The Multicomputer Toolbox: Current and Future Directions

Anthony Skjellum
Computer Science Department & NSF Engineering Research Center
Mississippi State University
Mississippi State, MS 39762

Abstract

The Multicomputer Toolbox is a set of "first-generation" scalable parallel libraries. The Toolbox includes sparse, dense, direct and iterative linear algebra, a stiff ODE/DAE solver, and an open software technology for additional numerical algorithms. The Toolbox has an object-oriented design; C-based strategies for classes of distributed data structures (including distributed matrices and vectors) as well as uniform calling interfaces are defined.

At a high level in the Toolbox, data-distribution-independence (DDI) support is provided. DDI is needed to build scalable libraries, so that applications do not have to redistribute data before calling libraries. Data-distribution-independent mapping functions implement this capability. Data-distribution-independent algorithms are sometimes more efficient than fixed-data-distribution counterparts, because redistribution of data can be avoided. Underlying the system is a "performance and portability layer," which includes interfaces to sequential BLAS, the Zipcode *message passing system, and a minimal set of Unix-portability functions. In particular, the* Zipcode *system provides communication contexts, and process groups, collective operations, and virtual topologies, all needed for building efficient scalable libraries, and large-scale application software.*

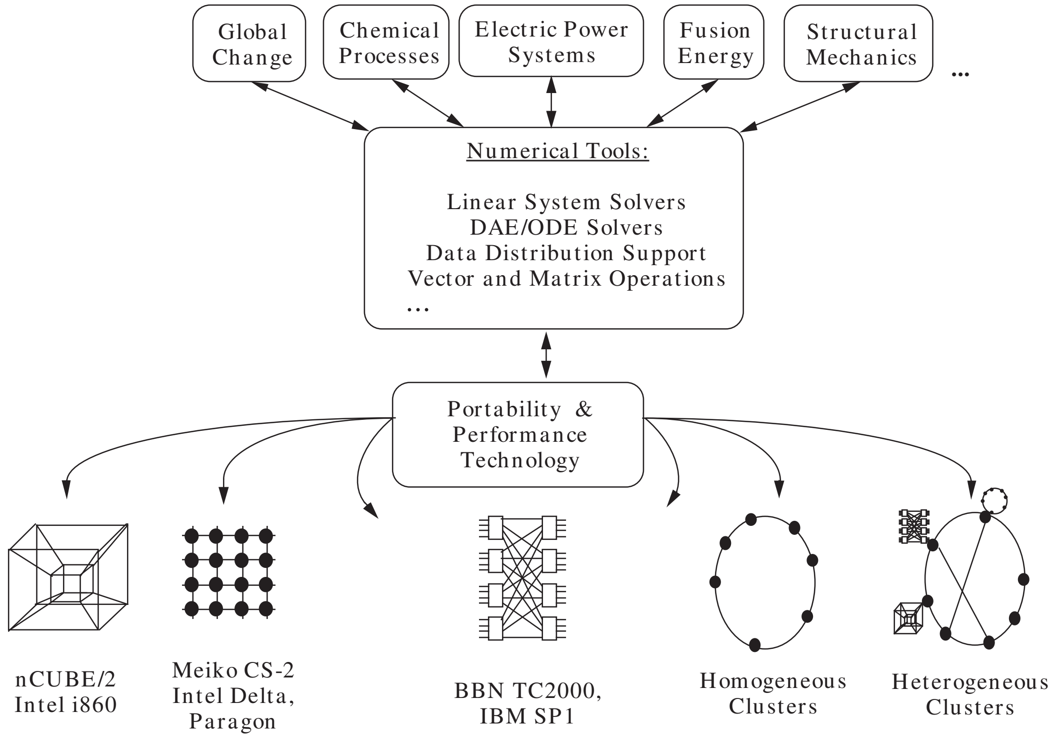
1 Introduction

The *Multicomputer Toolbox* is a set of "first-generation" scalable parallel libraries [12, 13, 14]. The *Toolbox* includes sparse, dense, direct and iterative linear algebra, a stiff ODE/DAE solver, and an open software technology for additional numerical algorithms. The *Toolbox* has an object-oriented design; C-based strategies for classes of distributed data structures (including distributed matrices and vectors) as well as uniform calling interfaces are defined.

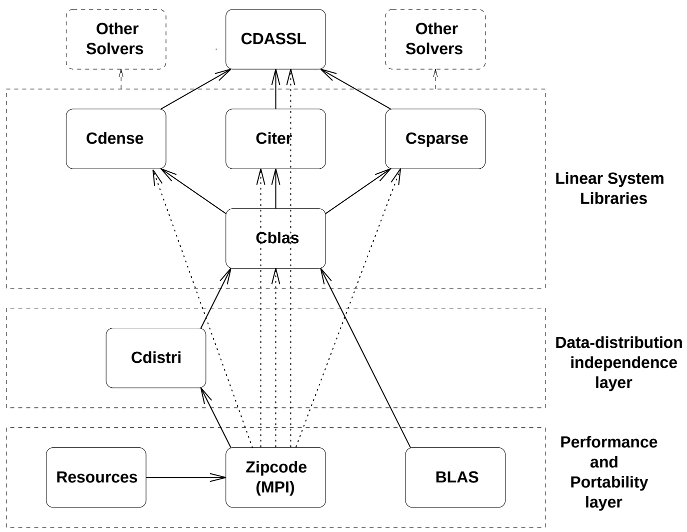
1.1 Issues Addressed

The development of parallel applications is an inherently top-down activity starting with problem and performance requirements, and working toward code that executes on a parallel architecture. Conversely, the development of libraries is inherently a bottom-up activity, abstracting architecture and operations to provide higher level development platforms for codes. To resolve these seemingly conflicting features of application and library design, parallel libraries must be both efficient and provide flexible application programmer interfaces. Figure 1a shows applications at the high level, the *Toolbox* libraries in the middle (signifying software reuse by applications), all relying on a performance and portability layer.

Figure 2 shows conceptually that specific multicomputer and cluster architectures are abstracted to fully connected networks, with particular latency and bandwidth properties. This abstraction is needed to avoid writing libraries that are intimately tied to a specific network topology, and therefore have limited portability to other architectures. Figure 3 illustrates the problem of large-scale software development; parallel software, like sequential software, must often be constructed from pieces of diverse origins. As such, resource utilization of each piece must be managed so that the pieces do not need significant re-engineering when they are connected. Tied to this issue of resource allocation is the most obvious one that arises in the distributed memory regime: message-passing conflicts, and multiple invocations of libraries. Figure 4 illustrates this idea schematically for a global climate model, which can conceivably use many kinds of messages, and rely on the message-passing interface to provide modularity between sections of code. Libraries working in such an environment must be able to protect themselves from stray messages, and other library invocations. To achieve this goal, specific semantics for message-passing, and parallel library development are needed.



1a. A conceptual overview of the *Toolbox* is depicted. At the high level, applications interface to flexible libraries. At the low level, architectures are abstracted by the performance and portability technology. The key to merging the bottom-up view of library programming, and the top-down view of applications comes through data distribution independence and the *Toolbox*'s object-oriented design.



1b. A hierarchical view of the *Toolbox* libraries.

Figure 1: Two views of the *Multicomputer Toolbox*'s structure.

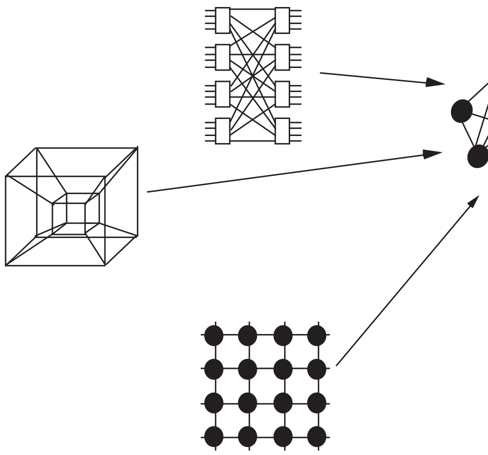


Figure 2: The *Toolbox* provides an abstraction of the computational model.

The goal of performance optimization is to minimize the time spent in the critical path of a computation. The data-distribution-independent approach allows for optimization by explicit data redistribution when absolutely necessary, and for compromise data distributions when this proves faster overall. Figure 5 illustrates the steps through the critical path of a hypothetical application. Each stage of a computation has a locally optimal data distribution (locality of data, number of processes, etc.). However, because the optimal distribution differs in general from step to step, compromises must be made to reduce the overall time. Libraries that enforce explicit redistribution to fixed data formats (that they define) cannot realize the benefits of distribution compromises, nor can they utilize "free" communication that occurs during the natural course of typical loosely synchronize algorithms. In general, the data distribution independent approach is the most likely to be runtime optimizable. (For applications that tend to use one or a few, fixed distributions, general libraries can also be specialized for higher performance, but the data distribution independent format leads to the correct design choices for second-order optimizations of this kind.)

1.2 System Structure

Figure 1b illustrates the hierarchical layering of the *Toolbox* libraries, while showing further structure; Figure 6 provides a list of the libraries currently supported. At the highest level come libraries such as *CDASSL*, which utilize message passing, sequential BLAS, as well as Concurrent Linear algebra in the form of solvers. While *CDASSL* is the only "higher

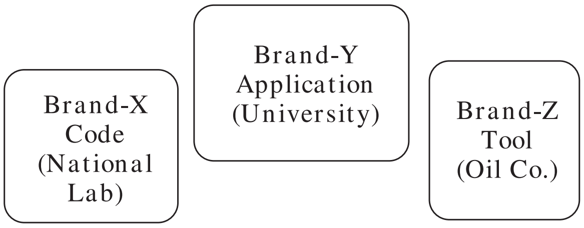


Figure 3: The *Toolbox* recognizes the need for building large-scale codes from diverse sources.

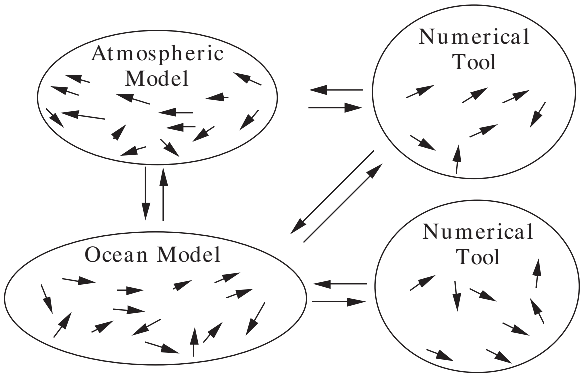


Figure 4: Large-scale codes involve multiple invocations of libraries, and overlapping communication contexts. The *Toolbox* supports this complexity.

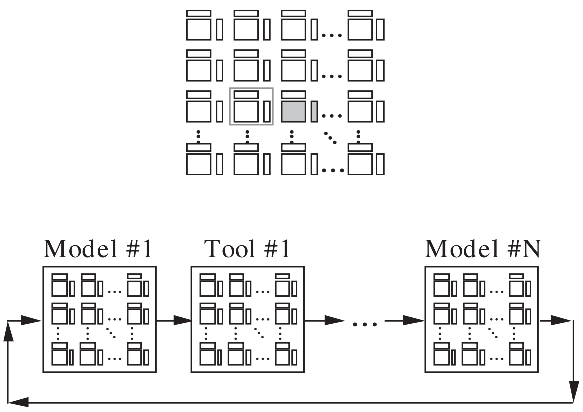


Figure 5: The goal of performance optimization is to minimize the time spent in the critical path of a computation. The data-distribution-independent approach allows for optimization by explicit data redistribution when absolutely necessary, and for compromise data distributions when this proves faster overall.

level" library at present, we plan to build more libraries of this type.

The linear algebra layer contains solvers at the higher level, and basic concurrent linear algebra and data motion operations at a lower level. At present, the solver level is further specialized into dense and sparse direct solvers, and to Krylov-type iterative solvers with general matrix-vector-operator interfaces, capable of use in dense, sparse, and matrix-free applications. The lower level is currently represented by two systems that are being merged into the "Concurrent BLAS" or CBLAS library. CBLAS includes vector operations and data motion, but also includes matrix-vector and matrix-matrix multiplication for general, non-square systems operating on non-square virtual topologies, with flexible data distributions.

Conceptually below this library is the data-distribution-independence (DDI) support, which permits the higher level libraries to be flexible about how data is stored. DDI is needed to build scalable libraries, so that applications do not have to redistribute data before calling libraries. Data-distribution-independent mapping functions implement this capability. Data-distribution-independent algorithms are sometimes more efficient than fixed-data-distribution counterparts, because redistribution of data can be avoided.

Underlying this is a "performance and portability layer," which includes interfaces to sequential BLAS, the *Zipcode* message passing system[1], and a minimal set of Unix-portability functions. In particular, the *Zipcode* system provides communication contexts, static process groups, collective operations, and virtual topologies, all needed for building efficient scalable libraries, and large-scale application software.

1.3 Algorithmic Scope

The *Multicomputer Toolbox* supports the libraries described in Figure 6. Notable current omissions are FFTs, QR factorizations, and eigensolvers. We hope to close some of these gaps in the future. We are also confident that the data distribution independent paradigm will benefit these additional algorithms.

1.4 Applications

The applications that currently use *Toolbox* technology are described in Figure 7. These codes have run on several different configurations on machines including the Intel Delta, Paragon, nCUBE2, and networks

[1] We will support MPI soon.

Toolbox Libraries Currently Supported	
CDASSL	Concurrent Differential-Algebraic Solver [18],
Citer	Krylov-subspace methods for linear system solution,
Csparse	Sparse LU solvers [15],
Cdense	Dense level-2 and level-3 LU solvers (see section 3),
Cblas	Concurrent BLAS library (in development) [6, 7],
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.

Figure 6: *Toolbox* libraries currently supported include high-, medium-, and low-level numerical libraries, as well as portability and performance libraries.

Current *Toolbox*-based Applications		
Name	Application	Institution
Ardra	Neutron Transport [5]	LLNL
Parflow	Groundwater Modeling [2]	LLNL
Cdyn	Process Flowsheeting [10]	MSU

Figure 7: At present, a few applications have been developed for the *Toolbox* but we expect many more to be developed in the future.

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 [1, 9].

1.5 Paper Organization

The paper is organized as follows. We have already given much about the structure of the *Multicomputer Toolbox* in the Introduction. In section 2, we describe the "performance and portability layer" of the *Toolbox* already introduced. Following this, section 3 describes the data distribution independence support of the *Toolbox*, introduced above. We discuss library design and implementation issues in section 4. Related papers on linear system libraries are mentioned in section 5.1.1, with "higher level libraries" covered in section 5.1.2. Future work is summarized in section 5.2.

2 Performance and Portability Layer

2.1 BLAS Interface

Data distribution independence is a general requirement that does not necessarily yield contiguous blocks of data in local data structures. The sequential BLAS that are currently in use in many systems do not support strided and non-contiguous data well. However, their use is essential in order to secure high performance floating point on most systems. "BLAS compatibility" is achieved in our linear solvers by requiring that pivots eliminate submatrices in an ascending or descending order (coefficients of compatible distributions have the same local and global ordering). This restriction has not been significant. However, when working with linear algebra generated in the GMRES system, the restriction of BLAS calls affects the performance of CBLAS layered on top of them. This can only be fixed by extending the definition of BLAS to support both row-major and column-major data with equal stature, as well as by the inclusion of more general striding. These new calls will evidently have to be standardized so that vendors make them efficient.

2.2 Resources

Not all of the systems supported provide exactly the same set of Unix libraries, because some are based on System V, some on POSIX, etc. The purpose of this library is to provide minimal annexes to what a vendor offers to allow *Toolbox* libraries to execute on new systems. This system defines, for instance, random number generators and stubs for `getrusage()` on systems where it might not exist.

2.3 *Zipcode* Communication System

Zipcode , whose specific design purpose was to support parallel libraries in parallel applications, provides basic services useful for library management. *Zipcode* assumes a multiple-instruction, multiple-data programming model. 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, a user could define his or her own library for an asynchronous collective operation).

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. 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 or other graph topology, for instance, to make them most natural to program.

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 MPI standard [8, 16]. 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* [11, ?, 21]. 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.

3 Data Distribution Independence

3.1 Building DDI 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 communication contexts, including the two-dimensional grid, and its logical one-dimensional row/column children.

Cdistri includes the following constructors based in part on grid mailers:

```
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 this date.

```
typedef struct _Cdistrib
{
  ZIP_MAILER *g2mlr; /* grid mailer */
  /* row or column mailer + type */
  ZIP_MAILER *rcmlr;
  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 */
  /* extra info. needed by mappings */
  void       *mu_extra;
  /* problem size global/local data */
  Cdistrib_data *data;
} Cdistrib;
```

The above structure defines the basic one-dimensional data mapping of elements onto a process topology (either rows or columns). Two of them are consequently needed to define a two-dimensional cartesian mapping. Notice that the "mu" functions define a one-to-one and onto mapping of the global coefficients onto local coefficients. In [18], we discuss weakened requirements for data mappings that also result in correct linear operations; at present, only the "strong" mappings are used in practice. The `CMdistrib` data structure, described next, uses two `Cdistrib` data structures to assemble the cartesian mapping; it also utilizes extra data specified in `CMdistrib_data *data` to specify problem size information (number of coefficients). Contained within `CMdistrib_data` is the `Inv_proj` data structure, that stores mapping invariants between rows and columns. The latter is used by data remapping operations; a `CMdistrib` is more than the sum of two `Cdistrib`'s.

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

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

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

which are based, in turn, on the following:

```
typedef struct _Inv_proj_entry
{
  /* global name of invariant */
  int  Inv;
  /* local names in row/col sets */
```

```
  int i, j;
} Inv_proj_entry;

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

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 elements 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.

3.2 Data Distribution Functions

The following are the actual functions used to define linear, load-balanced data distributions in the *Toolbox*. See also [10, 13].

```
/* linear distribution function family: */

/* "mu": */
void cdistri_linear(I,p,i,P,M, extra)
int I;
int *p, *i;
int P, M;
void *extra;
{
        int L, R;
        int arg1, arg2;
        L = M / P;
        R = M % P;
        arg1 = I/(L+1); arg2 = (I-R)/L;
        *p = max(arg1, arg2);
        *i = I - (*p)*L - min(*p,R);
}

/* "mu_inv": */
int cdistri_inv_linear(p,i,P,M, extra)
int p, i;
int P, M;
void *extra;
{
        int L, R;
        L = M / P;
        R = M % P;
        return (p*L + min(p,R) + i);
}

/* "mu_lim": */
int cdistri_lim_linear(p,P,M, extra)
int p;
int P, M;
void *extra;
{
        return ( (M+P-p-1)/P );
}

/* "mu_init": */
int cdistri_init_linear(extra)
void *extra;
{
        return(0);
}
```

4 *Toolbox* Library Issues

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 possibly by collective communications implemented by a vendor. Having a programmer publish the "range of tags" utilized by a library, 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 implement 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 unknown initially), an additional communication context is needed. For each level of stack depth of libraries called, an additional context of communication is potentially needed. For each overlapping pair of process groups, separate contexts must be defined for safe communication.

4.2 Objects & Interactions

Thus far, we have written numerical libraries, such as dense matrix-vector multiplication, to utilize pairs or triplets of distributed objects. A dense matrix is distributed on a two-dimensional virtual topology, by specifying a 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 contexts of communication 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 contexts of communication separately. This is so that the equality test can be satisfied in libraries that do error checking for compatible distributed objects.

4.3 Implementing Data Conversion

In the fully heterogeneous environment, data conversion is needed within all heterogeneous communication contexts. We have achieved the following goals:

- 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 [11, 21]; this approach ports trivially to the MPI environment [8].

4.4 Abstraction vs. Performance

One of the clearest lessons of our work thus far is that abstractions nsuch as the gather/send, receive/scatter semantics of message-passing open the way for greater runtime optimization, at the same time they provide the user with greater expressivity, and ease of programming. Abstraction need not imply less performance, as is commonly thought.

The invoice (data types in MPI) 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 part of a non-uniform memory architecture hierarchy) could be determined at runtime.

5 Summary and Conclusions

In this paper, we have discussed the Multicomputer Toolbox, a set of "first-generation" scalable parallel libraries. The Toolbox includes sparse, dense, direct and iterative linear algebra, a stiff ODE/DAE solver, and an open software technology for additional numerical algorithms. The Toolbox has an object-oriented design; C-based strategies for classes of distributed data structures (including distributed matrices and vectors) as well as uniform calling interfaces are defined. Data-distribution-independence (DDI) support is provided. DDI removes the need for applications to redistribute data before calling libraries. A "performance and portability layer" is also provided, which includes interfaces to sequential BLAS, the *Zipcode* message passing system, and a minimal set of Unix-portability functions.

5.1 Where to Learn More

The following brief descriptions point at several others papers that cover the *Toolbox* in further detail.

5.1.1 Linear System Libraries

Toolbox Linear system libraries are described elsewhere in much greater detail than is possible here. Sparse direct linear algebra is covered in [10, 15]. Dense LU factorization is covered in [3, 4, 10, 13]. Concurrent BLAS are covered in [6, 7]. Krylov iterative solvers are covered in [17] and are mentioned also in [9]. The basic concurrent vector operations and data motion operations are detailed in [10] and are discussed further in [3, 13].

5.1.2 Higher Level Libraries

The only "higher level" library at present is CDASSL, introduced in [20], and discussed further in [9, 10, 18]. Plans for further work, to increase the number of such higher level libraries, is mentioned in [1, 9].

5.2 Future Developments

One of the lessons we have learned is that moving to a "what-I-want" versus "how-I-want-it-done" approach to library interfaces makes programming with libraries 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 some 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 Toolbox++/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). We would call such software the "second-generation" of scalable parallel libraries; our attention will turn to this effort as soon as the first-generation software is released.

Acknowledgements

The 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 and CRPC, who provided (more than six years ago) the initial software and encouragement that motivated us to make the *Multicomputer Toolbox*.

A complete, alphabetical list of *Toolbox* authors follows: Chuck H. Baldwin (UIUC), Purushotham V. Bangalore (MSU), Nathan E. Doss (MSU), Robert D. Falgout (LLNL), Alvin P. Leung (Syracuse University), Steven G. Smith (LLNL), Charles H. Still (LLNL).

References

[1] Kamala Anupindi, Anthony Skjellum, Paul Coddington, and Geoffrey Fox. Parallel Differential-Algebraic Equation Solvers for Power System Transient Stability Analysis. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[2] Steven F. Ashby, Robert D. Falgout, Steven G. Smith, and Andrew F. B. Tompson. Modeling Groundwater Flow on MPPs. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[3] Purushotham V. Bangalore, Anthony Skjellum, Chuck Baldwin, and Steven G. Smith. Dense and Iterative Linear Algebra in the Multicomputer Toolbox. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[4] Purushotham V. Bangalore, Anthony Skjellum, Chuck Baldwin, and Steven G. Smith. Data-Distribution-Independent, Concurrent Block LU Factorization. *In preparation.*, January 1994.

[5] Milo R. Dorr and Charles H. Still. A Concurrent, Multigroup, Discrete Ordinates Model of Neutron Transport. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[6] 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.

[7] Robert D. Falgout, Anthony Skjellum, Steven G. Smith, and Charles H. Still. The Multicomputer Toolbox Approach to Concurrent BLAS. *Submitted to Concurrency: Practice & Experience.*, October 1993.

[8] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report Technical Report No. CS-93-214., University of Tennessee, November 1993. Available on **netlib**.

[9] Alvin P. Leung, Anthony Skjellum, and Geoffrey Fox. Concurrent DASSL: A Second-Generation DAE Solver Library. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[10] Anthony Skjellum. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic Process Systems in Chemical Engineering*. PhD thesis, Chemical Engineering, California Institute of Technology, May 1990.

[11] Anthony Skjellum. The Design and Evolution of Zipcode. *Parallel Computing*, 1993. (Invited Paper, to appear in Special Issue on Message Passing).

[12] 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).

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Anthony Skjellum, Alvin P. Leung, Charles H. Still Steven G. Smith, Robert D. Falgout, and Chuck H. Baldwin. The Multicomputer Toolbox – First-Generation Scalable Libraries. In *Proceedings of HICSS–27*. IEEE Computer Society Press, 1994. HICSS–27 Minitrack on Tools and Languages for Transportable Parallel Applications.

[18] 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.

[19] Anthony Skjellum and Manfred Morari. Zipcode: A Portable Communication Layer for High Performance Multicomputing. Technical Report UCRL-JC-106725, Lawrence Livermore National Laboratory, March 1991. To appear in Concurrency: Practice & Experience.

[20] Anthony Skjellum, Manfred Morari, Sven Mattisson, and Lena Peterson. Concurrent dassl: Structure, application, and performance. In *Proceedings of the Third Conference on Hypercubes, Concurrent Computers, and Applications (HCCA4)*, pages 1321–1328. Golden Gate Enterprises, April 1990.

[21] 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].