programmer must determine the data motion needed to ensure memory consistency between distributed blocks. KeLP abstractions simplify the expression of the data motion pattern with intuitive geometric operations.

As shown in Table I, the KeLP data types fall into two sets of abstractions. The first set, consisting of Point, Region, FloorPlan, and MotionPlan, encode geometric information and are hardware-independent. The second set, consisting of Grid, XArray, and Mover, instantiate storage or communication activity. Their interpretation depends on the target architecture’s communication and storage characteristics.

### 2.1 Data Layout Abstractions

KeLP manages data layout with an extended version of the *structural abstraction* model, introduced in the LPARX programming system [24, 26]. With structural abstraction, a geometric abstraction stores the structure a data set separately from the data itself.

KeLP inherits four core data decomposition abstractions from LPARX: Point, Region, Grid, and XArray. KeLP extends the model with a fifth abstraction, the FloorPlan.

A Point is an integer $n$-tuple representing a point in $Z^n$. Element-wise addition and scalar multiplication are defined over Points in the obvious way. For a Point $P$, $P(i)$ returns the $i$th
to appear in *J. Parallel and Distributed Computing*

component of $P$.

A Region is a first-class object that represents a rectangular subset of $\mathbb{Z}^n$. A Region is identified by $[P_1, P_2]$ where $P_1$ and $P_2$ are the two Points at the lower and upper corners of the Region. For a Region $R$, $\text{lwb}(R)$ and $\text{upb}(R)$ return $P_1$ and $P_2$, respectively. $\text{Extents}(R, i)$ returns the length of the Region along the $i$th axis; i.e., $P_2(i) - P_1(i) + 1$.

To help the programmer manipulate Regions, KeLP provides a set of high-level geometric operations called the *Region calculus* (Table II). Three useful Region calculus operations are *shift*, *intersect*, and *grow*. For a Region $R$ and a Point $P$, $\text{shift}(R, P)$ denotes $R$ translated by the vector $P$: $[\text{lwb}(R) + P, \text{upb}(R) + P]$. *Intersection* between two Regions is defined in the usual geometric way. Given a Point $g$, $\text{grow}(R, g)$ returns the Region that is equivalent to $R$ padded with $g(i)$ cells in the $i$th direction. If $g(i) < 0$, the Region is shrunk in the obvious way. Note that all Region operations can be implemented efficiently with a small number of integer operations, and Regions are closed under *shift*, *intersection*, and *grow*.

A Grid is an array of objects, all of the same type, whose index space is a Region. For example, the Fortran-90 array defined as `real A[3:7]` corresponds to a one-dimensional Grid $A$ of `real` with $\text{region}(A) = [3, 7]$. A Grid $G$ may not span multiple processor memories; rather, it lives in a single address space. For convenience, most KeLP Region calculus operations are defined over Grids. For example, if $G_1$ and $G_2$ are Grids, then $G_1 \cap G_2$ returns the Region $\text{region}(G_1) \cap \text{region}(G_2)$.

An XArray is an array of Grids, which collectively represent a potentially irregular block-structured data set. Each Grid element in an XArray may have a different shape and processor owner, but all must have the same rank and element type (see Figure 3). We may access the Grids in an XArray through an index; i.e. $X(0)$ and $X(1)$ are the first and second (Grid) elements of XArray $X$.

The KeLP FloorPlan is a one-dimensional array of Regions with processor assignments. The FloorPlan provides a useful first-class abstraction to encapsulate the notion of a data decomposition. The programmer may use Region calculus operations to modify a FloorPlan, and then use the FloorPlan to instantiate an XArray with a given data decomposition. Just as a Region holds the index domain of a Grid, a FloorPlan represents the index domain and processor assignments for an XArray.
2.2 Data Motion Abstractions

In applications such as adaptive multigrid, a collection of grids (i.e., a KeLP XArray) communicate as a single logical unit. For example, in Figure 1b, the irregularly-shaped fine level communicates with the irregularly-shaped coarse level in the shadow cast by the fine level. KeLP represents a data motion pattern between XArrays using the MotionPlan abstraction. The MotionPlan is a first-class communication schedule [16, 1] object that encodes a set of array section copy operations between XArrays (Fig. 4). The programmer builds and modifies a MotionPlan using Region calculus operations described in the previous sub-section.

KeLP data structures communicate via block copy operations. A single block copy operation is described by two Regions and two Grids. Let \( G_1, G_2 \) be two Grids, and \( R_1, R_2 \) be two Regions. Then \( \{ G_1 \text{ on } R_1 \Rightarrow G_2 \text{ on } R_2 \} \) denotes a block copy operation, meaning copy the values from \( G_1 \) at indices in \( R_1 \) into \( G_2 \) at indices in \( R_2 \).

To incrementally construct a MotionPlan, the programmer adds block copy operation descriptors to a MotionPlan, one at a time. For a MotionPlan \( M \), two XArrays \( X \) and \( Y \), and Regions \( R_1 \) and \( R_2 \), \( M.copy(X, i, R_1, Y, j, R_2) \) appends a descriptor of the block copy operation \{ \( X(i) \) on \( R_1 \Rightarrow Y(j) \) on \( R_2 \) \} to \( M \). We emphasize that this generates a description of the copy operation. The MotionPlan does not carry out any communication; that task falls to the Mover.

The Mover is a first class object that interprets a MotionPlan and performs the indicated data motion as a collective communication operation. The Mover object analyzes a MotionPlan and performs memory-to-memory copies and message-passing to effect the data motion pattern. If data for a particular message happens to reside contiguously in memory, the Mover bypasses intermediate message buffer packing. As we will see in Section 5, such optimization can reduce communication overhead significantly.

Prior to executing a Mover the programmer may modify an extant MotionPlan using Region calculus operations. KeLP provides operators to query and modify individual block copy descriptors in a MotionPlan. MotionPlan concatenation and record deletion are also possible, along with other modifications enabled by the Region calculus.

Since the Mover is a self-contained object, independent of the rest of KeLP, the programmer can extend and replace Mover objects to realize different communication operations. For example, the KeLP distribution includes a Mover that provides message aggregation, concatenating multiple Region copies between a pair of physical processors into one message. The programmer may
also derive objects from Mover to perform numeric operations along with communication. For example, we have implemented a linear multigrid elliptic solver that requires the error on the coarse level to be interpolated and added to the solution on the fine level. The multigrid program derives a specialized object from Mover that copies and adds simultaneously, by overriding the Mover’s buffer packing virtual member functions. This facility allows efficient implementation of numeric operators by eliminating temporary storage and additional user-level buffer packing. From a software engineering viewpoint, the ability to customize Movers provides a convenient interface to encapsulate communication that accompanies a numeric operator.

2.3 Data Motion Examples

Using the Region calculus, it is easy to build MotionPlans to describe common collective communication patterns. For example, one common pattern that arises in finite difference calculations is \textit{fillpatch}, which fills in ghost cells with data from logically overlapping grids. Figure 5 shows the algorithm to generate the \textit{fillpatch} MotionPlan for an irregularly partitioned XArray. Another common pattern fills in ghost cells across periodic boundary conditions. Figure 6 shows the algorithm to generate a MotionPlan to satisfy periodic boundary conditions along one dimension of an irregularly partitioned XArray.

Additionally, the programmer may compose complicated collective communication patterns from simpler patterns. For example, suppose an application demands that ghost regions be filled in both between overlapping Grids and across periodic boundary conditions. Using the algorithms of Figures 5 and 6, the programmer may write independent subroutines to build the \textit{fillpatch} and \textit{periodic} communication MotionPlans and compose these two patterns into a single MotionPlan. With this mechanism, one may define a library of pre-defined MotionPlan generators which form building blocks for more complicated patterns. MotionPlan composition may also give better performance, since the composed MotionPlan may expose more opportunities for optimizations like message aggregation.

3 Programming Example

To better illustrate the KeLP model, we present excerpts from a simple KeLP program to solve an elliptic PDE. The example solves Laplace’s equation over a unit square using 5-point Gauss-Seidel iteration with Red/Black ordering. Although KeLP is intended for more elaborate computations,
we may readily generalize this simple example to more irregular computations.

We discretize the computation using the method of finite differences, solving a set of discrete
equations defined on regularly-spaced \( (N + 2) \times (N + 2) \) mesh. We number the \textit{interior}
points of the mesh from 1 to \( N \) in the \( x \)- and \( y \)-coordinates. The \textit{boundary}
points, which contain the Dirichlet boundary conditions for the problem, are located along \( x \)-coordinates of 0 and \( N + 1 \) and along \( y \)-coordinates of 0 and \( N + 1 \). The KeLP code for the main procedure appears in Figure 7.

3.1 Data Decomposition

The first step in implementing this example is to partition the global domain across processors and
allocate distributed data structures to hold the finite difference mesh.

We first define our global computational domain with an object of class \texttt{Region} (Figure 8). Statement (1) defines a 2d \texttt{Region} variable named \texttt{domain}. Class \texttt{Region} is strongly typed by the
number of spatial dimensions.

Next, we must decide how to partition the global domain across processors. That is, we must
set up a \texttt{FloorPlan} that holds the desired data decomposition. Recall that a \texttt{FloorPlan}
represents the \texttt{Regions} and processor assignments for a set of distributed \texttt{Grids}.

For complex irregular calculations, the programmer will generally implement an application-
specific partitioning policy using KeLP's Region calculus. Some partitioning strategies will apply
to many applications in a particular domain. For these cases, we may wish to reuse the partitioning
code in a Domain Specific Library (DSL) that implements policies germane to an application class.
For example, KeLP includes a DSL called \texttt{DOCK} (Decomposition Classes for KeLP) that implements
standard Fortran-D style block decompositions [19] for regular applications. DOCK includes two
classes, \texttt{Processor} and \texttt{Decomposition}, that implement first-class C++ versions of the virtual
processor arrays and templates in High Performance Fortran [20]. The DOCK classes are modeled
after similar classes introduced in pC++ [7].

Using the \texttt{DOCK} mapping classes, we first declare a virtual processor array. By default, the
\texttt{Processors2} object creates a 2D logical processor array corresponding to the physical processor set (2). Next, we must decompose the global domain across the virtual processor set. The \texttt{Decomposition} class, publicly derived from \texttt{FloorPlan}, creates a HPF-style \texttt{BLOCK} data distribution for a rectangular domain. Statements (3) and (4) create a \texttt{Decomposition} \( T \) representing a block data decomposition, mapped to the physical processor array. Figure 8a shows a 2D block
partitioning for 16 processors.

Next, we pad each processor's partition with one ghost cell in each direction to hold Dirichlet boundary conditions and off-processor values (Figure 8b). The loop in (5)-(7) accomplishes this. The for.1 loop iterates over each index in Decomposition T. For a Region R, grow(R,1) defines a Region padded with one cell in each direction. Thus, statement (6) replaces each partition in the Decomposition T with a Region grown one cell larger in each direction.

Now that we have defined the data partitioning, we are ready to instantiate the actual storage in an XArray. Statement (8) instantiates an XArray using the structure we have set up in Decomposition T. Statement (9) sets up initial boundary conditions and an initial guess for the unknowns. These details are straightforward and are not discussed here.

### 3.2 Data Parallel Execution

Statements (11-17) contain the main loop of the iterative solver. Function fillGhost() fetches off-processor values for each Grid(Figure 8c). Using the for.all loop, we iterate in parallel over each Grid in the XArrays (13-15). The sweep() subroutine (14) is a serial smoother that performs red-black relaxation on a single Grid on one processor. Thus statement (14) will invoke sweep() on each processor, with each invocation independent of the others. The serial smoother will often call an extrinsic programming language like Fortran 77. Since separately compiled Fortran modules do not understand the KeLP class system, KeLP provides an external interface to export descriptors of KeLP data to Fortran subroutines. Interfaces to other extrinsic languages are possible. Some KeLP users use extrinsic numerical routines in C, and a simple interface to an HPF compiler has been implemented by Merlin and Baden [29].

### 3.3 Data Motion

The fillGhost() function (Figure 9) fills in the ghost cells for each processor's partition in an XArray (Figure 8c). We now describe the KeLP code to effect this data motion.

First we declare a MotionPlan data structure to hold the communication pattern (1). Lines (2-9) implement the pseudo-code algorithm of Figure 5. We iterate over each Grid in the XArray X (2). For each Grid X(i), first we determine the Region inside of X(i) that corresponds to virtual processor i's portion of the solution. The grow() operation (3) trims off the ghost cell padding.

We then loop again over each Grid X(j) in X (4). For each Grid other than X(i), we add
to the MotionPlan the following block copy operation: “copy from $X(i)$ into $X(j)$ where these Grids intersect the Region inside” (6). As a consequence of the data layout, this MotionPlan corresponds to copying “solution” values from Grid $X(i)$ into ghost cells on $X(j)$.

After specifying all block copy operations in the MotionPlan we are now ready to perform the communication pattern. We instantiate a Mover object to interpret the MotionPlan $M$, sending from $XArray X$ to (the same) $XArray X$ (10). The execute() function performs the data motion as an atomic operation (11).

In this simple example we recompute the MotionPlan on each call to fillGhost(). In practice, we have found that on current architectures, the message-passing overhead dominates the time to build the MotionPlan. However, for especially complex patterns, the programmer may choose amortize this cost by storing the MotionPlan for re-use.

3.4 Discussion

This example used both the DOCK partitioning library and KeLP primitives to implement a simple numerical application. DOCK is an example of a Domain Specific Library(DSL), a library encapsulating policies for a specific application class. DOCK, in particular, provides data partitioning policies for applications which use regular block decompositions.

KeLP has simplified construction of DSLs for several application classes, including dense blocked linear algebra algorithms, structured adaptive mesh refinement, and partitioning for non-uniform computational environments. Using layered software design, the programmer can rely on DSL facilities where appropriate, and then “drop down” to the KeLP level in order to perform application-specific optimizations beyond the scope of a DSL.

It is also possible to drop down even lower, to bare message-passing, where needed. For example, a code that mixes structured and unstructured representations could use KeLP to manage the structured representations, and drop to message-passing to manage unstructured data. In the current implementation, KeLP implements a program with SPMD MPI processes. Thus, the current implementation allows the programmer to freely mix KeLP and MPI calls, using MPI communicators to separate message contexts. In general, many sticky issues regarding parallel library and language interoperability remain unaddressed, and deserve more attention in future work.
4 Implementation

KeLP’s abstractions permit implementation of communication based on the inspector/executor paradigm [1]. The Mover class performs run-time processing of the communication pattern stored in a MotionPlan. That is, the Mover analyzes the MotionPlan and generates message-passing calls to realize the communication pattern. We compare this implementation strategy to a model without inspector/executor analysis.

LPARX [24], the predecessor of KeLP, implements an asynchronous communication strategy without run-time communication analysis. Whereas KeLP and LPARX employ the same coarse grain data parallel computation model, the two systems differ on where data motion operations may occur. In LPARX, each process may initiate block copy operations from within a for all loop. Since each SPMD process does not execute all iterations of the loop, process $i$ does not know the communication activity generated by process $j$ if $i \neq j$. Thus, block copy operations must be implemented with one-sided communication mechanisms [31]. To ensure memory consistency with one-sided communication, at the end of the for all loop all processors participate in a global synchronization point. The barrier ensures that all block copies complete before the program may proceed and potentially modify Grid data.

This asynchronous one-sided communication implementation introduces several overheads on message-passing multicomputers. First there is the direct cost of synchronization; a barrier synchronization on current message-passing multicomputers typically takes several hundred microseconds. Additionally, unnecessary global synchronization may exacerbate load imbalance and increase network congestion by forcing all processes to run in close synchrony even when then program’s data dependencies do not require it. Furthermore, data transfers conditionally initiated by one partner without the other’s knowledge preclude run-time preprocessing of the communication pattern. This prevents some optimizations such as posting for contiguous messages directly into user data structures.

KeLP eliminates the need for one-sided communication by disallowing block copy operations from inside a for all loop. Instead, all block copy operations are treated as collective communication. Following the inspector/executor paradigm [1], all processors store a locally relevant piece of the distributed communication pattern (MotionPlan). No costly global synchronization is required to detect termination of data movement.

In the current Mover implementation, each processor generates MPI non-blocking messages
based on the MotionPlan information stored locally. First, the Mover allocates contiguous buffers for incoming and outgoing messages. If data to be communicated lies contiguously in memory, the Mover avoids unnecessary buffer-packing. Then, the Mover sends all outgoing messages with non-blocking MPI point-to-point messages. While these messages are in flight, the Mover polls for incoming messages, and delivers and unpacks incoming messages in the order they arrive. Finally, the Mover waits for outgoing messages to finish, and recycles message buffers. The Mover will perform message aggregation to minimize the number of message starts.

Since the Mover is a self-contained object, the implementation could be optimized for specialized communication hardware. For example, we have experimented with an implementation that performs user-level flow control to reduce conflicts in an ATM network. Another possible implementation could generate MPI all-to-all communication calls to realize a MotionPlan pattern. Finally, the pack and unpacking functions can be redefined to perform numeric operations or message compression. By re-implementing the Mover, the KeLP programmer can reuse the communication optimization code in all applications, without modifying application-level code.

5 Performance Results

To assess the efficiency of the KeLP primitives, we present performance results for one irregular application and several regular applications.

KeLP's abstractions present a much higher level programming model than bare message-passing. To evaluate the performance penalty of using the higher-level KeLP abstractions, we compare KeLP performance against message-passing for three regular codes. In all cases, KeLP performance is comparable to the original MPI codes.

The next application, an adaptive multigrid eigenvalue solver, exercises the full expressive power of KeLP's abstractions. This application requires five different types of communication patterns between dynamic irregular collections of grids. We show that KeLP's inspector/executor analysis substantially improves performance compared to a LPARX [24] implementation relying on one-sided communication.

Finally, using a uniform finite difference method, we show how to optimize MotionPlans using KeLP primitives to reduce communication costs. In this example, high-level tuning of the MotionPlan reduces communication costs by a up to factor of two.
5.1 Comparison to MPI

In this section, we assess KeLP performance by direct comparison with hand coded applications in MPI. In contrast to irregular calculations such as adaptive mesh refinement, these applications are fairly simple and regular, and thus amenable to programming with the low-level message-passing primitives of MPI.

The methodology used in this comparison is as follows. We obtained three publicly available block-structured MPI codes for reference. The first two codes, NAS-FT and NAS-MG, are application kernels from the NAS Parallel Benchmark MPI implementations version 2.1 [3]. The third code, the Scalable Universal Matrix Multiply Algorithm (SUMMA), is the matrix multiply kernel appearing in [40].

For each code, we translated the MPI version into KeLP by allocating the distributed data structures using KeLP, and replacing all MPI communication with KeLP primitives. We did not change basic implementation policies, such as the choice of data decomposition. We lifted the numerical kernels verbatim, so any differences in running times between the two versions are due to KeLP overheads in managing data layout and data motion.

Table III compares the KeLP performance with the MPI reference codes on between 8 and 64 thin nodes of an IBM SP2. (Due to memory constraints, we can not run the NAS-FT Class A benchmark on fewer than 8 nodes.) In all cases, KeLP performance is comparable to the original MPI codes.

The NAS-MG benchmark solves a Poisson equation using a multigrid solver. We report results from the class B benchmark: a 256x256x256 grid. On fewer than 64 processors, the KeLP run-time system is able to avoid some user-level buffer packing on faces that lie contiguously in memory. The NPB2.1 code does not recognize this situation, so KeLP performs better. As we move to 64 processors, KeLP can avoid less buffer packing, resulting in performance more comparable to the MPI version.

The NAS-FT benchmark solves a 3D partial differential equation using 3D Fast Fourier transforms. We report results from the class A benchmark: a 256x256x128 grid. The main communication activity in the NAS-FT code is a global matrix transpose. The NPB2.1 code implements the transpose with an MPI collective operation. The KeLP implementation cannot recognize a collective transpose, and performs the global transpose with point-to-point messages. This accounts for the slight performance difference between the KeLP and MPI versions.
to appear in *J. Parallel and Distributed Computing*

The SUMMA code performs a matrix multiplication using the pipelined, blocked algorithm of [40]. We scaled the size of the matrices with the number of processors, so each processor owned 250,000 elements of each matrix. The SUMMA algorithm relies on a pipelined broadcast of matrix block rows and columns. Once again, the KeLP run-time system optimizes communication performance by avoiding buffer-packing for contiguous data structures.

All these codes leave room for performance improvements through tuning. However, we believe that the reference MPI codes represent typical performance for portable message-passing codes. These results show that although KeLP provides a more abstract, simplified programming model, performance sacrifice is negligible. Also note that although these codes are regular, the underlying KeLP implementation treats the data with more general mechanisms for irregularly blocked data. Thus, we may confidently expect similar KeLP overheads even when the underlying problem structure is irregular.

5.2 Adaptive Multigrid

**lda3d** is a 3D adaptive multigrid eigenvalue solver used in *ab initio* materials science simulations [24, 9]. This application was originally coded in LPARX and then ported to KeLP. The **lda3d** algorithm demands complex irregular communication to transfer data within and between levels of the mesh hierarchy (see Figure 1b). MotionPlans and the Region calculus provide a natural paradigm to describe these data motion patterns.

Since the **lda3d** code is very complex, even using high-level KeLP abstractions, it was impractical to attempt an implementation using bare message-passing. In order to evaluate KeLP overheads on **lda3d**, we instead compare the KeLP performance to performance in LPARX. As discussed in Section 4, LPARX relies on a one-sided communication implementation that introduces unnecessary synchronization. By comparing KeLP to LPARX, we can evaluate the efficacy of run-time communication analysis to improve performance by relaxing synchronization.

Figure 10 shows performance results for **lda3d** on the Intel Paragon and IBM SP2. The results show that KeLP’s data orchestration implementation reduces communication time by 40% to 70% compared to LPARX. On the SP2, **lda3d** is communication bound and the KeLP version reduces overall running time by up to 40%. On the Paragon, communication is not the bottleneck and although KeLP reduces communication overheads significantly, Amdahl’s law limits overall application improvement to approximately 10%.
In Figure 10, load imbalance is measured as the time a processor must wait for other processors when entering a communication phase. KeLP’s relaxed synchronization requirements expose load imbalance due to unequal communication loads as a significant problem in 1da3d. Depending on the partitioning algorithm, some processors may incur more message-passing activity than others.

In LPARX, all processors synchronize at the end of a communication phase with a barrier, and the processes exit the communication phase in very close synchrony. Any variance in running time due to unequal communication loads increases LPARX communication time in Figure 10. Since KeLP does not introduce any unnecessary synchronization, the communication load discrepancy shows up as KeLP load imbalance in Figure 10. The KeLP implementation exposes a problem in the partitioning algorithm, since it poorly balances communication load. A better partitioning algorithm might improve scalability, and is a subject for future research.

5.3 Jacobi

The previous section evaluated KeLP performance without any special tuning of the KeLP primitives. We now use a simple, static regular application to illustrate tuning of communication performance using high-level KeLP primitives. The application is jacobi3d, a seven-point stencil relaxation which solves Laplace’s equation over a uniform 3D grid. The 3D grid is distributed blockwise on each axis. The structure of the jacobi3D code is similar to the example code of Section 3.

Since the Jacobi algorithm is relatively simple, it was possible write an optimized hand-coded version that carefully manages message-passing efficiently. Some communication optimizations performed in the hand-coded version are:

- Deliver and unpack messages asynchronously in the order they arrive.
- Send contiguous faces in place to avoid buffer packing.
- Prepost for incoming messages one iteration in advance.

We developed three KeLP versions of the jacobi3d application. All three versions reuse the MotionPlan. The first version uses the fillpatch routine listed in Figure 5 to generate the MotionPlan. Note that this generic fillpatch pattern fills in all ghost cells, communicating with up to 26 neighbors. However, for the jacobi3d relaxation, we only need ghost cells from the 6 Manhattan
neighbors—no corners or edges (see Figure 11). The second KeLP version builds a MotionPlan which only fills in ghost regions from Manhattan neighbors.

When communicating faces of a 3D array, user-level buffer packing and unpacking contribute substantially to communication overheads. However, note that two of the six faces generated by the Manhattan pattern are nearly contiguous. Growing these face Regions by one cell, they become contiguous regions of memory which permit the KeLP run-time system to avoid user-level message buffer packing and unpacking (see Figure 12). Thus, we implemented a third version which obtains the Manhattan pattern from version 2, and then directly modifies the MotionPlan to create contiguous faces wherever possible. The pseudocode for this direct schedule modification is shown in Figure 12c. For the 4x4x2 processor arrangement used here, the buffer-packing optimization reduces the volume of user-level copying by a factor of 2.

Figure 13 compares the running times of these three KeLP versions with the hand-coded version. The Figure shows that compared to version 1, the MotionPlan modifications in version 3 reduce communication costs by 55% on the Intel Paragon and 25% on the IBM SP2. The buffer-packing optimization is particularly effective on the Paragon, where buffer-packing is relatively expensive compared to message-passing. The buffer-packing has less impact on the SP2, with faster processors and slower inter-node communication. In all cases the KeLP code achieves performance within 15% of the hand-coded version. The 15% overhead is due mostly to the fact that the hand-coded version exploits the iterative program structure to pre-post for asynchronous messages one iteration in advance. Pre-posting in advance for asynchronous messages reduces operating system overhead in buffering incoming messages. A future version of KeLP may support this type of optimization.

We note that considerable effort went into tuning the hand coded version here to use message-passing primitives efficiently. Such tuning effort may be too time-consuming for some programmers. We speculate that in most cases an optimized KeLP implementation will be at least as efficient as a straightforward hand-coded message-passing version.

6 Related work

KeLP combines the structural abstraction concepts of LPARX [24] and inspector/executor communication analysis [16, 1]. The structural abstraction model uses geometric abstractions to describe data layout and data motion in irregular dynamic structured applications [24]. Structural abstraction derives from the domain abstractions used in the FIDIL programming language [21]. LPARX
uses an asynchronous communication model, as do several other parallel languages and libraries [15, 23, 7, 11, 10]. An asynchronous communication model is most appropriate for applications where run-time preprocessing of the communication pattern is impossible. However, when communication dependences can be computed beforehand, run-time preprocessing can significantly reduce communication overheads and synchronization.

Structural abstraction also appears in software targeting structured adaptive finite difference solvers. Rendleman et al. have developed BOXLIB, a library for implementing adaptive mesh refinement on single processor computers [38]. Parashar and Browne employ similar techniques in a DAGH, a software infrastructure for parallel structure adaptive mesh refinement applications [34].

KeLP’s block-structured inspector/executor implementation is based on concepts introduced in the Multiblock PARTI [1] run time system. Multiblock PARTI supports regular block distributions for dynamic arrays, but does not directly support irregular block decompositions as in systems like KeLP. Multiblock PARTI provides two common communication patterns, one to fill in ghost regions and one that moves data over regular sections. KeLP differs from Multiblock PARTI in that KeLP exposes communication management to the programmer, who can then describe arbitrary communication patterns with high-level geometric operations. The Multiblock PARTI communication schedule is an opaque object over which the user has limited control. In contrast, KeLP exposes the schedule to the programmer as a first class mutable object, which may be manipulated, modified and interpreted according to the programmer’s needs.

The design of the PETSc library [4] follows some of the same principles used in KeLP. PETSc provides first-class objects to represent index sets that describe gather/scatter operations, and performs inspector/executor analysis of communication patterns. PETSc does not support irregular grid collections nor irregular partitionings of rectangular domains.

Run-time scheduled communication mechanisms have proven useful in several other contexts. The PASSION [14] and Jovian [36] systems pre-process communication patterns to optimize parallel I/O. Communication schedules have traditionally been considered for message-passing architectures. However, recent work [32, 17] has shown that inspector/executor techniques may be necessary to achieve good performance on distributed shared-memory architectures as well. Tempest [37] allows the programmer to write application-specific communication protocols for a distributed shared-memory architecture. One interesting Tempest protocol builds a communication schedule transparently, and reuses it where appropriate.
KeLP applies to block-structured applications, where the arrangement of blocks and communication patterns between blocks are potentially irregularly. KeLP does not support fine-grained data structures and communication patterns, as supported by systems such as CHAOS [41], CHAOS++ [12], PETSc [4], pC++ [7], and Multipol [10]. KeLP is not appropriate for more irregular calculations such as those involving sparse arrays, lists, trees, or asynchronous parallelism.

7 Conclusion

We have introduced a small set of geometric programming abstractions for managing data layout and communication in dynamic block-structured applications. The abstractions allow the programmer to manipulate descriptors of data decompositions and communication schedules with first-class language objects. A distinguishing feature of our approach is the application of first-class, user-level communication schedules. Structural abstraction, pioneered in LPARX, provides a convenient geometric interface to hide the bookkeeping in communication schedule generation. An object-oriented implementation of the programming abstractions facilitates code reuse through Domain Specific Libraries.

KeLP supports structured applications that contain dynamic or irregular block partitionings or data structures. The ability to represent and interpret complex dynamic data motion patterns using data orchestration is vital in applications that employ computational adaptivity. To our knowledge, KeLP is the first library to apply run-time communication scheduling techniques to a dynamic multilevel structured application.

Performance results show that although KeLP provides a high-level programming interface, performance compares well to coding directly in MPI. In all cases considered, KeLP performs within 15% of codes utilizing message-passing directly. Experience with a structured adaptive mesh code shows that inspector/executor communication analysis significantly improves performance by relaxing synchronization requirements. Furthermore, KeLP mechanisms simplify tuning of communication performance with a concise, abstract interface.

KeLP’s abstractions fall into two categories: geometric structural descriptors and classes to interpret these descriptors. By changing the interpretation of the descriptors, KeLP can be retargetted to different computational models and hardware resources. In ongoing work, we are investigating applications of the KeLP abstractions to hierarchical programming on clusters of symmetric mul-
tiprocessors, coupled multidisciplinary applications, out-of-core algorithms, and coordination of multiple HPF invocations.

The KeLP software and documentation is available at http://www.cse.ucsd.edu/groups/hpcl/scg/kelp.html.

References


to appear in *J. Parallel and Distributed Computing*


to appear in *J. Parallel and Distributed Computing*


to appear in *J. Parallel and Distributed Computing*


Figure 1: a) An adaptive discretization of a hypothetical molecule (courtesy John Weare). b) Example of inter-level transfer operations required for structured adaptive mesh refinement.


set up distributed data structures
do i = 1, niter
    perform data motion
    for all(...)
        SMPD computation
    end for all
end

Figure 2: KeLP applications manage data decomposition and data motion in a single thread of control, but drop into concurrent SPMD processes to carry out numeric computation.
Figure 3: The XArray is a coarse-grained distributed array of blocks of data, whose structure is described by a FloorPlan. The blocks may have different sizes and each is assigned to a single address space.

Figure 4: The MotionPlan encodes a set of block copy operations between grids.
BuildFillpatch(XArray X, MotionPlan M)
begin
    for each $i \in X$
        $I = grow(X(i), -1)$
        for each $j \in X$
            if ($i \neq j$) {
                $R = I \cap X(j)$
                $M.Copy(X, i, R, X, j, R)$
            }
        end for
    end for
end

Figure 5: a) Pseudocode to generate a fillpatch MotionPlan $M$ to fill in ghost cells for XArray $X$.

b) The dark shaded regions represent ghost regions that are copied into the central Grid.

BuildPeriodic(XArray X, MotionPlan M, Region Clip)
begin
    for each $i \in X$
        $Ghost_R = X(i) \cap shift(\text{Clip}, [\text{extents}(\text{Clip}, 1), 0])$
        $Ghost_R = shift(Ghost_R, [-\text{extents}(\text{Clip}, 1), 0])$
        for each $j \in X$
            if ($i \neq j$) {
                $R_D = I \cap Ghost_R$
                $R_S = shift(R_D, [\text{extents}(\text{Clip}, 1), 0])$
                $M.Copy(X, j, R_S, X, i, R_D)$
            }
        end for
    end for
end

Figure 6: a) Pseudocode to generate a MotionPlan $M$ that fills in ghost cells for XArray $X$ with periodic boundary conditions around the domain represented by Region $\text{Clip}$. b) The central white Region represents the Region $\text{Clip}$. The dark shaded regions are copied to satisfy periodic boundary conditions.
(1) Region2 domain(1,1,N,N);
(2) Processors2 P;
(3) Decomposition2 T(domain);
(4) T.distribute(BLOCK,BLOCK,P);
(5) for i=1,i,T
(6) T.setregion(i,grow(T(i),i));
(7) end_for
(8) XArray2< Grid2<double> > U(T);
(9) InitGrid(U);
(10) int RedBlack = 0
(11) for (int k=0 ; k<NITERS*2; k++) {
(12) fillGhost(U);
(13) for_all(i,U)
(14) sweep(U(i),RedBlack);
(15) end_for_all
(16) RedBlack = 1 - RedBlack;
(17) }

Figure 7: Main procedure for Red/Black Gauss-Seidel example.

Figure 8: a) Block partitioning into 16 regions. b) Each Grid is padded with a layer of ghost cells to hold boundary conditions. c) Dependencies that must be satisfied to refresh the ghost cells.
void fillGhost(XArray2<Grid2<double>> & X)
{
    (1) MotionPlan2 M;
    (2) for_1(i,X)
    (3) Region2 inside = grow(X(i).region(), -1);
    (4) for_1(j,X)
    (5) if (i != j) {
    (6)    M.CopyOnIntersection(X,i,X,j,inside);
    (7)    }
    (8) end_for
    (9) end_for
    (10) Mover2<Grid2<double>, double> DM(X,X,M);
    (11) DM.execute();
}

Figure 9: FillGhost() function for code of Figure 7.

![Graph a) and b)](image)

Figure 10: Performance comparison for adaptive multigrid eigenvalue solver on a) IBM SP2 and b) Intel Paragon.
Figure 11: 2D ghost Region patterns. a) The general fillpatch pattern fills in ghost cells from all neighbors. b) For a 5 point stencil, only Manhattan neighbors are necessary.

Figure 12: a) The 3D Manhattan pattern transmits ghost faces that are nearly contiguous in memory. b) By growing these faces by one cell, we realize a block copy that is contiguous in memory and can avoid user-level copying. c) Pseudocode that directly modifies the Manhattan MotionPlan M to perform this optimization.
Figure 13: Comparison of several `jacobi3d` implementations on the a) Intel Paragon and b) IBM SP2. The first KeLP version uses the naive `fillpatch` schedule, the second uses a Manhattan neighbor pattern, and the third version is optimized to avoid buffer packing.
<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointD</td>
<td>&lt; int i₀, int i₁, ... int i_{D-1} &gt;</td>
<td>A point in $Z^D$</td>
</tr>
<tr>
<td>RegionD</td>
<td>&lt; PointD l, PointD h &gt;</td>
<td>A rectangular subset of $Z^D$</td>
</tr>
<tr>
<td>FloorPlanD</td>
<td>Array of &lt; RegionD Rᵢ, int p &gt;</td>
<td>A set of regions, each assigned to a processor $p$</td>
</tr>
<tr>
<td>MotionPlanD</td>
<td>List of &lt;$int f, RegionD R_f &gt; ,&lt;int t, RegionD R_t &gt;$</td>
<td>Block-structured communication pattern between two FloorPlans</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridD</td>
<td>A multidimensional array whose index space is a RegionD</td>
</tr>
<tr>
<td>XArrayD</td>
<td>An array of GridDs; structure represented by a FloorPlanD</td>
</tr>
<tr>
<td>MoverD</td>
<td>Object that atomically performs the data motion pattern described by a MotionPlan</td>
</tr>
</tbody>
</table>

Table I: A brief synopsis of the KeLP data types.
to appear in *J. Parallel and Distributed Computing*

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extents(RegionR,inti)</code></td>
<td>length of Region R along the ith axis</td>
</tr>
<tr>
<td><code>shift(RegionR,PointP)</code></td>
<td>translation of Region R by the vector P</td>
</tr>
<tr>
<td>Region $R \cap$ Region $S$</td>
<td>geometric intersection of Regions $R$ and $S$</td>
</tr>
<tr>
<td><code>grow(RegionR,PointP)</code></td>
<td>Region $R$ padded with $P(i)$ cells on ith axis</td>
</tr>
</tbody>
</table>

Table II: Region calculus operations used in the examples.
<table>
<thead>
<tr>
<th>Code</th>
<th>KeLP Performance (MFLOPS)</th>
<th>MPI Code Performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 nodes</td>
<td>16 nodes</td>
</tr>
<tr>
<td>NAS-MG Class B</td>
<td>311</td>
<td>558</td>
</tr>
<tr>
<td>NAS-FT Class A</td>
<td>111</td>
<td>203</td>
</tr>
<tr>
<td>SUMMA</td>
<td>1231</td>
<td>2452</td>
</tr>
<tr>
<td></td>
<td>Normalized KeLP Running Time (Normalized so MPI running time = 1.0)</td>
<td>Percentage of Time Spent Communicating in KeLP version</td>
</tr>
<tr>
<td></td>
<td>8 nodes</td>
<td>16 nodes</td>
</tr>
<tr>
<td>NAS-MG Class B</td>
<td>0.91</td>
<td>0.86</td>
</tr>
<tr>
<td>NAS-FT Class A</td>
<td>0.91</td>
<td>1.10</td>
</tr>
<tr>
<td>SUMMA</td>
<td>1.02</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table III: Comparison of KeLP performance with three MPI reference codes on an IBM SP2.