Scalable Libraries in a Heterogeneous Environment

Anthony Skjellum
Computer Science Dept. & NSF Engineering Research Center
Mississippi State University
PO Drawer CS
Mississippi State, MS 39762

Abstract

This paper concerns itself with efforts to extend multicomputer libraries to a hierarchical, heterogeneous network environment. Two classes of support for such libraries are discussed: first, the message-passing features needed to establish groups of communicating processes, and communication contexts within which libraries can safely work. Second, we discuss message-passing primitives that encapsulate heterogeneity, hiding it from the user program (and library alike), and eliminating it when it proves unnecessary (within a homogeneous invocation, for instance). The **Multicomputer Toolbox** *first-generation scalable libraries, and* **Zipcode** *message-passing systems are the means by which we demonstrate our research, so they are discussed here. We relate* **Zipcode** *syntax and semantics to the emerging MPI standard, when appropriate.*

1 Introduction

First-generation scalable libraries have been developed within the *Multicomputer Toolbox* schema, described elsewhere [10, 11, 9]. 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 [8]), 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 [3].

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 how our invoice notation (aka buffer descriptors) contribute to data-structure-oriented conversions. Reusability of invoices (gather/scatter specifications), and their creation outside inner loops, helps reduce overheads. The use of the ELROS system to implement efficient data conversions is mentioned, as an alternative to XDR, which cannot vectorize/pipeline. We quickly conclude that XDR is a deadend for high-performance data conversion.

We indicate how C++-based systems would further strengthen our efforts, and how abstractions in the current work are seen as a means not only to hide the complexity of heterogeneity, or to manage it, but also as a means to discover runtime optimizations that cannot be discovered with low-level semantics found in other systems. Such work will result in the second-generation scalable mathematical libraries we hope to complete in 1994 and 1995.

2 Language Bindings

Nodal C/C++, Fortran-77, or mixed C, C++, and Fortran-77 programming are the currently most viable approaches to programming on multicomputers / distributed computers (with multicomputer C++ still lagging behind workstation support for it), with a few exceptions. Support for data structures in Fortran-77 is absent, yet scalable libraries need complex data structures to encapsulate communication scope, data layout, methods, and architectural tuning. We do not address the complex issue of supporting Fortran-77 with *Multicomputer Toolbox* libraries here, except to say that macro preprocessing of Fortran-77 will definitely be needed, to provide a reasonable "look and feel" (*e.g.*, `m4`).

3 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, 7]. 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, 2]; integration of *Zipcode* with ELROS messaging capabilities is also being undertaken [4]; 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; the PVM systems also provides this capability, but with different semantics (*e.g.*, after creation, processes join named groups that are cached by dæmons, with possible race conditions):

```
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, 3]. 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, 13, 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.

3.6 Zipcode vs. MPI1

A communication context degrades the raw performance of any message-passing system, in return for useful guarantees of program correctness, and manageability; how much degradation results depends on hardware characteristics, number of processes per processor, and whether additional queueing is required to support contexts. This speaks to the need for contexts in vendor primitives, rather than just in a user-level library such as *Zipcode*. With vendor implementations of MPI1, lighter-weight context support will become a reality, and *Toolbox/Zipcode* applications will increase performance either by moving directly to MPI1 primitives, or by our planned light-weight port of *Zipcode* that will run on top of MPI1. We view this evolution as highly satisfactory, as it has provided good portability, performance enhancement to the *Multicomputer Toolbox* libraries. Therefore, no *Zipcode*-based application or library will suffer in face of the standardization, but will instead benefit almost immediately by it, and be even more portable, as multiple vendors will eventually implement MPI1, whereas we were obliged to undertake each *Zipcode* port as new hardware became available.

For reasons of development latency (MPI1 is very large, and vendors have limited resources), and because *Zipcode* provides process management (whereas MPI1 does not), we will continue actively to develop, distribute, and support *Zipcode*, for the next one to two years, and perhaps longer. Particularly, the fully heterogeneous environment has not been addressed or explored fully in practice, and MPI1 does not address dynamic communication contexts. All of these factors suggest that new *Zipcode* features can potentially provide a continued useful role in providing practical input to the 1994 MPI2 effort, while remaining a "full" portability platform. Of course, we hope that MPI2 will include dynamic context support, and process control, and strict guidelines for multi-MPI2-implementation interoperability.

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 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, 8].

5.1 XDR vs. Bandwidth

We note that XDR is a bad system for handling vectors of data conversions because it imposes a function-call overhead for each datum converted, and vendor-supplied basic operations have these semantics as well (no argument for repeat count). So, a system like ELROS [4], which writes pipelineable, or vectorizable conversion code, provides much better bandwidth to the overall communication, than can XDR. Even if ELROS's code does not vectorize or pipeline on a machine, just moving the function call overhead outside the do-loop increases performance dramatically. Though ELROS is an embedded language, it could be used to write the conversion code used by invoices in *Zipcode*. Advanced users could employ ELROS and *Zipcode* together for more complex, hierarchical data structures.

At most, XDR should be used for low-volume negotiations, not for long-haul communications. Otherwise, high bandwidth links will not realize their full potential. A version of XDR that allowed multiple items to be converted per function call, would also be a big step in the right direction. Actually, what one wants is for vendors to implement their conversion operations in assembly language, with such an option. (Then, under *Zipcode*, for instance, something much simpler than XDR can be used to work on messages, because none of this is revealed by *Zipcode* to the user. Everything is done opaquely via invoices. See [8, 14].)

6 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. The availability of inlining suggests a reduction in the minimum overhead required to effect a message-passing call in C++ compared to C (though C could inline as well), with concommitant potential for better user-level performance. It should be noted that Fortran's minimum granularity is a function call, so using Fortran-77 for message-passing implies a performance hit compared to inlined C++ (or inlined C) calls: message-passing will be more efficient from inlined C/C++ than from Fortran-77, if implemented properly.

7 Summary and Conclusions

In this paper, we have raised issues that emerge when trying to move multicomputer libraries into 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*, MPI1, and PVM, to relate these efforts, but this paper was in no way meant as a sincere comparison of these systems.

Acknowledgements

The author acknowledges financial support by the NSF Engineering Research Center for Computational Field Simulation (NSF ERC). Mississippi State University. The author acknowledges the contributions of Steven G. Smith, and Charles H. Still (both of Lawrence Livermore National Laboratory, Livermore, CA) as well as Nathan E. Doss (NSF ERC).

References

[1] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM: Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[2] 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.

[3] 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.

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

[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] Charles L. Seitz, Sven Mattisson, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig M. Steele, and Wen-King Su. The architecture and programming of the ametek series 2010 multicomputer. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3)*, pages 33–36. ACM Press, January 1988. (Symult s2010 Machine).

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

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

[9] 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).

[10] 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.

[11] 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.

[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. 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.

[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*. 1992. (Also as LLNL UCRL-JC-112022).

[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.