The Multicomputer Toolbox – First-Generation Scalable Libraries

Anthony Skjellum*

Computer Science Dept. & NSF Engineering Research Center
Mississippi State University

Alvin P. Leung

Syracuse University
Northeast Parallel Architectures Center

Steven G. Smith, Robert D. Falgout, Charles H. Still

Lawrence Livermore National Laboratory
Numerical Mathematics Group

Chuck H. Baldwin

University of Illinois, Urbana Champaign

Submitted to HICSS–27:

Minitrack on Tools and Languages for Transportable Parallel Applications

June 18, 1993

*To whom correspondence should be addressed. Address: Mississippi State University, CS Department, PO Drawer CS, Mississippi State, MS 39762. e-mail: *tony@cs.msstate.edu.*

Abstract

"First-generation" scalable parallel libraries have been achieved, and are maturing, within the Multicomputer Toolbox. The Toolbox includes sparse, dense, iterative linear algebra, a stiff ODE/DAE solver, and an open software technology for additional numerical algorithms, plus an inter-architecture Makefile mechanism for building applications. We have devised C-based strategies for useful classes of distributed data structures, including distributed matrices and vectors. The underlying Zipcodemessage passing system has enabled process-grid abstractions of multicomputers, communication contexts, and process groups, all characteristics needed for building scalable libraries, and scalable application software.

We describe the data-distribution-independent approach to building scalable libraries, which is needed so that applications do not unnecessarily have to redistribute data at high expense. We discuss the strategy used for implementing data-distribution mappings. We also describe high-level message-passing constructs used to achieve flexibility in transmission of data structures (Zipcode invoices). We expect Zipcode and MPI *message-passing interfaces (which will incorporate many features from Zipcode, mentioned above) to co-exist in the future.*

We discuss progress thus far in achieving uniform interfaces for different algorithms for the same operation, which are needed to create poly-algorithms. Poly-algorithms are needed to widen the potential for scalability; uniform interfaces make simpler the testing of alternative methods with an application (whether for parallelism or for convergence, or both). We indicate that data-distribution-independent algorithms are sometimes more efficient than fixed-data-distribution counterparts, because redistribution of data can be avoided, and that this question is strongly application dependent.

1 Introduction

First-generation scalable libraries have been developed within the *Multicomputer Toolbox* schema, also described elsewhere [9, 10, 8]. In this system, we have devised distributed data structures for vectors and matrices, defined relative to virtual process topologies (logical grids), as well as an advanced message-passing notation and system, called *Zipcode*, that manages processes, communication scope, and virtual topologies. All of this software has initially been tied to homogeneous assumptions, both in performance, and in data format. We describe steps to relax these assumptions systematically, work that has been underway for the past year.

We describe message-passing operation scope, process groups, and communication contexts, within the framework of parallel libraries. In a heterogeneous environment, the limited message-passing scope attainable by basing message-passing on groups plus a logical partition of receipt selectivity between user-specified and system-registered segments is conceptually pivotal; the data structures that arise are called "mailers" or intra-group-communicators. Libraries can readily be written to work without interference, given communication contexts, because they can acquire (through a

registry mechanism) additional communication contexts using a single safe mechanism for communication. Furthermore, the mailer framework is an ideal data structure in which to "cache" information on how operations should be implemented for a particular part of a communication hierarchy. Because of space limitations, we cannot describe in detail here how this is done in *Zipcode* (see [7]), but we do mention that introducing mailer-scope for communication (and data conversion) operations is an important source of runtime optimization for message passing. We mention the issues of library initialization and communication context use. From time to time, we relate *Zipcode* calls to the forthcoming MPI1 standard [2].

We consider the advances in message-passing notation that have been made to help improve the performance potential of codes based on *Zipcode*, particularly those that can incorporate "gather-send" and "receive-scatter" semantics. We describe how these become the building blocks for fully heterogeneous mathematical libraries, and which also offer encapsulation of any heterogeneous conversions. As such, we are able to assure portability between environments with reasonably compatible mathematical precision. We also see these semantics as offering the most optimizable message-passing constructs proposed as yet in any real system (what to do vs. how to do it).

We describe the data-distribution-independent approach to building scalable libraries, which is needed so that applications do not unnecessarily have to redistribute data at high expense. We discuss the strategy used for implementing data-distribution mappings, and how this can be advanced further. We also describe high-level message-passing constructs used to achieve flexibility in transmission of data structures (Zipcode invoices). We expect *Zipcode*and MPImessage-passing interfaces (which will incorporate many features from *Zipcode*, mentioned above) to co-exist in the future.

We discuss progress thus far in achieving uniform interfaces with for different algorithms for the same operation, which are needed to create poly-algorithms. Poly-algorithms are needed to widen the potential for scalability; uniform interfaces make simpler the testing alternative methods with an application (whether for parallelism or for convergence, or both). We indicate that data-distribution-independent algorithms are sometimes more efficient than fixed-data-distribution counterparts, because redistribution of data can be avoided, and that this question is strongly application dependent.

2 Overview of *Toolbox* Libraries

The following applications currently use *Toolbox* technology:

Current *Toolbox*-based Applications			
Name	Problem Domain	Principal Author(s)	Institution
Ardra	Neutron Transport	Milo Dorr	LLNL
Parflow	Groundwater Modeling	R. Falgout/S. Ashby	LLNL
Cdyn	Process Flowsheeting	A. Skjellum	MSU

These codes have run on several different configurations on machines including the Intel Delta, Paragon, nCUBE2, and networks of Sun workstations. Note that we expect the number of applications to grow markedly once we are able to release the software to the general public. Work on a dynamic power systems application (based on *CDASSL*and *Cdyn*) is also in progress.
The *Multicomputer Toolbox* supports the following libraries at present:

Toolbox Libraries Currently Supported	
CDASSL	Concurrent Differential-Algebraic Solver[13],
Citer	Krylov-subspace methods for linear system solution,
Csparse	Sparse LU solvers [11],
Cdense	Dense level-2 and level-3 LU solvers (see section 7),
Cblas	Concurrent BLAS library (in development) [4],
Cvector	Concurrent Vector operations (and transformations),
Cdistri	Data-Distribution-Independence Support,
Range	Index set manipulation,
Resources	Portability Support,
Zipcode	High-level message passing library with virtual topologies, groups, contexts, and communicators (mailers),
CE/RK	Message-Passing Porting Layer for Multicomputers and Homogeneous Networks.

Notable current omissions are FFTs, QR factorizations, and eigensolvers. We hope to close some of these gaps in the future.

3 *Zipcode* Communication System

In this section, we describe how the message passing system *Zipcode*, whose specific design purpose was to support parallel libraries in parallel applications, has been designed and extended to provide

basic services useful for library management. We also compare current practice in *Zipcode* to standardization efforts underway in MPI1.

3.1 Programming Model

We assume a multiple-instruction, multiple-data programming model. Multiple program texts are possible within the system. Libraries typically operate in a loosely synchronous fashion. However, multiple independent instances and, overlapping process groups are permitted. Support for asynchronous operations is included (for instance, users could define their own library for an asynchronous collective operation).

3.2 Porting Strategy

Zipcode currently relies on the basic process management (spawn/kill) and messaging services (x-primitives) of the Reactive Kernel / Cosmic Environment, or, more usually, emulations thereof [5, 6]. This strategy has been effective in that we have produced stable, usable ports for the Symult S2010, nCUBE/2, iPSC/2, iPSC/860, Delta, Paragon, BBN TC2000, CM-5 scalar machine, Sun workstation network, and RS/6000 networks during the past five years. A port to the PVM systems is nearly completed [1]; integration of *Zipcode* with ELROS messaging capabilities is also being undertaken [3]; a direct TCP/IP port is also underway, which omits a PVM-like intermediate library,

Zipcode supports the common multicomputer HOST/NODE model of computation, which essentially means that there is an initial process that is responsible for the main part of the "sequential fraction" of computation, including spawning, killing, and initializing the parallel processes of an application. This model is not as general as one would prefer in an hierarchical, heterogeneous environment, but is a reasonable starting point. On a related note, certain multicomputer systems we have addressed in the past do not allow for dynamic process management (*e.g.*, Intel Delta), and many restrict programming to one process per processor. For such systems, operations like "spawn process" and "kill process" are NULL operations, but reasonable portability is still maintained.

3.3 Process Naming

In all *Zipcode* versions to-date, we have utilized the *Reactive Kernel*'s {`node, pid`}-pairs to describe processes in a pool, whether in a single multicomputer, or attached to a network. For a given

implementation, the {`node, pid`} pair will be mapped to hardware, but this naming remains visible during the initialization process, during which processes are created (spawning). This notation is seen as extremely unattractive for programming by the user, but is rarely used because of automatically generated process groups (addressee lists) and virtual topologies.

Once message-passing has been set up, most *Zipcode* programs work with logical process grids of one-, two-, or three-dimensions. Furthermore, an advanced user can add new virtual topologies to the system. Some libraries might like to have tree topologies, for instance, to make them most natural to program. Virtual topologies provide naming that maps to an addressee list. This level of virtualization hides the implementation of addressee lists from the application, while improving notation.

3.4 Process Groups

A process group is a basic abstraction that has been found to be useful in a number of message-passing systems. A process group might be a way of describing the participants in a communication primitive, such as a synchronization (perhaps with additional information to encapsulate that particular collective operation instance from other operations). In *Zipcode*, a process group is called an "addressee list," and has the following properties:

- It is a logical, ordered collection of {`node, pid`} pairs.
- It has a rank (number of members).
- It is a purely local object.
- Communication cannot be expressed solely in terms of addressee lists.
- An addressee list can be transmitted in an extant communication context (see below).

In *Zipcode*, we have consistently implemented addressee lists as enumerations of {`node, pid`} pairs; originally, users assembled addressee lists, but addressee lists are now to be considered opaque; a standard constructor is provided:

```
ZIP_ADDRESSEES *addressees =
  zip_new_cohort(int N, int node_bias, int cohort_pid, int pm_flag);
```

where

- `N` is the number of processes involved, or one less than the number of processes involved if `pm_flag` is true,
- `node_bias` is the suggested node-number offset to start with when spanning the user's logical allocation of processors,
- `cohort_pid` is the suggested, constant process ID of the entire collection of processes,
- `pm_flag` flags whether the process calling `zip_new_cohort()` is introduced as its zeroth entry, and hence the "postmaster" (group leader) for communication based on this addressee list (see below, under context creation).

This call builds a sensible set of process names over the range of logical nodes available in the user's allocation. The system may choose to override the `cohort_pid` suggestion never, immediately, or when processes are spawned using the addressee list (see below), The system may choose to override the `node_bias` naming never, immediately, or when processes are spawned using the addressee list (see below). These relaxations retain the opaque nature of the underlying addressee list, which is important to future generalizations of process naming. Furthermore, the needed transmission of addressee lists during context creation retains this desirable opacity.

Since internal manipulations of addressee lists is denigrated practice, addresee lists can be generalized in future *Zipcode* releases without breaking conforming code. Particularly, there could be additional portable ways to construct, modify, and transmit addressee lists, and particular environments could provide non-portable calls to provide additional addressee lists with appropriate opaque structure. Within the *Zipcode* system itself, there remains the need for non-enumerative representation inside an addressee list, and more general process naming (*e.g.*, PVM task ID's, or, preferably, handles to general opaque name objects). *Zipcode* will be generalized appropriately.

For completeness, *Zipcode* provides the following process management support, for which there is no planned analog in MPI1:

```
int result = zip_spawn(char *prog_name, ZIP_ADDRESSEES *addressees,
   void *state, int not_pm_flag);
```

where

- `prog_name` is the ASCII name of the program to spawn, local to the spawner's file system,
- `addressees` addressee list upon which to spawn the program,
- `state` is unused, future expansion,
- `not_pm_flag` flags if TRUE, program is spawned on the zeroth addressee, of addressees.

and where `result` is non-zero on failure. Most implementations require that this spawning function be effected in the HOST process, though this restriction is less likely in a distributed setting. If the caller to `zip_spawn` is the zeroth entry in the addressee list (role of postmaster), it is erroneous to attempt to set `not_pm_flag` true. So, a valid HOST/NODE spawning procedure would be

```
#define FALSE 0
#define TRUE ~FALSE
   int N = 256, try_pid = 33;
   addressees = zip_new_cohort(N, 0, try_pid, TRUE);
   result = zip_spawn("./testprog", addressees, NULL, FALSE);
```

A compatible `zip_kill()` is also defined:

```
    result = zip_kill(addressees);
```

With the inclusion of these functions, *Zipcode* specifies an entire programming environment that can be completely divorced from its original relationship with the *Reactive Kernel / Cosmic Environment (CE/RK)*.

3.5 Contexts of Communication

A communication context is an abstraction that was introduced by the author in the original (1988) *Zipcode* system, and which also will appear in the MPI1 standard [12, 2]. In order to write practical, "safe" distributed-memory, and/or distributed-computing libraries, communication contexts are needed to restrict the scope of messages. This is done to prevent messages from being selected improperly by processes when they do message passing. We described contexts in several papers on *Zipcode* [12, 15, 14]. Without this type of scope restriction, it quickly becomes intractable to build up code without globalizing the details of how each portion of a code utilizes the message-passing resource. Communication contexts are therefore central to creating reusable library code, and to maintaining modularity in large-scale distributed application codes, with or without third-party libraries.

A context of communication has the following properties:

- A context of communication is based on an addressee list, the members of which are the assumed participants in the communication,
- A context of communication has one or more system-defined, labelings ("zipcodes," or context_ids in MPI1) of message passing for its addressee list the system,

- It provides a logical partitioning of receipt selectivity into user-defined, and system-registered components.
- If used correctly, allocated zipcodes guarantee that messages will not be misdirected.
- A zipcode is a globally unique quantity, but may be reused in disjoint groups.

To enforce safe programming, the following strictures are placed on message-passing in *Zipcode*:

- Send/receive (point-to-point) and collective communication work only within contexts of communication,
- Wildcarding, where permitted, does not violate context boundaries.

The creation of a context synchronizes the participants in that (future) context, while promulgating the addressee list, and zipcodes (`context_id`'s). Only the "postmaster" (initial member of an addressee list) must know the addressee list initially. All other processes just need to know that they are going to create a communication context. The context server provides the needed zipcodes, and promulgates them with the addressee list information to all participants. A token released by the context server is held to ensure that the process completes without the chance that mailers fail because of race conditions on overlapped collections with distinct postmasters. (A server-free model is also possible, but has not yet been implemented in *Zipcode*.) The following call allocates globally unique zipcodes (`context_id`'s) that can subsequently be used to build contexts of communication, when merged with addressee lists:

```
ZIP_ZIPSET *zips = zip_newzips(int N);
```

The following is the simplest form of the mailer creation call, in the 3D virtual topology (of shape $P \times Q \times R$).

```
ZIP_MAILER *g3_grid_open(int *P, *Q, *R, ZIP_ADDRESSEES *addressees)
```

The postmaster for the communication context calls this procedure with a valid addressee list, and valid values for the grid shape. All other participants call with undefined values for the grid shape, and NULL for the addressee list. A variant exists that permits the zipcodes to be specified. In that case, if all participants know their addressee list, then mailer creation is communication-free.

4 Issues from *Toolbox* Libraries

4.1 Initialization

Library initialization is a difficult question for conventional message-passing systems, because it is extremely tricky to predict how the receipt-selectivity-space will be partitioned by multiple invocations of the same library, by distinct libraries, by user programs, and even by collective communications implemented by a vendor (*e.g.*, Intel NX/2 uses the tag space to enforce order in collective combine). Having a library writer publish his/her "range of tags" utilized, which is a common alternative suggested to contexts, simply does not provide enough safety.

Zipcode provides two communication contexts (both encapsulated in the same mailer), by default: one for point-to-point and one for loosely synchronous, collective communication. This is defined to be a basic, safe environment for message passing, from which libraries could acquire additional contexts, as needed. (The second context is needed in the portable *Zipcode* implementation, since point-to-point messages are used to effect collective operations, rather than through alternative network hardware, as a vendor might do.) For each additional type of collective operation that is asynchronous or non-deterministic (*e.g.*, an asynchronous broadcast where the source is not known initially, and where tags must be used with point-to-point operations to preserve correctness), an additional communication context is needed. For each level of stack depth of libraries called, an additional context of communication is needed. For each overlapping pair of addressee lists, separate contexts must be defined for safe communication.

Since users interact mainly with virtual topologies (which are small collections of mailers), it is important to understand virtual-topology requirements and properties. *Zipcode*'s `context_id`'s (zipcodes) are globally unique when issued, yet the built-in functions that implement virtual topologies reuse them safely on non-overlapping children of a virtual topology. As such, a three-dimensional virtual topology of shape $P \times Q \times R$ requires one pair of contexts for transmissions across the three-dimensional collection, and some number $3K$ additional contexts, where K is the number required for any one of the two-dimensional children (any process belongs to three plane subsets). Each of these two-dimensional logical process planes requires a pair of communication contexts for planar communication, and $2L$ additional contexts, where L is the number required by a one-dimensional process grid (a row or column). Evidently, $L = 2$, resulting in a total of two contexts for a one-dimensional grid, six for a two-dimensional grid, and twenty for a three-dimensional grid.

If contexts could not be reused as just described, then the total number of contexts would grow with P, Q, R, rather than being constant (which is undesirable).

The initialization process for a three-dimensional grid instigates one synchronization of all participant processes, allocates the needed zipcodes (using *Zipcode*'s context server process), and then builds all lower-dimensional children without further communication. This sub-children creation can be done safely without further synchronization because of receipt-selectivity semantics of *Zipcode*: It is legal to post a message to a context before any mailer has been established on the recipient that supports that context of communication. After significant discussion during the definition of MPI1, we have also confirmed the need to support these semantics there, regardless of how system-defined virtual topologies are built in MPI1. For both message-passing systems, libraries can allocate reserve (*i.e.*, "extra") context_id's, and then use them at will, without costly synchronizations. This provides for safe message passing without synchronizations as one nests library calls in an application. So, the natural interpretation of the communication context as partitioning the receipt-selectivity space also leads to the best semantics for libraries, from the point of view of minimizing the number of synchronizations. It does not necessarily lead to the cheapest message-passing system from a low-level implementation perspective.

4.2 Objects & Interactions

Thus far, we have written numerical libraries, such as dense matrix-vector multiplication, to utilize pairs of distributed objects. A dense matrix is defined as distributed on a two-dimensional virtual topology, by specifying a specific mailer that is also identified as a two-dimensional topology. Initially, that mailer has available a safe context of communication for point-to-point and loosely synchronous collective communication. Similarly, we define vectors as replicated objects along one axis of that same topology, again relative to the identical mailer. When objects are created, additional zipcodes (`context_id`'s) could have been allocated, to provide safe communication for member functions working on the object. However, when friend functions are applied (*i.e.*, between a matrix and vector, or two matrices), one has to be careful to utilize a valid context of communication for the operations. For instance, validity of operating on two distributed objects is based on the equality of their mailers in the current *Toolbox*, rather than by performing an expensive congruency test on logical process grids. Hence, it is currently necessary for distributed objects to reveal the base virtual topology, and manage extra zipcodes (`context_id`'s) separately (though

they imply ephemeral contexts of communication, in general). This is so that the equality test can be satisfied in libraries that do error checking for compatible distributed objects.

We have not fully faced this issue in *Toolbox* libraries as yet, because we have assumed up to now that users do not leave dangling communications before calling a constructor. This is a bad assumption, in general. In fact, one has to question how safe a context of communication is if libraries have no state, but that they only initialize in the sense of creating objects like distributed matrices. One would have to be certain that no spurious communications were pending before creating such objects, or the program would be erroneous. As we don't want to leave this complexity to the user (a decision consistent with our efforts to create contexts), we might not want to rely on the supposedly safe context of communication provided by the mailer, but only rely on a `context_id` provided to the library (*e.g.*, when it was globally initialized), coupled with the addressee list of the passed-in mailer. So, each unique library in the system needs a pair of zipcodes, so it can bind them when constructing distributed objects. The correct user program is going to have to initialize each library in the system explicitly (or implicitly with static constructors, when we use C++ in the future). This type of library-scope initialization would not be user thread safe, returning the burden to the caller to provide safe context of communication when calling constructors for distributed objects.

5 Implementing Data Conversion

In the fully heterogeneous environment, data conversion will be needed within all heterogeneous communication contexts. We wanted to achieve the following goals (and have done so):

- No explicit conversion calls in user or library code.
- No extra data motion when homogeneous communication contexts are involved.
- Support for collective operations in the heterogeneous model.
- No user intervention with "how" buffers (if any) are formatted, nor how message protocol is handled (who converts, converts to what intermediate form, etc).

In other words, we make the message passing itself opaque. For point-to-point, the user's interface is a gather specification and destination on the sender's side, and a source and scatter specification on the recipient side. For collective communication, a macro procedure is used so that the user

specifies the associative-commutative operation; *Zipcode* generates the code needed to handle both the fully heterogeneous and homogeneous cases. For more details, see [14, 7].

5.1 Building Encapsulated Data Distribution Objects

Zipcode, together with the *Cdistri* library, support data distribution objects. Currently, the model is restricted to representations on 2D grids (though this can be relaxed). Specifically, the call

```
mailer = g2_grid_open(&P, &Q, addressees)
```

on the "postmaster" and

```
mailer = g2_grid_open(&P, &Q, NULL)
```

on the non-postmaster processes result in a $P \times Q$ grid, which excludes the postmaster (the postmaster can send and receive from the grid, but only via "out-of-band" communication, which we don't cover here). The pointer "mailer" represents a hierarchy of mailers, including the two-dimensional grid, and its logical one-dimensional row/column children. *Cdistri* includes the following constructors as well:

```
Cdistrib *new_Cdistrib(ZIP_MAILER *g2mlr, int rctype, int dist,
                           void *init_mu_extra)
CMdistrib *new_CMdistrib(Cdistrib *rdis, *cdis)
```

The data structures `Cdistrib` and `CMdistrib` encapsulate powerful mappings between global (sequential) naming of indices, and corresponding {process, local_index} naming in each dimension (currently row or column). The underlying two-dimensional logical process grid is included. These data structures form the basis of all distributed mathematical objects in the *Toolbox*to date.

```
typedef struct _Cdistrib
{
        ZIP_MAILER *g2mlr;              /* grid mailer */
        ZIP_MAILER *rcmlr;              /* row or column mailer + type */
        short      rctype;              /* specifies axis */
        short      dist;                /* Distribution type. */
        void       (*mu)();             /* distribution mapping */
        int        (*mu_i)();           /* and its inverse */
```

```
        int        (*mu_lim)();         /* ``limits'' fn. */
        int        (*mu_init)();        /* initialization fn.  */
        void       *mu_extra;           /* extra info. needed by mappings */
        Cdistrib_data *data;            /* problem size global/local data */
} Cdistrib;

typedef struct _CMdistrib
{
        ZIP_MAILER *g2mlr;              /* the subject 2D grid mailer */
        Cdistrib *rdis;                 /* row distribution */
        Cdistrib *cdis;                 /* column distribution */
        CMdistrib_data *data;           /* problem data */
} CMdistrib;
```

which are based, in turn, on the following:

```
typedef struct _Inv_proj_entry
{
        int      Inv;                   /* global name of invariant */
        index    i, j;                  /* local names in row/col sets */
} Inv_proj_entry;

typedef struct _Inv_proj                /* dis/dis invariant projection */
{
        int      n_invariants;  /* # of invariants, this process */
        Inv_proj_entry *entries; /* array of entries */
} Inv_proj;

typedef struct _Cdistrib_data
{
        int   M;               /* Global problem size in this dimension */
        index m;               /* Local  problem size in this dimension */
        void *extra;
} Cdistrib_data;

typedef struct _CMdistrib_data
{
        Cdistrib_data *rdata;  /* row problem data */
        Cdistrib_data *cdata;  /* column problem data */
        Inv_proj *rc_inv_proj;          /* row/col. data distribution proj. */

} CMdistrib_data;
```

These latter structures provide encapsulation of the (optional) problem size information within a `Cdistrib` or `CMdistrib`. The "invariant projection" data stores the information about which ele-

ments do not cross process boundaries when considering a conversion from row to column mapping within the grid. As such, they are useful for reducing the cost of specific operations that redistribute data.

6 *Toolbox* Linear System Libraries

We recognize three distinct classes of approaches for solving linear systems: direct methods (*e.g.*, LU factorization), semi-iterative or Krylov-subspace methods (*e.g.*, GMRES, QMR, PCG) and stationery iterative methods (*e.g.*, Jacobi, SOR). The *Toolbox* provides libraries to solve linear systems using the first two methods through the *Cdense*, *Csparse* and *Citer* libraries, and will soon include Jacobi and SOR algorithms (with support for dense and sparse data structures). In what follows, we illustrate *Cdense* and *Citer*data structures, and subsets of applicable calls.

6.1 Uniform Calling Interface

If there were no high-level uniform calling interface for these diverse methods, however, the user would be forced to spend considerable effort to interface these libraries to his or her program. Furthermore, in view of the need for "poly-algorithms" in order to increase application scalability, uniform calling interfaces become increasingly important. Consequently, a coherent linear solver interface is needed to ease the programming effort and to increase the portability of the user program to the latest generation of *Toolbox*linear solvers. This is our first version of such an interface.
The linear library interface consists of one structure and several calls:

```
typedef struct tbx_linear_system
{
   void   *Info; /* This corresponds to "A", methods, data */
   void   *b;    /* This corresponds to 1 or more right-hand sides */
   void   *x;    /* This corresponds to 1 or more unknown vectors */

   Method *linear_solver; /* specific method from Info for solving system */

   Extra *extra;

} Tbx_linear_system;
```

The information content is broadly as follows:

- `Info` — Everything to do with the matrix (or generalized linear operator)
- `b` — Everything to do with right hand sides
- `x` — Everything to do with unknowns
- `linear_solver` — The method for solving the system
- `extra` — Other information needed by system

The constructor for this object is as follows:

```
void (*new_linear_solver)(void *Info, void *b, void *x, linear_solver);
```

This constructor returns the `Tbx_linear_system` structure with all the default parameters set for the specified solvers. After the user has obtained this structure, problems can be solved by specifying the compact call:

```
error = Tbx_Solve_Linear_System(linear_system);
```

which expands to

```
error = (*linear_system -> linear_solver)(Info, b, x, extra);
```

The destructor is as follows:

```
void Tbx_free_linear_system(Tbx_linear_system *linear_system);
```

This operation destroys only the top-level data structure created by the previous constructor.
We find this high-level encapsulation to be good as long as there is no need for a broader interface between the solver and a higher-level accuracy check (such as if an inexact Newton method is coupled to a linear solver). For such cases, the Newton solver will have to be more intimately tied to the underlying linear solver, and a set of these methods will be needed, rather than a single Newton method. We comment on this further under section 6.3.

6.2 *Cdense* Functionality and Interfaces

The concurrent dense matrix has a local dense matrix, the global problem size (`M, N`), and the matrix distribution, `CMdistrib *Mdis`. The latter contains the two-dimensional grid mailer from Zipcode, as well the row and column data distribution mappings. As such, the matrix so represented is general:

```
typedef struct _Cmatrix
{
        matrix  *a;             /* The local matrix. */
        int     M, N;           /* Dimensions of Cmatrix. */
        int type;               /* more data on Cmatrix type */
        CMdistrib *Mdis;        /* Data distribution on Grid */

} Cmatrix;
```

The local `matrix` data structure in turn includes the local storage, local size, and orientation (row- or column-major):

```
typedef struct _matrix
{
        index   m, n;
        double  **s;
        index   orient_flag; /* row- or column-major */
} matrix;
```

In order to solve linear systems, a linear system data structure is provided for both the level-2 and level-3 factorizations. For simplicity, the level-2 variant of this data structure is depicted here:

```
typedef struct _Clu_info
{
        Cmatrix   *A;
        int       *perm;        /* permutation info [stored scalably] */
        void      (*pivot)();   /* pivot selection strategy fn. */
        double    *piv;         /* U temporary data space */
        double    *updt;        /* L temporary data space */

        int       rank;         /* estimated rank after factorization */
        int       done;         /* has factorization in place been done? */
```

```
        double    condition;    /* part of pivot tolerance calculation */
        double    tolerance;    /* part of pivot tolerance calculation */

} Clu_info;
```

The following function calls implement LU factorization within *Cdense*:

```
void lu_factor_Cmatrix_lvl_2(Clu_info *LU, Cmatrix *B, Cvector *rhs)
void lu_factor_Cmatrix_lvl_3(Clu_info *LU, Cmatrix *B, Cvector *rhs)
```

Both single (replicated) and multiple right-hand sides are supported. The following function calls implement triangular solves:

```
void fwd_solve_lu_Cmatrix(Clu_info *LU, Cvector *rhs, *sol) /* forward only */
void back_solve_lu_Cmatrix(Clu_info *LU, Cvector *rhs, *sol) /* back only */
void solve_lu_Cmatrix(Clu_info  *LU, Cvector *rhs, *sol) /* forward/back */
```

Other variants (such as those that can exploit pipelined back-solve techniques, despite data distribution independence) are in the works.

6.3 *Citer* Functionality and Interfaces

Citer currently supports the following Krylov sub-space algorithms:

- GMRES — General Minimized Residual Method
- PCGS — Preconditioned Conjugate Gradient Squared
- PCG – Preconditioned Conjugate Gradient
- PTFQMR - Preconditioned Transpose-Free Quasi-Minimum Residual Method

As other methods are demonstrated to provide distinct advantages as compared to these four algorithms, additional methods will be added. For now, this appears to be a reasonable set of methods from which to choose. One exception to this is our intent to add the ability to support polynomial preconditioned iterative methods, specifically polynomial preconditioned conjugate gradient method (PPCG). On another note, we intend to add stationery methods such as Jacobi, SOR/Gauss-Seidel. These will be done as soon as possible, that is, before the *Citer* library has its first public release (4th quarter of 1993).

6.4 General Structures for *Citer*

Significant encapsulation occurs within *Citer*. For instance, the following structure houses the concurrent inner products to be supported for a typical iterative method:

```
typedef struct _inner_product_bundle
{
   Method *inner_product;       /* inner_product(x,y,&ip,extra)    */
   Method *skew_inner_product;  /* as above but for skew vectors   */
   Method *multi_inner_product; /* multi_inner_product(v1,v2,ip,num, extra)
                                 * v1, v2 are arrays with num # of vectors.
                                 * ip is array of doubles for results   */
   Method *skew_multi_inner_product; /* as above but for skew
                                      * vectors  */
} Inner_Product_Bundle;
```

(Actual instantiations of these methods reside in *Cvector*.) Furthermore, the following structure packages matrix-vector multiplication functions for iterative solvers:

```
typedef struct _matvec_bundle
{
   Method *matvec;      /* x := bAy + cz matvec(b,A,y,c,z,x,extra) */
   Method *matvec_T;    /* x := by^T A + cz
                         * matvec_T(b,A,y,c,z,x,extra)       */
} Matvec_Bundle;
```

It should be noted that, separate from *Citer*, one supports specific matrix data structures and matrix-vector multiplications. For instance, the "Parflow" application at Lawrence Livermore National Laboratory uses its own stencil library, but this has not been abstracted and added to the *Toolbox* as of yet. We currently support dense matrix-vector multiplication for the `Cmatrix` data structure defined by the *Cdense* library, and we are considering the addition of a general sparse technology as well. This remains to be done.

Within *Citer*, we add the notion of a "matrix," though matrix-free methods are equally well supported:

```
typedef struct _citer_matrix
{
```

```
   void           *A;
   CMdistrib      *Mdis;
   Matvec_Bundle  *mv_bundle;
   long            rows, columns;
} Citer_Matrix;
```

The culmination comes in the top-level data structure `Citer_Info`; the following structure describes the entire iterative linear system:

```
typedef struct _citer_info
{
   Citer_Matrix    *coeff_matrix;
   Citer_Matrix    *left_precond;/* z := M r                          */
   Citer_Matrix    *right_precond;

   Method         *iter_solver; /* Solves Ax = b   iter_solver(iter_info,x,b)
                                 * method dependent data is located in
                                 * iter_solver -> extra                */

   Method         *store_results;       /* called at each iteration to store
                                         * results                          */

   Method         *term_cond;   /* called at start of each iterations    to
                                 * determine if solver should be terminated. */
} Citer_Info;
```

Note that we support both left and/or right preconditioning in the formalism. Furthermore, we provide a general format in which the user can elect to store arbitrary information about some or all of the past iterates through the `store_results` mechanism. Finally, "extra information" for the implementation of the termination condition method is also supported:

```
typedef struct _citer_residual_test_extra
{
   int             max_iter;
   double          tol;
} Citer_Residual_Test_Extra;
```

Standard error codes are supported by *Citer*, and all libraries must conform to these return codes:

Error Codes for *Citer*		
Symbolic Name	Value	Description
CITER_CONVERGED	0	Normal Completion
CITER_INFO_PROBLEM	1	Error with Info Structure
CITER_BREAKDOWN	2	Method had "breakdown" anomaly
CITER_MATVEC_ERROR	3	Matrix-vector error code reported
CITER_PRECOND_ERROR	4	Preconditioner reported error

6.5 Example of Data Structures for Preconditioned Conjugate Gradient Squared (PCGS)

In the foregoing section, we defined the framework for iterative solvers within *Citer*. In this section, we describe the additional data structures for one of the supported solvers, Preconditioned Conjugate Gradient Squared (PCGS).

```
typedef struct _citer_pcgs_extra
{
   Cvector        *rt, *p, *z, *r, *q, *u, *v, *w, *t;
   Inner_Product_Bundle *ip_bundle;
   Method         *wvsum;
   double          breakdown_tolerance;
} Citer_PCGS_Extra;
```

The constructor and destructor the the extra information needed by PCGS are as follows:

```
Extra *citer_new_PCGS_extra(Inner_Product_Bundle * ip_bundle,
        Method * wvsum, CMdistrib *matrix_dis,
          long row_size,long col_size, double tol);

void citer_free_PCGS_extra(Extra * extra);
```

Finally, the actual call for this solver is as follows:

```
int citer_PCGS(Citer_Info * PCCG, Cvector * rhs, Cvector * sol);
```

which is the same calling sequence used by all *Citer* solvers.

7 About Data Distribution Independence

As a proof-of-principle, as well as a significant related area of research, we have worked on high performance, data-distribution-independent dense LU factorization algorithms, and demonstrated these on the Caltech Intel Delta prototype, during its "acceptance test phase." Within the *Toolbox* framework, we were able to generate sustained performance of three double-precision gigaflops (3.0×10^9) with a non-blocking, right-looking, data-distributiion independent LU factorization (on an order 10,000 dense matrix). This level of performance occurs for a particular logical grid shape, 18×28, with a scatter-scatter distribution of both rows and columns. Less-than-optimal grid shapes and data distributions generate somewhat inferior results (see Tables 2, 3, 4). However, for a number of variations, performance degrades only slightly. Hence, the needs of the application generating the matrix, in terms of row- and column- parallelism, and in terms of data locality can still be factored into the overall tuning of an application. For problems not much smaller than our test example, explicit redistribution of data is prohibitively expensive on the Delta and similar machines. Consequently, the data-distribution-independent algorithm is the most important one: it can generate the high performance results for the special distribution that is "optimal" for it; yet, it can also function when an application code needs to generate some other distribution.

For the present solver, the computational kernel is a BLAS level-2, rank-1 update called `dger`, the least efficient of level-2 operations, because of its data reuse characteristics. Its single-node performance is depicted in Table 1. Our further work, still in progress, has led us to a data-distribution-independent level-3 BLAS right-looking solver, capable of approximately eight gigaflops for the best data distribution, and useful (with somewhat degraded performance) for other distributions that may occur in real applications. When complete, this solver will offer higher computational performance, without sacrificing data-distribution independence. In other words, we find that, with greater effort, we can still exploit nodal pipelining or vectorization, without seriously compromising data distribution independence.

To summarize, for this class of operations, data-distribution independence becomes less important for $N > N^*$, where N^* is the smallest matrix size for which explicit data redistribution costs an order-of-magnitude less than factoring in the "optimal data layout," both operations considered with the same number of nodes. Even for $N > N^*$, it may not be economical to redistribute data, however, from nearly optimal or even mediocre distributions, to the optimal distribution, because,

the improvement in performance may be marginally less valuable than the cost of redistribution. Hence, the data-distribution-independent algorithm remains relevant for large N, depending more or less on the application requirements. For very bad data distributions, however, explicit redistribution will make sense even for $N \ll N^*$...there is a "shrinking table" of distributions for which redistribution is economical as N decreases. For computational steps with lower time complexity, problems will have to be even larger in dimension before one can begin to neglect data distribution independence.

Table 1. i860 : dger mflops		
Size	*Assembled*	*Compiled*
100	10.343	8.368
250	11.221	9.171
500	11.434	9.560
1000	11.434	9.901

Table 1: Performance of single-node double-precision rank-1 update of a dense matrix, on the Intel i860 chip, based on code optimized with a compiler, or by hand coding. This serves as the computational kernel for our non-blocking, data-distribution-independent LU factorization.

Table 2. Size = 10000x10000, Scatter/Scatter			
Shape	*Time*	*Gflops*	*Mflops/node*
6x84	264.827s	2.517	4.995
7x72	242.605s	2.748	5.452
9x56	233.643s	2.853	5.661
10x51	226.765s	2.940	5.765
11x46	225.850s	2.952	5.834
12x42	224.525s	2.969	5.891
13x39	221.401s	3.011	5.939
14x36	224.419s	2.971	5.894
15x34	219.533s	3.037	5.954
16x31	219.273s	3.040	6.130

Table 2: The Intel Delta performance for the right-looking LU algorithm generated these performance results. Data-distribution-independent, partial-row pivoting LU factorization utilized the assembly coded level-2 rank-1 update `dger`. Gigaflops were computed as $\frac{2}{3}N^3 10^{-6}/T$ where $N = 10,000$ here, and T was the observed maximal runtime in seconds. Each time T quoted is the average of two or more repetitive runs. A number of similar grid shapes and node counts produce similar performance, suggesting an important degree of freedom left to applications that will call this kernel .

Table 3. Size = 10000x10000, Scatter/Scatter			
Shape	*Time*	*Gflops*	*Mflops/node*
18x28	217.508s	3.065	6.081
21x24	223.732s	2.980	5.912
24x21	241.501s	2.761	5.477
28x18	249.990s	2.667	5.291
32x8	406.676s	1.639	6.404
16x16	351.916s	1.894	7.400
8x32	353.402s	1.886	7.369

Table 3: Here we see, for the LU factorization, the optimal runtime achieved for the 18×28 grid shape. We also see that using less nodes (last three lines utilize 256 nodes) achieves higher effective per-node performance, but longer overall runtime.

Table 4. Size = 10000x10000, 8x63 shape				
Distribution				
Row	*Col*	*Time*	*Gflops*	*Mflops/node*
Scat	Scat	225.230s	2.960	5.873
Scat	Lin	254.054s	2.624	5.210
Lin	Scat	343.812s	1.940	3.850
Lin	Lin	444.143s	1.501	2.980

Table 4: The LU factorization on the efficient 8×63 grid shape, with various data locality choices. Performance degrades with the deviation from scatter-scatter distribution, but by not more than a factor of two in the worst case.

8 Abstraction vs. Performance

One of the clearest lessons of our work thus far, is that abstractions such as the gather/send, receive/scatter semantics of message-passing, because they are expressive, open the way for greater optimization, at the same time they provide the user with greater expressivity, and ease of programming. So, it is not true that higher levels of abstraction always imply less performance, as is commonly held.

The invoice (or buffer-descriptor) semantics allow the total encapsulation of heterogeneity within the calls, removing expensive data motion or conversion when it proves unnecessary (such as when an application is used on a homogeneous subset of machines). Furthermore, the careful binding of a communication context (*Zipcode* mailer) to such calls provides a means to maintain ("cache") appropriate methods, and architectural information about the group of communicating processes. Such information (such as the realization that a context is homogeneous, or in a single memory hierarchy) could be derived at runtime.

The lesson is that moving to a "what-I-want" not "how-I-want-it-done" approach to message passing (which incidently removed buffer structuring from user control), made message passing less error prone, potentially much faster, and simultaneously easier to understand. The limitations of C as the implementation language are relevant in this discussion. In C++ we could discover optimizations at compile-time because of tighter type checking (and overload more appropriate operators), runtime optimizations are no longer our only avenue of improved performance. C++ would open the way for inlining, and would also help instigate much safer message-passing constructs. In fact, the combination of contexts of communication, virtual topologies, and gather/send, receive/scatter semantics could be quite effective (*e.g.*, "a Zipcode++" system), but most of the benefits would be lost if the program were not entirely in C++ (because operator overloading would be lost, and type checking would have to be sacrificed). The invoice-oriented message-passing constructs, plus C++ extensions to allow data-only structs to be transferred would provide a reasonably simple extension to C++ for parallel computing. Such a system is certainly within our immediate reach.

9 Summary and Conclusions

In this paper, we have raised issues that emerge when trying to create multicomputer libraries and when trying to move to the fully heterogeneous domain. We have touched on several issues:

namely, process control, communication context control, and the semantics for how messages should be transmitted in order to encapsulate heterogeneity. We used our own software, *Toolbox/Zipcode* to motivate this discussion. Future work will include large-scale demonstrations of this software technology on heterogeneous platforms. We drew analogies between *Zipcode* and MPI1. We discussed several aspects of the mathematical libraries, including data structures, and interfaces. We indicated that a few substantial *Toolbox* applications already exist, though we have not released the software publicly; we expect serious interest once we get the software productized (read: manuals written) and onto anonymous ftp and **netlib**.

Acknowledgements

The first author acknowledges financial support by the NSF Engineering Research Center for Computational Field Simulation (NSF ERC). Mississippi State University. We acknowledge Eric Van de Velde, of Caltech, who provided (more than six years ago) the initial software and encouragement that motivated us to make the *Multicomputer Toolbox*, and who has consistently insisted on the importance of data distribution independence.

References

[1] A. Beguelin, G. A. Geist, W. Jiang, R. Manchek, K. Moore, and V. Sunderam. The PVM project. Technical report, Oak Ridge National Laboratory, February 1993.

[2] Scott Berryman, James Cownie, Jack Dongarra, Al Geist, Bill Gropp, Rolf Hempel, Bob Knighten, Rusty Lusk, Steve Otto, Tony Skjellum, Marc Snir, David Walker, and Steve Zenith. Draft document of the MPI standard. Available on **netlib**, May 1993.

[3] M. L. Branstetter, J. A. Guse, D. M. Nessett, and L. C. Stanberry. An ELROS primer. Technical report, Lawrence Livermore National Laboratory, October 1992.

[4] Robert D. Falgout, Anthony Skjellum, Steven G. Smith, and Charles H. Still. The *multicomputer toolbox* approach to concurrent BLAS and LACS. In J. Saltz, editor, *Proc. Scalable High Performance Computing Conf. (SHPCC)*, pages 121–128. IEEE Press, April 1992. Also available as LLNL Technical Report UCRL-JC-109775.

[5] Charles L. Seitz et al. The C Programmer's Abbreviated Guide to Multicomputer Programming. Technical Report Caltech-CS-TR-88-1, California Institute of Technology, January 1988.

[6] Jakov Seizovic. The Reactive Kernel. Technical Report Caltech-CS-TR-88-10, California Institute of Technology, 1988.

[7] Anthony Skjellum. The Design and Evolution of Zipcode. *Parallel Computing*, 1993. (Invited Paper, to appear).

[8] Anthony Skjellum, Steven F. Ashby, Peter N. Brown, Milo R. Dorr, and Alan C. Hindmarsh. The Multicomputer Toolbox. In G. L. Struble et al., editors, *Laboratory Directed Research and Development FY91 – LLNL*, pages 24–26. Lawrence Livermore National Laboratory, August 1992. UCRL-53689-91 (Rev 1).

[9] Anthony Skjellum and Chuck H. Baldwin. *The Multicomputer Toolbox:* Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[10] Anthony Skjellum, Chuck H. Baldwin, Charles H. Still, and Steven G. Smith. The Multicomputer Toolbox on the Delta. In Tiny Mihaly and Paul Messina, editors, *Proc. of the First Intel Delta Applications Workshop*, pages 263–272. Caltech Concurrrent Supercomputing Consortium CCSF-14-92, February 1992.

[11] Anthony Skjellum and Alvin P. Leung. LU factorization of sparse, unsymmetric Jacobian matrices on multicomputers: Experience, strategies, performance. In *Proc. Fifth Distributed Memory Computing Conf. (DMCC5)*, pages 328–337. IEEE, April 1990.

[12] Anthony Skjellum and Alvin P. Leung. Zipcode: A portable multicomputer communication library atop the Reactive Kernel. In *Proc. Fifth Distributed Memory Computing Conf. (DMCC5)*, pages 767–776. IEEE, April 1990.

[13] Anthony Skjellum and Manfred Morari. Concurrent DASSL applied to dynamic distillation column simulation. In *Proc. Fifth Distributed Memory Computing Conf. (DMCC5)*, pages 595–604. IEEE, April 1990.

[14] Anthony Skjellum, Steven G. Smith, Charles H. Still, Alvin P. Leung, and Manfred Morari. The Zipcode Message-Passing System. In Geoffrey C. Fox, editor, *Parallel Computing Works!* Morgan Kaufmann, 1992. (Also as LLNL UCRL-JC-112022) [To appear in February, 1994].

[15] Anthony Skjellum and Charles H. Still. Zipcode: and the Reactive Kernel for the Caltech Intel Delta Prototype and nCUBE/2. In *Proc. Sixth Distributed Memory Computing Conf. (DMCC6)*, pages 26–33. IEEE, April 1991. Also available as LLNL Technical Report UCRL-JC-107636.