

On a multicomputer, communication interfaces like MPI [6] support many of the above communication algorithms. For example, MPI provides a data type mechanism to express regular sections with strides, communication domains to configure processor geometries, and a variety of global communication primitives. But MPI has limitations. Support for irregular problems is lacking, and irregular data structures incur high software overheads in the form of bookkeeping. In addition, the quality of MPI implementations varies widely, hindering efforts to write portable codes with good performance. This is true because development cost constraints may compel software providers to implement only a subset of MPI efficiently, or because the hardware may not support the required functionality [7].

For example, many of today's MPI installations do not understand MPI datatypes; the user achieves better performance by explicitly packing and unpacking data into contiguous buffers. Significantly, the MPI standard does not mandate the ability to overlap communication with computation—even if the hardware can support it—and many commercial implementations fail to do so. Furthermore, MPI-2 [8] does not define non-blocking collective communication, out of concern for imposing a hardship on the MPI implementer. Such capabilities are up to the system provider.¹ Thus, developers of numerical libraries who implement ubiquitous functionality like the FFT [9] must come up with a custom, non-portable solution. On multicomputers with SMP nodes, an alternative way to realize overlap is to employ mixed-mode programming involving threads and message passing [4] [10] [11]. However, the interactions of threads and message passing lead to complex programs, and many users lack the background or time to experiment with such advanced programming techniques.

The requirement that a communication library offer portability with performance poses difficult challenges. The problem becomes even more daunting when we consider that other communication interfaces besides MPI may be available, e.g. IBM's LAPI [13], the VIA standard [14], or even single sided communication. In the face of these myriad options, we cannot effectively deliver portability with performance relying on the native communication substrate. We argue that a higher level model is needed instead.

In this paper, we present a simple model of machine-independent communication called the *Data Mover*. The Data Mover model provides an interface for expressing customized collective communication patterns for block-structured scientific calculations. As a compiler or programmer target, the Data Mover model has three advantages over lower level communication models such as MPI:

1. *Abstraction*: the model supports a concise, high-level description of a communication pattern;
2. *Expressive power*: the model can express a larger set of collective patterns, without relying on code with point-to-point messaging; and
3. *Separation of concerns*: the programmer or compiler can separate the expression of correct programs from implementation policies affecting performance.

We have implemented the Data Mover as part of the KeLP framework. The Data Mover provides meta-data types and a geometric model for encoding data-dependence patterns arising in block structured computations. A meta-data object, called a MotionPlan, encapsulates these data-dependence patterns in a canonical form. A Mover object carries out the communication pattern stored in a MotionPlan. The Mover may run as a proxy on a multicomputer with SMP nodes, masking communication on architectures that do not support overlap via non-blocking message passing [4] [10] [16]. The Data Mover model supports application-specific optimizations through direct manipulation of communication meta-data, inspector-executor analysis [17], and object-oriented inheritance mechanisms. The Mover object hides implementation details that may work around limitations in the communication substrate or its implementation.

The rest of the paper is organized as follows. In the next section we describe the Data Mover communication model. Section 3 presents empirical results, and section 4 a discussion, including related work. Section 5 concludes the paper and suggests future research directions.

¹ Notably, IBM's MPI implementation provides non-blocking collective communication primitives, but the hardware is only able to realize overlap under restricted conditions.

2. The Data Mover Communication Model

The Data Mover has been implemented as part of the KeLP infrastructure. While a complete discussion of KeLP lies beyond the scope of this paper, we describe portions of the KeLP model relevant to the ensuing discussions. We refer the reader to a recent paper [10] or to the KeLP web site at <http://www.cse.ucsd.edu/groups/hpcl/scg/kelp.html>.

KeLP supports distributed collections of block-structured data, and their distribution across multiple address spaces. Some examples are shown in Fig. 1. At the core of KeLP are two capabilities: user-level meta-data, and a collective model of communication. To support this model, KeLP provides two kinds of abstractions: *meta-data* and *instantiators*, which are listed in Table 1. KeLP meta-data objects represent the abstract structure of some facet of the calculation, such as data decomposition, communication, or controlled execution under mask. Instantiation objects carry out program behavior based on information contained in meta-data objects; they allocate storage or move data.

KeLP supports two levels of control flow, collective level and node level, as articulated in the Phase Abstractions programming model [18] and embodied by SPMD programming with MPI. A program performs collective operations, such as reductions, barriers and broadcasts, interspersed within a node level program. The node-level instructions form separate threads of control and execute independently. In most cases, the node level will invoke highly tuned serial numeric kernels. KeLP meta-data, and most instantiator objects, may live at either the collective level or the node level. KeLP does not assume the existence of a global shared address space, though a clever implementation may take advantage of shared memory to improve performance.

Meta-data abstractions. There are four meta-data abstractions: the `Region`, `FloorPlan`, `Map`, and `MotionPlan`. The `Region` represents a rectangular subset of Z^d ; i.e., a regular section with stride one. KeLP provides the *Region calculus*, a set of high-level geometric operations to help the programmer manipulate `Regions`. Typical operations include `grow` and `intersection`, which appear in Fig. 2. The `Map` class implements a function $Map: \{0, \dots, k-1\} \rightarrow Z$, for some integer k . That is, for $0 \leq i < k$, `Map(i)` returns an integer. The `Map` forms the basis for node assignments in KeLP partitioning. The `FloorPlan` consists of a `Map` along with an array of `Regions`. It is a table of meta-data that represents a potentially irregular block data decomposition. An example of a `FloorPlan` is depicted in Fig. 3. The `MotionPlan` implements a dependence descriptor, e.g., a communication schedule. The programmer builds and manipulates `MotionPlans` using geometric `Region` calculus operations, a process to be described shortly.

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

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

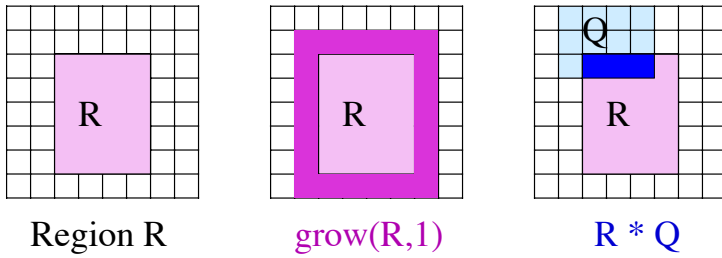


Figure 2. Examples of Region calculus operations `grow` and intersection `*`. `Q` and `R` are regions. `Grow(R, 1)` expands the Region `R` by one unit in all directions, and is commonly used to grow a halo region for stencil computations. We can intersect this halo region with the other subdomain to determine data dependencies requiring communication.

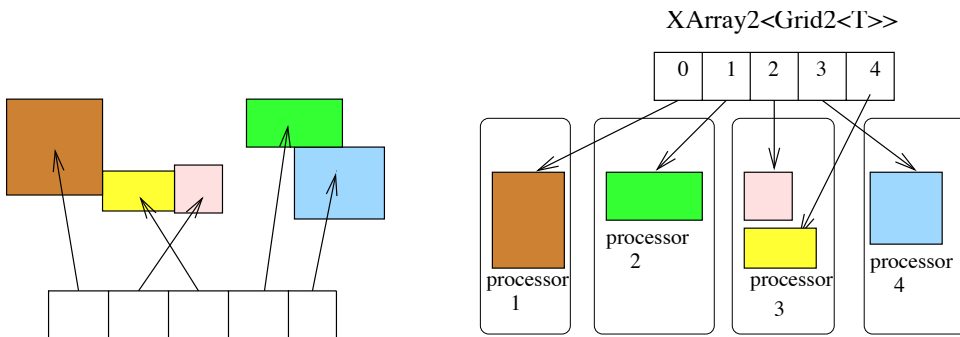


Figure 3. Example of a FloorPlan (left) and XArray (right). The arrows on the right denote processor assignments, not pointers.

Storage Model. KeLP defines two storage classes: `Grid` and `XArray`. A `Grid` is a node-level object which lives in the single address space of a single node. This restriction is made in the interest of efficiency, since KeLP does not assume the existence of a global shared memory.² A `Grid` is like a Fortran 90 allocatable array. For example, the Fortran 90 array `real A(3:7, -5:-1)` corresponds to `Grid2<real> A3` and has a region `A.region() = [3:7, -5:-1]`.

An `XArray` is a distributed array of `Grids` whose structure is represented by a `FloorPlan`. All elements must have the same number of spatial dimensions, but the index set of each `Grid` component (a KeLP Region) can be different. `Grid` components can have overlapping Regions, but they do not share memory. The application is responsible for managing any such sharing explicitly, and for interpreting the meaning of such overlap. An `XArray` is a collective object, and must be created from the collective program level using a `FloorPlan`. The number of `XArray` elements may be greater than the number of physical nodes or processors, and the mapping of `XArray` elements to processors may be many-to-one. This capability is useful in managing locality when load balancing irregular problems.

The Data Mover. The Data Mover supports a collective communication operation that performs block transfers of regular array sections between two `XArrays`. Two abstractions are used to express communication: the `MotionPlan` and `Mover`. There are three steps to program communication with the model: 1) Build a `MotionPlan` describing the collective data dependence pattern, 2) Construct a `Mover`

² A `Grid` that spans multiple nodes would, in effect, define a shared-memory address space across all nodes.

³ KeLP `Grids` are implemented in C++ as a templated class, and are typed by the number of dimensions, which is added as a suffix to the keyword `Grid`.

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 order (e.g., column-major order).

A MotionPlan encodes a set of dependencies of the above form. The MotionPlan M is a list containing n entries, denoted $M(k)$ for $1 \leq k \leq n$. 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. 4.

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

$$\text{Mover } \mu = \text{Mover}(M, S, D)$$

The result is an object μ 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 $M(k)$:

$$(D(M(k) \# i) \text{ on } M(k) \# R_d) \Leftarrow (S(M(k) \# j) \text{ on } M(k) \# R_s),$$

where we select the entries of the $M(k)$ 4-tuple with the `#` qualifier: $M(k) \# R_d$, $M(k) \# i$, 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. An example of MotionPlan construction is shown in Fig. 4.. A sample application kernel appears in Fig. 5.

```

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

```

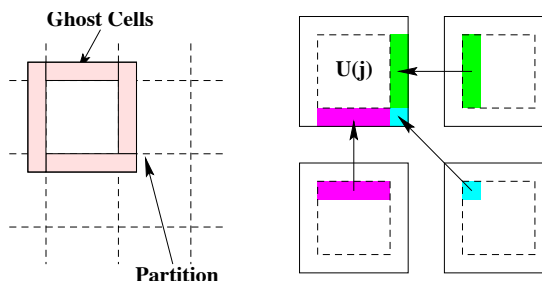


Figure 4. MotionPlan construction for a halo update (left). The grow() operation trims off the ghost cells (right), which are NGHOST cells deep. A graphical depiction of MotionPlan construction appears on the right.

```

( 1 ) FloorPlan3 F = BuildFloorPlan( );
( 2 ) XArray3<Grid3<double>> U(F);
( 3 ) MotionPlan3 M = BuildMotionPlan( );
( 4 ) Mover3<Grid3<double>, double> Mvr(U, U, M);
( 5 ) while (not converged) {
( 6 )   Mvr.start( );
( 7 )   Mvr.wait( );
( 8 )   for ( nodeIterator ni(F); ni; ++ni ) {
( 9 )     int i = ni( );
(10 )     _serialRelax( U(i) );
  }

```

Figure 5. A KeLP coding example: 3D iterative solver. MotionPlan construction is shown in Fig. 4.

Implementation. The KeLP infrastructure implements all the above abstractions as first-class C++ objects. The underlying Mover implementation uses MPI point-to-point asynchronous messages to carry out the indicated communication. KeLP defines its own private MPI communicator, and may interoperate with other software that uses MPI.⁴ KeLP is built on 16 MPI calls:

1. Send, Recv, Irecv, Probe, Iprobe, Wait, Test
2. Barrier, Allreduce, Comm_free, Comm_dup, Comm_size, Comm_rank
3. Type_size, Errhandler_set, Get_count

The calls in group (1) provide data transport. The calls in group (2) manage KeLP’s private communicator, and the calls in group (3) are used in an optional safety-checking mode.

In a typical KeLP program, the user constructs one or more FloorPlans, which are later used to build XArray. FloorPlans and XArrays are replicated on all processors. However, the data in an XArray is not replicated, only the meta-data. A presence flag determines whether or not a process actually has a copy of the data for a particular Grid. This solution is ultimately non-scalable, since the replicated meta-data storage grows as the square of the number of nodes. However, in practice, the meta-data is a small fraction of the total storage, and we have not encountered any difficulties.

The Mover implementation deserves special attention, since it performs inspector/executor analysis of the data motion pattern, and issue message-passing calls and memory copies to effect data motion.

⁴ KeLP applications commonly call Fortran in order to improve performance of the numerical code. In theory, KeLP may be invoked from an extrinsic language like Fortran, but this hasn’t been tested.

The Mover constructor does some preliminary communication analysis, determining which data transfers may be satisfied locally, and which must be transmitted via a message. The remaining analysis is triggered by calls to the `start()` and `wait()` member functions, and falls into two phases.

Phase 1 performs the following operations on each node:

1. Allocate message buffers for incoming and outgoing messages.
2. Post for incoming non-blocking MPI messages from other nodes.
3. Pack outgoing data into message buffers, where necessary.
4. Send non-blocking MPI messages to other nodes.
5. Perform local memory-to-memory copies for XArray elements assigned to the same process.

Phase 2 performs the remainder. While incoming messages are expected:

1. Poll until an incoming message arrives.
2. Unpack the detected message data into the target XArray element.
3. Free the message buffer.

By default, the implementation avoids buffer-packing for contiguous data. That is, if source or destination data happens to lie contiguously in memory, then the Mover will send or receive the data directly from the original storage locations. Using object-oriented inheritance mechanisms we have derived specialized Movers that perform message aggregation, combining multiple messages with a common destination; message packetization, to handle flow control; or update forms of data transport, that perform an operation like addition while moving the data. These specialized Movers meet specific needs of an application or an architecture, and can work around defects in the underlying communication substrate.

Notably, the KeLP Mover allocates all message buffers dynamically—at the time the call to the `start()` member function is made. It frees the buffers when the call to `wait()` completes. This strategy was designed to avoid tying up temporary storage. It imposes a message start overhead that would not be seen in a hand-coded MPI implementation that allocated buffers statically, but does not affect peak bandwidth [4]. As will be seen in the next section, this overhead is significant only when the problem size is small relative to the number of processors, and does not noticeably affect coarse-grained applications. Alternatively, one could provide an alternative Mover constructor that preallocated the message buffers, which would be freed by the Mover destructor. Such a Mover implementation might be useful in a fine-grained application, but would not change the programmer’s model.

As mentioned previously, the KeLP implementation supports asynchronous execution of the Mover activity by providing the `start()` and `wait()` member functions, which asynchronously initiate and rendezvous collective communication. Different invocations of the `start()` member function are guaranteed not to interfere so long as they do not incur any race conditions. For example, the following code

```
Mover m1 = ..., m2 ...;
m1.start();
m2.start();
m1.wait();
m2.wait();
```

executes correctly so long as the Movers do not access common data. (Even a synchronous Mover invocation can incur race conditions if some MotionPlan entries overlap.)

The implementation realizes communication overlap with either MPI asynchronous messages or by dedicating a proxy thread exclusively to communication. The proxy thread approach is particularly effective on multicomputers with SMP nodes, since the implementation can temporarily assign a physical processor to the Mover, without directly impacting other processors. A distinguished user thread on each node communicates with the proxy using shared queues; see Fink’s Ph.D. dissertation [4] for additional details.

3. Results

In this section we provide empirical evidence demonstrating that the convenience of the Data Mover’s higher-level abstraction does not come at the expense of performance. The performance of KeLP applications are often competitive with equivalent versions written with MPI. We look at four applications: *Redblack3D*, Red Black Gauss-Seidel relaxation in three dimensions on a 7-point stencil; two NAS benchmarks [20], *NAS-MG*, which solves the 3D Poisson equation using multigrid V cycles, and *NAS-FT*, which solves the time-dependent diffusion equation in three dimensions using an FFT; and *SUMMA*, which performs blocked matrix multiply [21].

The applications we have chosen represent a diverse range of communication patterns encountered in block-structured applications. *Redblack3D* carries out halo updates; *NAS-MG* carries out halo updates with periodic boundary conditions and prolongation and restriction operations between multigrid levels. *NAS-FT* carries out a massive 3D array transpose (total exchange), and *SUMMA* broadcasts horizontal and vertical slices of data within rows and columns of a two-dimensional processor geometry.

The KeLP applications were written in a mix of C++ and Fortran 77. KeLP calls were made from C++, and all numerical computation was carried out in Fortran. The hand-coded versions were implemented somewhat differently from their KeLP equivalents, except for *RedBlack3D*. *Redblack3D* had a similar organization to the KeLP version, allocating storage in C++, and performing numerical computation in Fortran. It also allocated all message buffers dynamically. The NAS benchmarks, v2.1, were written in Fortran only. All message buffers and other data structures were allocated statically, as the problem size and number of processors had to be fixed at compile time (By contrast, the KeLP versions input the problem size and number of processors at run time.). *SUMMA*[21] was written entirely in C, but invoked a vendor supplied matrix multiply routine (`dgemm`)⁵. All storage was statically allocated.

We report results on an IBM SP2 with 160 MHz Power-2 SC thin nodes (256 Megabytes of memory per node), running AIX 4.2, located at the San Diego Supercomputer Center. We compiled C++ using `mpCC`, and compiler options `-O -Q -qmaxmem=-1`, and compiled Fortran using `mpxlf`, and compiler options `-O3 -qstrict -u -qarch=auto -qtune=auto`. We used KeLP version 1.2.95. We also report results on the IBM ASCI Blue Pacific CTR machine, running AIX 4.3.1, located at Lawrence Livermore National Laboratory. On this machine, each node is a 4-way SMP based on 332 MHz Power PC 604e processors, sharing 1.5 Gigabytes of memory. We compiled C++ with `mpCC_r`, using the compiler options `-O3 -qstrict -Q -qmaxmem=-1 -qarch=auto -qtune=auto`, and compiled `f77` using `mpxlf_r` and compiler options `-O3 -qstrict -u -qarch=auto -qtune=auto`.

On the SDSC SP2 we ran with the following problem sizes. We ran *Redblack3D* on a fixed size 360^3 mesh, *NAS-MG* on a 256^3 mesh, *NAS-FT* on a $256 \times 256 \times 128$ mesh. The *SUMMA* computations multiplied square matrices. We linearly increased the amount of *work* with the number of processors, such that number of floating point operations per node was fixed at approximately 3.4×10^8 (When we double the number of processors we increase the linear dimension of the matrix by $2^{1/3}$).

On the SDSC SP2, the performance of the applications running with the KeLP Mover are competitive with the equivalent MPI encodings. These results are shown in Fig. 6, and corroborate previous findings on the Cray T3E [4]. With the exception of *NAS-FT*, the performance with KeLP and hand-coded MPI are roughly comparable, demonstrating that the Data Mover’s higher level of abstraction does not substantially hinder performance.

We note that KeLP enjoys a slight performance advantage with *RedBlack3D*, which diminishes as the number of processors increases. At 64 processors, the MPI version runs slightly faster, taking less time to communicate than the KeLP version. Since both the hand-coded and KeLP version of *RedBlack3D* allocate message buffers dynamically, the difference in performance is attributed to miscellaneous overheads in the

⁵ We used IBM’s `essl` library to provide `dgemm`.

KeLP run time library, and to the differing dynamic communication patterns generated by the respective codes. KeLP makes no assumptions about the geometry of the underlying data decomposition, whereas the MPI implementation is aware of the processor geometry. Thus, the MPI code issues sends and receives in the X, then Y, and then Z direction. The KeLP implementation issues sends in order of increasing process number, and surprisingly edges out the MPI implementation when granularity isn't too fine. The second factor is due to miscellaneous KeLP overheads, which are involved in setting up Movers and MotionPlan, exclusive of storage allocation. This effect comes into play for small task granularity at 64 processors. (Local computation times are always smaller in KeLP, but the effect isn't significant.)

Performance in NAS MG is comparable. (Unfortunately, we were unable to break out the computation time in the hand-code NAS version) This code employs mesh sweeps like Redblack3D, but there are major differences between the two codes that account for a slight differences in performance. The multigrid algorithm uses a hierarchy of dynamic grids ($\log N$, where N is the linear mesh dimension), in which each successive mesh in the hierarchy has one-eighth the number of points as its predecessor. KeLP allocates these arrays dynamically, while the hand-coded Fortran version allocates them statically. This affects the running time of computation. In addition, task granularity decreases sharply at higher levels of the mesh hierarchy, since meshes at successive levels become increasingly smaller. Message sizes decrease sharply at the higher mesh levels, resulting in higher message overheads in KeLP. Nevertheless, the running times are still close.

Performance for the KeLP implementation of NAS-FT lags behind the MPI. We attribute this to a naïve implementation of a massive 3D transpose. The MPI version uses `All_to_all`, whereas KeLP uses a point-to-point call for each pair of communicating tasks. Perhaps we could improve the KeLP transpose using a more efficient algorithm when running on larger number of processors, or by implementing a KeLP Mover that recognizes the transpose pattern and called the system-supported collective routine. Performance of the KeLP and MPI implementations of SUMMA are comparable. We use a non-scaling, linear time broadcast algorithm, and the effect can be seen as we increase the number of processors. Again, our naïve point-to-point messaging implementation is surprisingly competitive, and could probably be improved when we ran on hundreds of processors.

The Mover interface simplifies the implementation of more complex algorithms. For example, we have restructured each of these applications to overlap communication with computation, in order to improve utilization of the SMP nodes. While a full discussion of these restructured algorithms[4] is beyond the scope of this paper, we present results from Redblack3D on the ASCI Blue-Pacific CTR. We ran with cubical domains, scaling the problem with the number of nodes to maintain a ratio of approximately 2 million unknowns per node. We ran with the KeLP2.0 prototype, which uses a proxy implemented as an extra thread to handle communication. Preliminary results reveal that the overlapped KeLP2 Mover is able to improve performance significantly. The use of communication overlap improves performance by as much as 19%, on 64 nodes (256 processors). Results are shown in Fig. 7. These results corroborate similar findings by Fink running on a cluster of Digital AlphaServer 2100 SMPs and on a cluster of UltraSPARC SMPs [4]. (The overlapped Mover was also observed to improve the performance of the other three benchmarks discussed earlier in this section.)

4. Discussion

Our experiences with the KeLP Data Mover have been positive, and indicate this higher-level abstraction is able to match the performance of hand-coded MPI, and in some cases exceed it. The Mover's versatility in supporting communication overlap on an SMP-based illustrates the importance of separating correctness concerns from policy decisions that affect performance.

The Data Mover resembles an MPI persistent communication object. However, KeLP provides inspector-executor analysis, which is particularly useful in irregular problems, and it also provides first-class support for multidimensional arrays, via user-defined metadata and a geometric region calculus. KeLP avoids MPI awkward data type mechanism to handle strides of non-contiguous faces that would entail registering a separate data type for each stride appearing in the data. More generally, the KeLP meta-data abstractions,

e.g. the Region calculus, provide a more intuitive and concise notation for expressing and managing customized communication domains.

The convenience of this notation has been demonstrated in a variety of full scale applications that use the Mover: a multiblock mortar-space method for subsurface modeling⁶, structured adaptive mesh refinement for first principles simulation of real materials [22], and Direct Numerical Simulation for turbulent flow [23]. The adaptive mesh refinement and multiblock codes employ irregular collections of meshes.

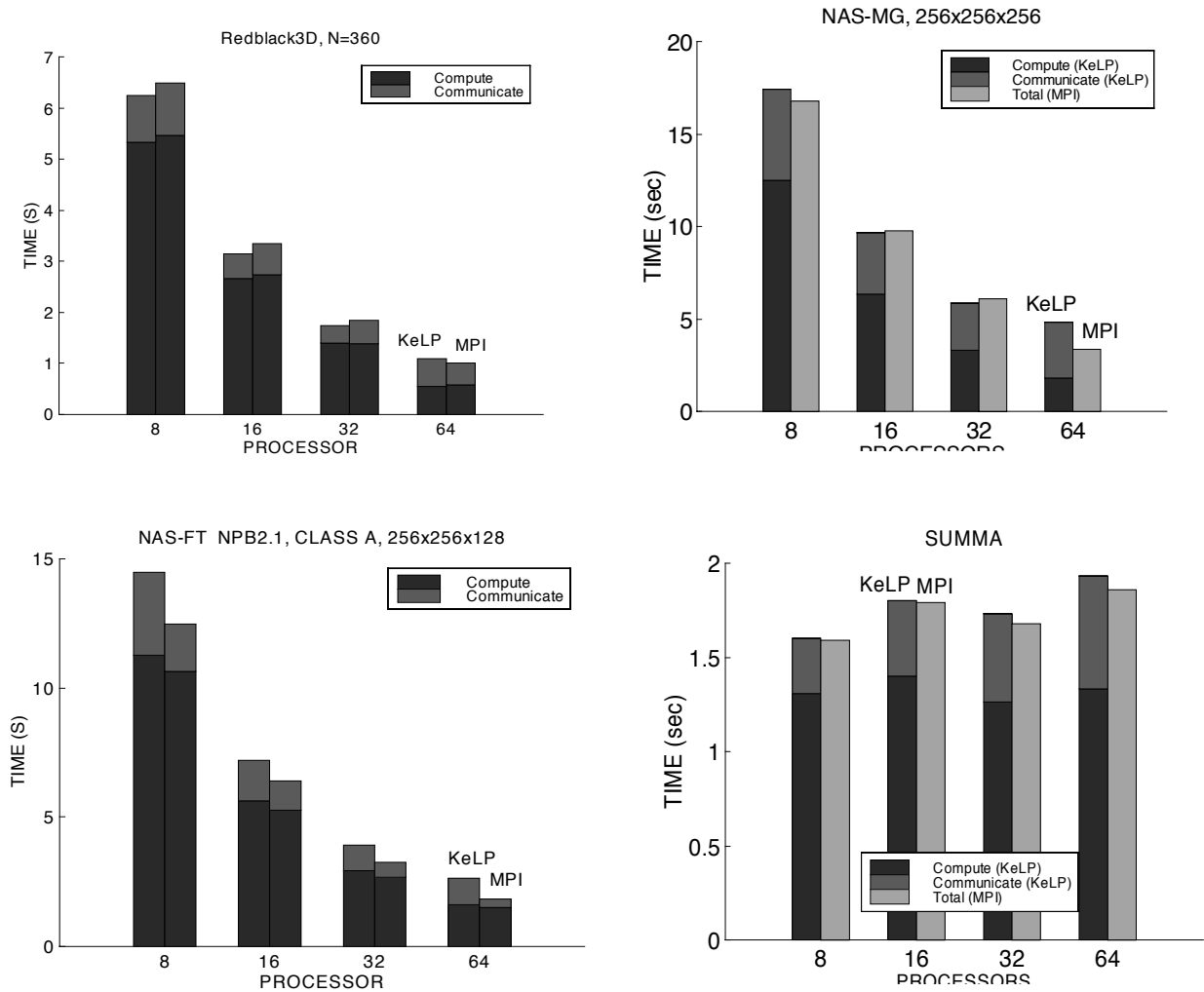


Figure 6. Comparative performance of equivalent KeLP and MPI codes running on the IBM SP2. The problem sizes are fixed, except for SUMMA, where the work increases with the number of processors (We used N=1400, 1800, 2200, and 2800 for 8, 16, 32, and 64 processors, respectively). “Communicate” gives the time spent waiting on communication.

Communication libraries like KeLP have been used in compilers or exist as middleware for compiler or application library builders. For example, sHPF, an experimental HPF compiler built at the University of Southampton [24], relies on a library called ADLIB, which provides high-level, array-based collective

⁶ See the URL http://king.ticam.utexas.edu/NPACI/IPARS_KELP_DAGH.

communication [25]. This C++ class library, built on top of MPI, not only provides runtime support for HPF but may also be used directly for distributed data parallel programming. ADLIB supports a variety of communication patterns to support HPF array assignment and dynamic redistribution. ADLIB also provides support for irregular communication via general gather/scatter.

Chamberlain, Choi, and Snyder describe the IRONMAN abstraction, which separates the expression of data transfer from the underlying implementation. In some cases this library was able to improve performance over MPI and PVM by making machine-specific optimizations [26]. Bela et al. Describe CCL, a portable and tunable Collective Communication Library [27]. Like KeLP, CCL is designed to run on top of an existing point-to-point message-passing interface. CCL supports a variety of collective operations, similar in spirit to what is now provided by MPI. The issue of multi-dimensional array support isn't addressed. The paper also contains several to earlier communication libraries of historical interest.

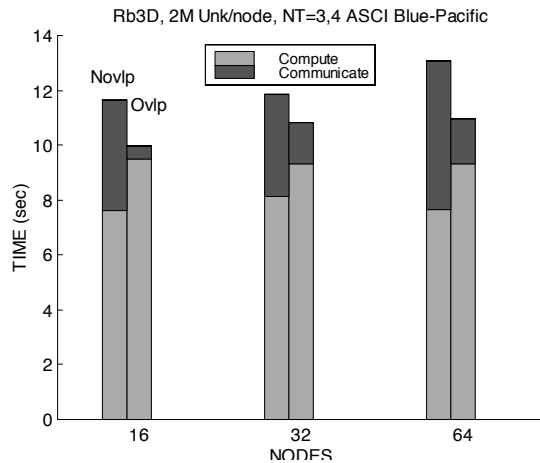


Figure 7. Communication overlap on the ASCI Blue Pacific CTR Redblack3D. A multi-tier KeLP implementation employs a dedicated extra thread to run the Mover as a proxy, and was able to improve performance by overlapping communication with computation. Overall running times improved by 8.7%, 14%, and 19%, on, respectively, 16, 32, and 64 nodes. The performance of the largest run was about 3.6 Gigaflops. In the non-overlapped runs, each node used 4 computation threads. One distinguished thread handled communication. The overlapped version ran with only 3 computation threads, and dedicated an extra thread to the Mover proxy. The numbers of nodes are given on the x-axis, total times (sec.) on the y-axis. “Communicate” gives the time spent waiting on communication, “Compute” the time spent in local computation. Computation time increases under overlap, since the proxy consumes processor cycles that could otherwise be devoted to computation.

5. Conclusions

We have demonstrated that the KeLP Data Mover is an effective abstraction for managing communication in diverse block-structured applications. This abstraction allows the programmer or compiler to express a variety of data motion patterns, without specifying underlying machine-specific implementation details. The Data Mover facilitates inspector-executor analysis, which is useful in irregular applications. The Mover is also useful on shared memory architectures or under single-sided communication; expensive global barrier synchronization may be replaced by a weaker and less costly form that enables processors to synchronize with point-to-point signals. In some cases it may be desirable to manage data motion explicitly [28], and the KeLP Mover implementation can take advantage of shared memory if doing so improves performance.

Though the KeLP Mover is currently defined to handle only regular section communication, an interesting possibility is to admit user-specified meta-data types. In the former case, an index domain could be defined along with appropriate region-calculus operations. This approach would make sense for finite element

methods that employ halo updates [29], using a different representation than stencil-based computations, but with a similar notion of locality.

We presented a Mover implementation that implicitly performs multi-method communication [12] [19], in which it will issue a memory copy in lieu of a call to MPI when it detects a data transfer involving blocks of data assigned to the same process. An interesting possibility is to generalize KeLP's multi-method communication to other kinds of data transfer methods, for example TCP/IP, to support topologically aware communication.

KeLP is currently defined to manage data motion among private memories of a multicomputer, among processors of an SMP, or both, in the case of SMP clustered technology. We are currently investigating Movers that transports data between disk and memory, or between a remotely executing program controlled by a Java front end. The KeLP Mover is a useful abstraction for managing I/O, since it supports the notion of scheduled, collective communication.

KeLP currently runs on a variety of architectures including the IBM SP2, IBM ASCI Blue-Pacific, SGI-Cray Origin 2000, and clusters of workstations running under Solaris, Linux, and Digital Unix. A port to the SGI-Cray T3E is in progress.

Acknowledgements

The authors wish to thank Jonathan May and Bronis de Supinski, with the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory, for the many illuminating discussions about the ASCI Blue-Pacific machine, and to Paul Kelly and Jarmo Rantakokko for advice on how to improve this paper.

This research was performed while the second author was with the Computer Science Department at UCSD, and supported by the DOE Computational Science Graduate Fellowship Program. Scott B. Baden was supported by NSF contract ASC-9520372 and NSF contract ACI-9619020, National Partnership for Advanced Computational Infrastructure. Work on the ASCI Blue-Pacific CTR machine was performed under the auspices of the US Dept of Energy by Lawrence Livermore National Laboratory Under Contract W07405-Eng-48. Computer Time on the IBM SP2 located at the San Diego Supercomputer Center was provided by SDSC and by a Jacobs School of Engineering Block grant. Thanks go to Richard Frost for assistance in building the KeLP distribution, and in providing KeLP tutorials to the user community.

References

1. J. Choi, *et al.* *ScaLAPACK: a portable linear algebra library for distributed memory computers. Design issues and performance.* In *Applied Parallel Computing, Computation in Physics, Chemistry and Engineering Science, second International Workshop (PARA '95)*. 1995.
2. V. Kumar *et. al.*, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. 1994: Benjamin-Cummings.
3. I. Foster, *Designing and Building Parallel Programs*. 1995, Menlo Park: Addison-Wesley.
4. S.J. Fink, *Hierarchical Programming for Block-Structured Scientific Calculations*, Ph.D. Dissertation, *Department of Computer Science and Engineering*, University of California, San Diego, La Jolla, CA. 1998. <ftp://ftp.cs.ucsd.edu/pub/scg/papers/1998/thesis.ps.gz>
5. S.R. Kohn, *et al.* *Software Abstractions and Computational Issues in Parallel Structured Adaptive Mesh Methods for Electronic Structure Calculations.* In *Workshop on Structured Adaptive Mesh Refinement Grid Methods*. 1997, in *Lecture Notes in Mathematics*, S. B. Baden, N. Chrisochoides, M. Norman, and D. Gannon, Editors. 1999. Springer-Verlag: Berlin. (To appear)
6. *The Message Passing Interface (MPI) Standard*, 1995, MPI Forum. <http://www-unix.mcs.anl.gov/mpi/index.html>
7. A. Sohn and R. Biswas, *Communication Studies of DMP and SMP Machines*, 1997, NAS, NASA Ames Research Center.
8. *MPI-2: Extensions to the Message-Passing Interface*, 1997, MPI Forum. <http://www-unix.mcs.anl.gov/mpi/index.html>

9. R.C. Agarwal, F.G. Gustavson, and M. Zubair. *An Efficient Parallel Algorithm for the 3-D FFT NAS Parallel Benchmark*. In *Scalable High Performance Computing Conference (SHPCC '94)*. 1994. Knoxville, TN.
10. S.J. Fink and S.B. Baden. *Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters*. In *1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97)*. 1997. Marina del Ray, California: Springer Verlag.
11. W.W. Gropp and E.L. Lusk. *A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters*. In *Programming models for massively parallel computers*. 1995.
12. S.S. Lumetta, A.M. Mainwaring, and D.E. Culler. *Multi-Protocol Active Messages on a Cluster of SMPs*. In *SC97*. 1997. San Jose.
13. *Understanding and using the communication Low-level Application Programming Interface (LAPI)*. 1997, IBM Corp.
14. P. Buonadonna, A. Geweke, and D. Culler. *An Implementation and Analysis of the Virtual Interface Architecture*. In *Conf. Proc. SC 98*. 1998. Orlando, FL.
15. S.J. Fink, S.R. Kohn, and S.B. Baden, *Efficient Run-time Support for Irregular Block-Structured Applications*. *Journal of Parallel and Distributed Computing*, 1998. **50**(1-2): pp. 61-82.
16. S.B. Baden and S.J. Fink. *Communication overlap in multi-tier parallel algorithms*. In *Conf. Proc. SC '98*. 1998. Orlando, FL.
17. 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*, 1995. **6**(7): p. 747-754.
18. L. Snyder, *Type architectures, shared memory, and the corollary of modest potential*. *Annual Review of Computer Science*, 1986. **1**: p. 289-317.
19. I. Foster and N.T. Karonis. *A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems*. In *Conf. Proc SC '98*. 1998. Orlando, Florida.
20. D. Bailey, *et al.*, *The NAS Parallel Benchmarks 2.0*, . 1995, NAS, NASA Ames Research Center
21. R.v.d. Geign and J. Watts, *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. *Concurrency: Practice and Experience*, 1997. **9**(4): p. 255-74.
22. S. Kohn and S. Baden, *Parallel Software Abstractions for Structured Adaptive Mesh Methods*. *Journal of Parallel and Distributed Computing*, In press.
23. J. Howe, *et al.* *Modernization of Legacy Application Software*. In *Applied Parallel Computing, 4th Int'l Workshop, PARA'98*. 1998. B. Kågström, J. Dongarra, E. Elmroth and J. Wasniewski (editors). *Lecture Notes in Computer Sci.*, No. 1541, Springer-Verlag, Berlin, 1998, pp. 255-262.
24. J.H. Merlin, D.B. Carpenter, and A.J.G. Hey, *SHPF: a Subset High Performance Fortran compilation system*. *Fortran Journal*, 1996 (March/April): p. 2-6.
25. B. Carpenter, G. Zhang, and Y. Wen, *NPAC PCRC Runtime Kernel (Adlib) Definition*. 1998, Northeast Parallel Architectures Center, Syracuse Univ., Syracuse NY.
<http://www.npac.syr.edu/users/dbc/Adlib>
26. B.L. Chamberlain, S.-E. Choi, and L. Snyder, *A Compiler Abstraction for Machine Independent Parallel Communication Generation*, in *Lecture Notes in Computer Science*, Z. Li, *et al.*, Editors. 1998, Springer-Verlag: Berlin. p. 261-276.
27. V. Bala, *et al.*, *CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers*. *IEEE. Trans. On Parallel and Distributed Sys.*, 1995. **6**(2): p. 154-164.
28. S.S. Mukherjee, *et al.* *Efficient Support for Irregular Applications on Distributed Memory Machines*. In *5th SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1995, pp. 68-79
29. D. Culler, *et al.* *Parallel programming in Split-C*. In *Conf. Proc. Supercomputing '93*. 1993. Portland, OR.