The Design and Evolution of *Zipcode*[*]

Anthony Skjellum

Computer Science Department &
NSF Engineering Research Center for Computational Field Simulation
Mississippi State University

Steven G. Smith

Numerical Mathematics Group
Lawrence Livermore National Laboratory

Nathan E. Doss

NSF Engineering Research Center for Computational Field Simulation
Mississippi State University

Alvin P. Leung

Northeast Parallel Architectures Center
Syracuse University

Manfred Morari

Chemical Engineering
California Institute of Technology

March 8, 1994

Abstract

Zipcode is a message-passing and process-management system that was designed for multicomputers and homogeneous networks of computers in order to support libraries and large-scale multicomputer software. The system has evolved significantly over the last five years, based on our experiences and identified needs. Features of *Zipcode* that were originally unique to it, were its simultaneous support of static process groups, communication contexts, and virtual topologies, forming the "mailer" data structure. Point-to-point and collective operations reference the underlying group, and use contexts to avoid mixing up messages. Recently, we have added "gather-send" and "receive-scatter" semantics, based on persistent *Zipcode* "invoices," both as a means to simplify message passing, and as a means to reveal more potential runtime optimizations. Key features in *Zipcode* appear in the forthcoming MPI standard.

Keywords: Static Process Groups, Contexts, Virtual Topologies, Point-to-Point Communication, Collective Communication, Data Types, Parallel Libraries, Large-Scale Software, Portability, Zipcode, MPI

[*]Work performed, in part, under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48. Work supported, in part, by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University.

1 Introduction

Zipcode was developed in 1988 at the California Institute of Technology by the first author, and was developed further at Lawrence Livermore National Laboratory, with additions and support continuing to this date at Mississippi State University [29]. This message-passing system was strongly influenced by the point-to-point semantics of the Reactive Kernel primitives [23, 24, 25], by the collective primitives of CrOS [15, Chapter 14], and by the process-management features of the Cosmic Environment [23] (all developed at Caltech). *Zipcode* includes features that were not found in then-existing vendor systems (like NX-2 [22]) and portability systems (like PICL [18]). Many of the features of *Zipcode*, particularly those that set it apart from other systems, are those that support parallel libraries. *Zipcode* is currently the effective basis for a collection of parallel libraries, called the "Multicomputer Toolbox," which we describe elsewhere [13, 26, 27, 30]. Its development also commenced in 1988, and its requirements largely drove the evolution of *Zipcode*.

Because key features in *Zipcode* that are needed to support libraries will also appear in the new MPI standard, we foresee further, rapid evolution in our emphasis for future message-passing research and development [14]. The purpose of this paper is, however, to describe the design and evolution of *Zipcode*, as well as to explain in part how we implemented the system. These features are salient now, as they were when we started. In some cases, our design strategies have differed significantly from what MPI has evolved, while in other areas only details are different. As MPI is the product of many minds, while *Zipcode* is the product of three main developers, with several other less-active participants, *Zipcode* is naturally less substantial in some areas than MPI. However, a discussion of *Zipcode* remains important because this system has demonstrated with working code what remains to this date hypothetical in the MPI draft standard. In fact, four of the five key contributions: contexts of communication, static process group support, mailers (called communicators in MPI), and virtual topology support were all completed (and put in practice) in 1988 and early 1989. Collective operations were supported over user-specified groups of processes from the beginning. Contexts, which provide separate, safe "universes" of message passing, were one of the first features implemented during August, 1988. In mid-1992, we added the concept of persistent gather/scatter specifications (invoices); this corresponds to MPI user-defined data types for gather/scatter message-passing [14, Section 3].

1.1 Related Work

In order to provide a fair perspective of work on multicomputer and cluster message passing, we wish to acknowledge related work in the field. This should also give readers a better perspective of how *Zipcode* fits in with the many other systems that are currently or were previously discussed in the literature.

When we started, there were already a number of portability systems actively being built, or in place. The Reactive Kernel / Cosmic Environment was supported on homogeneous workstation networks, on Intel hypercubes, and as the native message-passing system of the Symult S2010 mesh multicomputer [24]. PICL (Oak Ridge National Laboratory) was quite popular at the time, providing point-to-point communication (send/receive), modest collective communication, and excellent tracing support [17, 18]. PICL represented the "lowest common denominator" of Intel and nCUBE calls, and ran principally on the early machines of these two vendors. Express, from Parasoft, was the commercial version of the CrOS system (CrOS influenced the syntax and semantics of *Zipcode* collective operations) [21]. At that time, or soon after, a number of other efforts were brought forth, such as P4 [9], and later PARMACS [6, 10]. It is notable that PARMACS included the notion of virtual topology about the same time as *Zipcode*, but with different details in how such topologies are implemented. Notably, none of these systems except *Zipcode* provided contexts of communication. The Reactive Kernel is the only system that provided message-passing with system allocation of messages, preferred because we wished to create a layered set of increasing functionality in our system, rather than user-managed buffers as the other systems dictated.

Process-group management involves deciding who participates in collective operations (and their relative ranks), as well as how point-to-point message passing names the source and destination addressees. Collective operations are message-passing operations involving process-group participants. For both of these areas, there was but limited support in vendor systems and portability systems alike except over the scope of "all processes." Related work has appeared in the last eighteen months from IBM T. J. Watson research center corroborating the value of process groups to specify the scope of collective operations without including processes that are disinterested (Venus, EUI systems [2, 3, 16]). It is our understanding that the ELAN widget library from Meiko, supported on the Computing Surface-2 system, also includes the notion of process group when considering collective operations [37]. Express also has a limited notion of groups and topologies [21].

PVM (University of Tennessee, and elsewhere), a portable interface to sockets and XDR [35], which became the most widely used cluster message-passing system during 1992, currently supports the notion of dynamic process groups, but not static groups; furthermore, it does not have significant support for collective operations [4, 5]. P4 added heterogeneity during this period, but remained less used than PVM, despite P4's perceived performance advantages (somewhat higher speed message passing).

In mid-1992, we turned our attention to improving the ability to formulate message-passing operations in Fortran-77 (where structured data types are absent), and to supporting data conversion. In the area of gather/scatter communicaton, *Zipcode* was influenced at first by the support provided by PVM. Particularly, we set out to hide the effort associated with calls to data conversion and incremental packing functions that we had seen in PVM 2.4, and which remains in PVM 3.x. Later, we realized that this notation would also be a useful mechanism to support higher performance message-passing, especially given the emphases of ELROS [7, 8], which strove for vectorization of message conversion on Cray platforms. (MPI, however, provides the most powerful specification through its data types [14, Section 3], motivated both for notational and performance reasons.)

Contexts of communication are distinct universes of message passing. In *Zipcode*, they are implemented by bringing context_ids, static process groups, and virtual topology information together in a mailer data structure. To our knowledge, *Zipcode* is the only system to this date actually to implement contexts for safe message passing. This long-term experience with contexts was a motivating force for the inclusion of contexts, groups, and communicators in the MPI specification.

Most systems (except the Reactive Kernel) provided a means to select messages at least on user-specified tags. The Reactive Kernel (RK) omitted this feature because of layerability design principles. NX, PICL, Intel NX provided tag-based selectivity. Some of these systems also supported selectivity based on the source process of the message (such as nCUBE's Vertex). We wanted the *Zipcode* naming approach to be related to the virtual topologies of processes or at least to a one-dimensional rank-naming of processes in a process group. To do this, we chose to support a generalized receipt selectivity scheme, based on a "mailer class," and include a variable-length matching field ("PO Box") with each letter sent by the system. Each MPI implementation specifies the number of bits in its tag; an early MPI implementation has chosen 31 bits for its tags [12].

At the outset in 1988, and even more so after new discussions during the MPI Forum, we were convinced that source selectivity is a clear requirement for making collective message passing operations (based on send/receive) work reliably. Source selectivity was consequently included in the primitives used to build collective communication in *Zipcode*, while tags were provided in but a subset of messaging strategies within *Zipcode*. MPI provides source-based selectivity in point-to-point messages (in addition to tags); point-to-point is based on rank naming, and virtual topology functions act as auxiliaries to support other naming schemes.

2 Motivations

The primary motivations for *Zipcode* were and remain as follows: to support parallel libraries and to support the creation of large-scale multicomputer software. Even in the sequential world, reusable, portable software is essential to most cost-effective, large-scale software.

2.1 The Need for Parallel Libraries

Parallel libraries are needed to hide the cloying details and technical complexities associated with parallel implementations of important algorithms (such as linear algebra, FFT). Libraries help ensure consistency in program correctness and quality of implementation, while offering a higher level of portability than a message passing system like *Zipcode* or MPI can provide alone. Libraries consequently prevent programmers from repetitive effort or haphazard results. Furthermore, since poly-algorithms (collections of algorithms that solve the same problem but that are more or less appropriate over different problem sizes and concurrencies) are commonly needed for scalability, parallel libraries have to hide such technical software complexities from the everyday user [27].

2.2 Common Deficiencies of Message Passing Systems

In the sequential Fortran and C environment, it is reasonably easy to create libraries, because the stack-oriented procedural programming model has well-defined conditions about reasonable versus erroneous programs. However, in the distributed-memory, message-passing environment, it is difficult to write libraries with either vendor or portability systems. This seemingly strong indictment is backed up as follows: There is no way for libraries to isolate themselves from the on-going point-to-point message passing present in a running application. Message tags, the sole means for designating restrictions on message delivery, are insufficient for this purpose. For instance, more than one library (or invocation of the same library) could use the same tags. Vendor libraries (that lack source selectivity) have to use tags to create deterministic collective operations (*e.g.*, NX-2 [22]); in such a situation, collective and point-to-point messages can be misqueued unless the system reserves tags for this purpose (as NX-2 fortunately does). Finally, wildcard receipt-selectivity on tags destroys any promise of real protection that tags could otherwise afford.

Beyond the protection issue just mentioned, library writers don't want to describe point-to-point communication in terms of hardware-related names; in fact, many algorithms are more natural if described in terms of point-to-point calls relative to a virtual-topology naming scheme. Virtual-topology naming might reflect row or column parallelism in matrix operations, for instance [27, 28]. Furthermore, libraries need a full suite of collective operations, including broadcast (one-to-all communication), combine (all-to-all, associative, commutative operator), and collapse (all-to-one communication, associative, commutative operator) [15] (many additional collective operations are specified by MPI [14, Section 4]). In most vendor systems, collective operations are either absent, or must be called over all processes (or processors) of a user's allocation. To build real libraries, it is necessary for the library to stipulate those processes that should participate in a collective operation; those processes that don't want to participate should not have to synchronize artificially because of a programming-model requirement.

2.3 *Zipcode* Features Addressing these Deficiencies

The above arguments can be summarized as follows:

- A safe communication space guarantees that a library can send and receive point-to-point messages without interference from other point-to-point messages generated in the system,
- Collective operations take a static process group as the set of participants, allowing processes that do not participate to continue without an artificial synchronization,
- Abstract names for processes are based on virtual topologies, or at least rank-in-group names, thereby avoiding hardware dependencies, and ideally making application code more intuitive.

In *Zipcode*, the features that implement these important concepts are as follows:

- **Process Groups** define an ordered collection of processes, each with a rank. Process groups define the low-level names for inter-process communication (ranks are used for sending and receiving in certain types of *Zipcode*'s messages). We don't usually reveal the internal representations of process names at the application level (except in some *Zipcode* primitives), though the *Zipcode* implementation currently makes specific assumptions about a 64-bit naming convention based on a "node" and "pid" pair of integers [29].

  Thus, groups define a rank-naming for processes in point-to-point communication relative to the group. In addition, groups define the scope of collective operations. This scope allows us to make strong statements about the non-interference of sequential collective operations in the same context (see [31]). In *Zipcode*, groups are static objects, not shared dynamic objects as they are in PVM [5].

- **Contexts** provide the ability to have separate safe "universes" of message-passing in *Zipcode*. A context is conceptually implemented via a secondary or "hyper" tag, that differentiates messages from one another. Unlike user-manipulated tags, the message passing system manages contexts. In *Zipcode*, a "zipcode" is an integer that implements a single context of communication; we will refer to such an integer as a context_id. Users don't work directly with context_ids, they work with static process groups, and mailers.

- **Mailers** encapsulate contexts, groups, and virtual topologies in an object that provides the appropriate scope for all communication operations in *Zipcode*. Mailers bind process groups and context_ids together to form a safe communication space within the group. Usually, we talk about mailers as communication contexts, in the same spirit that the MPI draft refers to its communicators as contexts. Unlike MPI, *Zipcode* does not currently specify any way to communicate between communication contexts (inter-communication) [14, Section 5]. *Zipcode* (resp, an initial MPI implementation [12]) currently protects point-to-point and collective messages in a single mailer (resp, communicator) by using two context_ids.

  The use of separate communication contexts by distinct libraries (or distinct library invocations) will insulate communication internal to the library execution from external communication (group safety). This allows the invocation of the library even if there are pending communications, and avoids the need to synchronize each entry into and exit from library code.

3 Definitions

To simplify the exposition, we include needed definitions at this point.

Definition 1 (Receipt Selectivity) *Receipt selectivity represents the qualifications that the user can put on the receive call (*i.e.*, how picky the user can be about the message he/she is willing to accept from the system). Typical selectivity includes some notion of where the message came from, and some sort of tagging information (a tag or more complicated ID).*

The *RK* system provides no receipt selectivity at all – the next available message is returned. The Vertex and Thinking Machines CMMD systems both provide the ability to choose based on both source and message tag, where tag is a positive integer (31 bits only). PICL and (until recently) NX permitted selectivity based only on tag matching.

Zipcode provides selectivity that depends on the class of mail being used. In section 4.4.1, we define a number of different classes (L, Y, Z, G1, G2, G3).[1] *Zipcode* emphasizes source selectivity more than tagged source selectivity in its predefined classes. Source selectivity helps to achieve "safe" collective operations based upon point-to-point message passing. In summary, the predefined classes work with the following selectivity:

- Y-class: receipt of a message based on a short integer tag,
- L-class: receipt of a message based on a source {node,pid} and an integer tag,
- Z-class: receipt of a message based on a rank in a process group,
- G1-class: receipt of a message based on a rank in a 1-dimensional mapping of a group,
- G2-class: receipt of a message based on a rank in a 2-dimensional mapping of a group,

[1] *Zipcode* supports multiple notations for message passing, and each is a complete point-to-point message-passing notation, plus a set of collective operations, for those classes that support collective operations.

- G3-class: receipt of a message based on a rank in a 3-dimensional mapping of a group.

Definition 2 (Untagged and Source-Untagged Message System) *An untagged message system uses no tagging to help with receipt selectivity. A source-untagged system does selectivity based on message source without a tag.*

RK is an untagged message system. The *Zipcode* Z, G1, G2, and G3 classes to be defined in section 4.4.1 are all source-untagged.

Definition 3 (Tagged and Source-Tagged Message System) *A tagged message passing system does receipt selectivity exclusively with a single integer tag attached to each message in the system. Systems that also allow selectivity on source are said to be Source-Tagged Systems.*

NX and PICL are tagged message systems. Vertex, Express, MPI, and CMMD are source-tagged message systems. *Zipcode* has a class of messages (Y-class) that is tagged, and another class (L-class) that is source-tagged (see section 4.4.1 for the L- and Y-class definitions).

Definition 4 (Naming Abstraction/Virtual Topologies) *A naming abstraction is an application-relevant means to refer to the members of a process group; virtual topologies implement the naming abstractions. For example, if there are L members of a process group, one could assign a bijection (two indices) to describe the list: $(0 \leq p < P, 0 \leq q < Q)$. (p, q) becomes that abstract name of a process; $P \times Q = L$. In this case, we are viewing a process group as an effective two-dimensional collection of processes. The program does not refer to the hardware-style {node,pid}-pairs, nor to the original rank in the process group, but rather to machine-independent pairs of integers that describe a grid position.*

Express provides a simple notion of two-dimensional grid mapping. *Zipcode*'s G2 class creates exactly the abstraction suggested in the above definition. *Zipcode* also provides other abstractions, as well as the ability to define additional abstractions with new message classes. Other possible abstractions include a ring or binary tree of processes. While we have not implemented these particular abstractions in actual *Zipcode* classes as yet, users are free to add such classes and recompile, thereby augmenting *Zipcode* with new classes of mail.

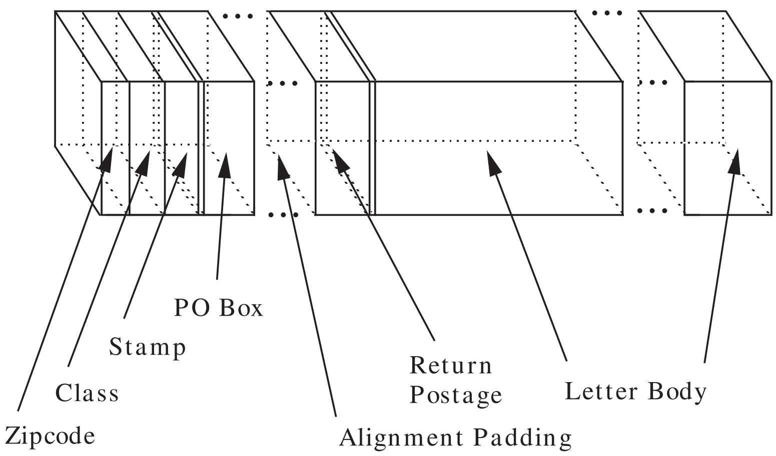

Figure 1: A schematic of the *Zipcode* letter, including the variable-length header (the "envelope") that permits different *Zipcode* mail classes to base receipt selectivity on differing amounts of class-specific information (stored in the "PO Box"). Each field has a specific function, as described in the definition of a letter below.

Definition 5 (Letter) *A letter is the* Zipcode *data structure that carries point-to-point messages. Letters are dynamically allocated as needed by the user, filled in, and deallocated upon "send." A "receive" has the side-effect of providing a letter on completion. This storage-management scheme means that users never have to know how large a letter is to be before actually receiving it*[2]*.*

A letter consists of two main parts: the variable-length header (the "envelope"), and user's data (the "letter body"). Items contained in the envelope allow Zipcode *to implement several important features of the system.*

- **Zipcode** *is a short-integer field containing the context of communication identifier for this letter,*
- **Class** *is a short-integer field describing the kind of message envelope, and specifically, the length and semantic contents of the "PO Box,"*
- **Stamp** *is a short-integer field that indicates the aligned length of the envelope, a quantity needed to find the beginning of the letter body,*
- **PO Box** *is the class-specific, variable-length receipt selectivity information. For instance, in Z-class mail, this is a process source identification,*
- **Alignment Padding** *is the variable-length padding that assures that the letter body begins with a maximally aligned address,*
- **Return Postage** *is a short-integer, with the same value as the "stamp" field, that allows a mapping from the beginning of the letter body back to the beginning of the envelope. This is needed because the user works with pointers to the letter body, whereas* Zipcode *works with pointers to the envelope.*

See Figure 1.

Definition 6 (Message Layering) *The structure of a layered* Zipcode *message is illustrated in Figure 1. Since messages in* Zipcode *are allocated like objects (forming the letter data structure), it is possible for each layer of software interface to add headers that are invisible to the layers above. This technique is used by* Zipcode *to attach class, context, and receipt selectivity information to messages, as well as length (through a lower level in the layering architecture, not shown in Figure 1.).*

4 Programming Model

In this section, we describe how the *Zipcode* implementation addresses such issues as safe communication, virtual topologies, and heterogeneity. We also compare current practice in *Zipcode* to standardization efforts underway in MPI.

We assume a multiple-instruction, multiple-data programming model. Multiple program texts are admissable within the system. Libraries typically operate in a loosely synchronous fashion. However, multiple independent instances of library invocations (as well as process groups that share common processes) are permitted. Support for asynchronous operations is included (for instance, users could define their own libraries for asynchronous collective operations).

4.1 Initialization / Termination

Zipcode was created during a time when the host-node model of parallel computation was more popular than it is currently. Though *Zipcode* does not work solely with the host-node model, most experience with *Zipcode*-baed programs is accomplished with that model. A host program (where appropriate) and each node program must call the appropriate initialization function to start *Zipcode* correctly:

[2] Letter length is supported by the RK semantics upon which *Zipcode* letters are built. Hence, no explicit length field appears in the envelope at present, though this would likely change as *Zipcode* is based on other message-passing systems, such as MPI.

```
int error = Zip_init(void);    /* assume default mode for initialization */
    error = Zip_global_init(void); /* assume a simpler host+SPMD model */
    error = Zip_nohost_init(void); /* no host model. MPMD  */
    error = Zip_nohost_global_init(void); /* no host model, SPMD  */
void zip_exit(void);               /* terminate Zipcode session */
```

The first call is used most generally; those with "global" in their name are used for a host+SPMD model, and omit the "Postmaster General" (context server) process (see Section 4.3, Figure 2). Those calls with "nohost" in their name assume a host-free model. Though supported, we have yet to write substantial programs under these assumptions.

4.2 Process Naming and Process Groups

A static process group (colloquially, an "addressee list" in *Zipcode*) is a basic abstraction that has been found to be useful in a number of message-passing systems. A static process group is used to describe participants in point-to-point operations, and the rank naming of processes in Z-class mailers (and the order of processes in more complicated naming strategies). A static process group has the following properties:

- It is a logical, ordered collection of {`node,pid`} pairs,
- It has a size (number of members),
- It is a purely local object,
- Communication cannot be expressed solely in terms of static process groups,
- A static process group cannot be transmitted between processes by the user.

In all *Zipcode* versions up to now, we have utilized the *Reactive Kernel*'s {`node,pid`}-pairs to describe processes in a pool, whether in a single multicomputer, or in clustered workstations [25]. For a given implementation, {`node,pid`}-pairs will be mapped to hardware names. 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 and virtual topologies. Once message-passing has been set up, most *Zipcode* programs choose to work with logical addressing based on the virtual topologies, hiding the details of the process group structure from the bulk (if not all) of user code.

Originally, users assembled process groups, but they are now to be considered opaque; a standard constructor is provided as follows:

```
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 process group (see Section 4.3, Figure 2).

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 process group. The system may choose to override the node_bias naming never, immediately, or when processes are spawned using the process group. These relaxations retain the opaque nature of the underlying process group, which is important to future generalizations of process naming. On some systems, giving the user the ability to specify the node placement and/or process names could be a help with optimization, but it is mainly a throw-back to an older programming style.

Since user manipulation of process groups is denigrated practice, process groups can be generalized in future *Zipcode* releases without breaking conforming code. In particular, *Zipcode* may provide additional portable ways to construct and modify process groups (similar to the MPI functionality [14, Section 5]), and particular environments could provide non-portable calls to provide additional process groups with appropriate opaque structure. Within the *Zipcode* system itself, there remains the need for non-enumerative representation of process groups, and more general process naming (*e.g.*, PVM task ids).

For completeness, *Zipcode* provides the following process management support, for which there is no analog planned in MPI; the PVM systems also provides this capability, but with different semantics (*i.e.*, 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);
```

where

- prog_name is the ASCII name of the program to spawn, local to the spawner's file system,
- addressees is the process group upon which to spawn the program,

and where result is non-zero on failure. Most ports require that this spawning function be effected in the host process, though this restriction is less likely in a distributed setting.

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, COHORT_FIRST_NODE, try_pid, TRUE);
result = zip_spawn("./testprog", addressees);
```

A comparable zip_kill() call is also defined. Although this function is defined for all implementations, in some environments this function may have no effect.

```
int result = zip_kill(ZIP_ADDRESSEES *addressees);
```

With the inclusion of these functions, *Zipcode* specifies an entire programming environment; codes need not explicitly reference vendor process management.

4.3 Contexts of Communciation and Mailers

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 previously in several papers on *Zipcode* [29, 31, 32, 33]. 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 a process group, the members of which are the participants in the communication,
- A context of communication has one or more system-defined labelings of message passing for its process group in the system, each of which is non-interfering,
- It provides a logical partitioning of receipt selectivity into user-defined, and system-managed components.
- If used correctly, contexts guarantee that messages will not be misdirected (group safety).

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 context_ids,
- A context_id is a globally managed quantity that may be reused by disjoint groups,
- No wildcarding of context_ids is permitted.

A mailer implements contexts of communication in *Zipcode*, with the following features:

- A mailer contains a process group,
- A mailer contains safe communication space for the process group's point-to-point and collective message passing (realized with context_id's),
- A mailer contains a set of methods[3] implementing point-to-point message passing (particularly appropriate to that process group's hardware),
- depending on the topology, a mailer may reference specific "child mailers," each recursively specifying further subsets of the parent group.

4.4 Mail Classes

All communication operations take a mailer as an argument to specify group and context properties of the communication operations, as well as to specify the methods implementing these operations (see Section 4.4.6). Virtual topology information and multiple contexts of communication are combined when creating "mailers" for the grid classes of mail.

The creation of a context synchronizes the participants in the participating group, while promulgating the process group and issuing valid context_ids (see Figure 2). Only the "Postmaster" is required to know the process group initially (it is always the rank-zero process of that group). All processes named in that group need to invoke the collective operation for mailer creation; the non-Postmasters receive the process group as part of the synchronization procedure. The context server process (Postmaster General) provides the needed context_ids and promulgates them with the process group information to all participants. A token issued by the Postmaster General is held by the Postmaster to ensure that the process completes without the chance that mailers fail because of race conditions on overlapped groups with distinct Postmasters. (A related, server-free model is implemented in MPI.)

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

[3]Methods are pointers to functions, plus extra state data.

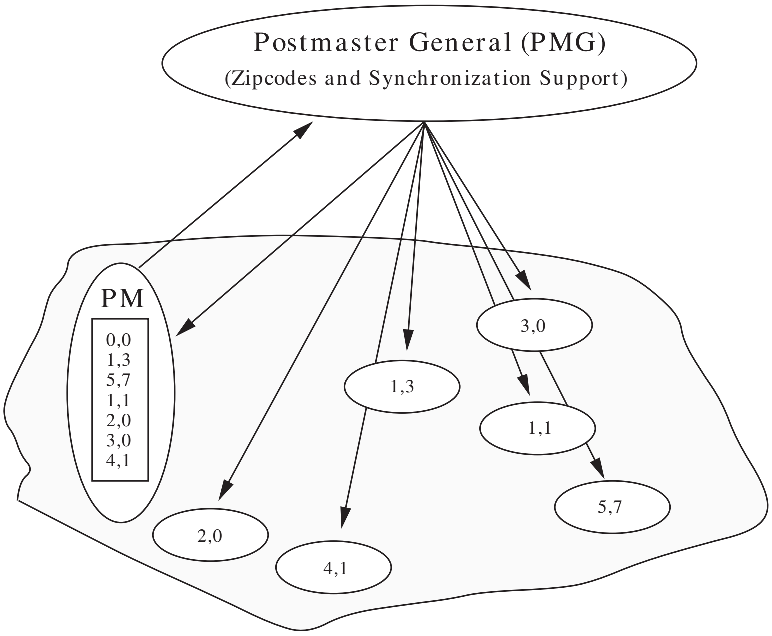

Figure 2: Schematic representation of information flow during a mailer creation. First, the "Postmaster" (PM) process transmits its process group to the "Postmaster General" (PMG). Then, the PMG broadcasts the process group to all members of that group; in addition, needed context_ids are transmitted simultaneously. The pairwise-ordering property of messages helps guarantee that this procedure completes reliably.

```
int P, Q, R;
ZIP_ADDRESSEES *addressees;

ZIP_MAILER *mailer = g3_grid_open(&P, &Q, &R, addressees);
```

The Postmaster for the mailer-open calls the `g3_grid_open()` procedure with a valid process group and valid values for the grid shape. All other participants call with undefined values for the grid shape and NULL for the process group. A seldom-used variant exists that permits the context_ids to be specified by each participant. In that case, if all participants know their process group and context_ids, then mailer creation is communication-free (and otherwise erroneous). However, this latter feature allows for efficient creation of hierarchical mailers.

A Postmaster for a logical grid need not be part of the logical grid (*e.g.*, creation by the host or another leader node, but participation by nodes excluding this leader). Whenever there are $P \times Q \times R + 1$ processes in the process group, the Postmaster is excluded from the actual grid, and children thereof, but gains the ability to communicate with the parent grid. In effect, it gets access to the context of communication without being a member of that context of communication. We support this non-member-access in only this timid way at present, but recognize that such non-member access is needed for more general "server" scenarios, which we would like to support better in the future (this is related to "inter-communication" in MPI, [14, Section 5]).

4.4.1 Predefined Mail Classes

The following are the predefined classes:

Y-Class mail is used mainly for *Zipcode* internal mechanisms. The receipt selectivity information is a single short integer tag. No collective operations are defined.

Z-Class mail is a general purpose class. Process names are abstracted to a single integer rank (based on rank in the underlying group); receipt-selectivity is based on that source name. This is a class of mail that is also used mainly by the system to implement higher-level functions, but is also usable by applications.

Originally, each class of mail that was sufficiently expressive could implement its own collective communication, working with a second context_id (to separate such messages from point-to-point operations). However, this was regarded as unwieldy. It also incurred additional overhead for those classes of mail with elaborate receipt selectivity. Now, regardless of the virtual topology, all collective communications for a mailer are implemented using an isomorphic Z-class mailer, with its own context_ids. This simplifies implementation, reduces the amount of repetitive code, and, generally, enhances the opportunities for runtime optimizations on a specific system.

L-Class mail provides for receipt selectivity based on message source in unabstracted {`node,pid`}-notation, and on a long-integer tag. It can be used to support emulation of tagged message notations such as Intel's NX or PICL [18].

We have been able to classify a number of message-passing systems in [31], though specific differences in sending and receiving strategies exist between common tagged-message-passing systems. L-class calls can be used to generate wrappers for all the major tagged-message-passing systems. We have implemented a *Zipcode*-based emulation for the Livermore Message Passing System (LMPS) [39].

For each context a user declares, he/she is guaranteed that the L-class messages will not be mixed up, so that if vendor-style calls are used in different libraries, then these will not interfere with other parts of a program. This allows several existing tag-oriented subroutines or programs to be brought together and face-lifted easily to work together, without changing tags or seeing when/where the message passing resources might conflict. In short, this provides a general means to ensure tagged-message registry as contemplated in [20].

Grid Classes of mail are supported in three forms, for one-, two-, and three-dimensional virtual topologies; higher dimensions can easily be added. G1 class is a 1D-grid-abstraction class, similar to Z-Class mail. For brevity, we omit the calls supported by this class, which are simplified notations of the G2 and G3 classes. As one might expect, G2 (resp, G3) class mail is a 2D-grid-abstraction (resp, 3D-grid-abstraction) class. A $P \times Q$ grid naming abstraction is attached to the mailer for G2, and a $P \times Q \times R$ grid naming abstraction is attached to the mailer for G3. For G2, each process is specified by a (p, q) pair; for G3, this is replaced by the analogous (p, q, r) triplet. When a G3 mailer is defined, three plane children (PQ-, PR-, QR-plane mailers) are automatically defined as G2-grid mailers. Recursively, through a further subgridding process, row and column (G1) mailers are defined in each process as the appropriate subsets for each G2 grid. Here a single mailer is in fact a pointer to a family of contexts defined through a single "open" of a process grid topology.

Shorthands provide access to the PQ-plane, QR-plane, and PR-plane children of G3-class mailers, to which G2 grid operations may be applied, as above.

```
ZIP_MAILER *g3_mailer;  /* 3D grid mailer */
ZIP_MAILER *plane_mailer; /* 2D grid mailer */
plane_mailer = g3_PQ_plane(g3_mailer);
plane_mailer = g3_QR_plane(g3_mailer);
plane_mailer = g3_PR_plane(g3_mailer);
```

Furthermore, shorthands for row/column access of G2 mailers are supported, to which G1 operations may be applied:

```
ZIP_MAILER *g2_mailer; /* 2D grid mailer */
ZIP_MAILER *row_mailer /* 1D */, *col_mailer; /* 1D */
row_mailer = g2_row(g2_mailer);
col_mailer = g2_col(g2_mailer);
```

It is often necessary to determine the grid shape as well as the current process' location on the grid when using logical grids. Often this information is housed only in the mailer (though some applications may choose to duplicate this information). The following C macros provide simple access to these quantities:

```
int p, q, r, P, Q, R; ZIP_MAILER *mailer;

/* set variables specified to grid shape: */
void g1_P(ZIP_MAILER *mailer, int P);
void g2_PQ(ZIP_MAILER *mailer, int P, Q);
void g3_PQR(ZIP_MAILER *mailer, int P, Q, R);

/* set variables to current processes' grid position: */
void g1_p(ZIP_MAILER *mailer, int p);
void g2_pq(ZIP_MAILER *mailer, int p, int q);
void g3_pqr(ZIP_MAILER *mailer, int p, q, r);
```

For historical purposes, we note that G2 was the main class used originally by *Zipcode*-based libraries, but both G1 and G3 are fully supported at present. G2 received the most extensive use because of the natural application to single-instance parallel linear algebra and related computations; multiple instance linear algebra problems and problems defined on three-dimensional spatial topologies find use for G3 abstractions. (This technology is somewhat related to, but predates the Basic Linear Algebra Communication Subprogram (BLACS) approach to providing grid-oriented communication for parallel libraries [1, 11].)

4.4.2 Letters

The *Zipcode* "letter" contains a user's message data, plus opaque (hidden) descriptive information in its "envelope"; the variable-length envelope includes its zipcode (context_id), PO Box (receipt selectivity information), and other needed structural data (as introduced in section 3). The postal analogy in *Zipcode* carries quite far because a process creates and mails a letter, first by grabbing and filling out a blank message, then by addressing its envelope, and finally, by posting the entire object. Letters may be created (freed) by using the class-independent zip_malloc (zip_free) function or by using class-specific letter management functions.

```
char *letter = zip_malloc (ZIP_MAILER *mailer, int length);
      letter = l_new_letter (ZIP_MAILER *mailer, int length);
      letter = z_new_letter (ZIP_MAILER *mailer, int length);
      letter = g1_new_letter (ZIP_MAILER *mailer, int length);
      letter = g2_new_letter (ZIP_MAILER *mailer, int length);
      letter = g3_new_letter (ZIP_MAILER *mailer, int length);

void zip_free (ZIP_MAILER *mailer, char *letter);
     l_free_letter (ZIP_MAILER *mailer, char *letter);
     z_free_letter (ZIP_MAILER *mailer, char *letter);
     g1_free_letter (ZIP_MAILER *mailer, char *letter);
     g2_free_letter (ZIP_MAILER *mailer, char *letter);
     g3_free_letter (ZIP_MAILER *mailer, char *letter);
```

4.4.3 Point-to-Point Communication

Class-specific send and receive primitives are provided for the each predefined class of mail. Receive primitives for the G1-, G2-, and G3-Classes are as follows:

```
char *letter = g1_recv(ZIP_MAILER *mailer, int p); /* unblocked: */
      letter = g1_recvb(ZIP_MAILER *mailer, int p);  /* blocked: */
      letter = g2_recv(ZIP_MAILER *mailer, int p, q); /* unblocked: */
      letter = g2_recvb(ZIP_MAILER *mailer, int p, q);  /* blocked: */
      letter = g3_recv(ZIP_MAILER *mailer, int p, q, r); /* unblocked: */
      letter = g3_recvb(ZIP_MAILER *mailer, int p, q, r);  /* blocked: */
```

Send primitives are as follows:

```
void g1_send(ZIP_MAILER *mailer, char *letter, int p);
void g2_send(ZIP_MAILER *mailer, char *letter, int p, q);
void g3_send(ZIP_MAILER *mailer, char *letter, int p, q, r);
```

Sends are presumed asynchronous in *Zipcode*.

4.4.4 Collective Communication

Zipcode provides several types of collective communication operations that are performed over the members of a mailer's group. The basic types of collective communication calls are the combine, broadcast (fanout), collapse (fanin), parallel prefix, and sync (barrier synchronization). Broadcasts share data of arbitrary length, assuming all participants know the source. Collapses combine information assuming all participants know the destination.

The combine, fanin, fanout_exec (a PICL-like variant of fanout), and parallel prefix collective operations use an associative-commutative "method" to perform an operation on the given data. The application programmer must provide an associative-commutative function, which is then encapsulated in a `Method` structure using a simple procedure (all *Zipcode* function pointers are so encapsulated, for symmetry). We illustrate several collective G2 calls (and a G1 and G3 call) to exemplify all the classes, which differ only in the description of the origin or destination process:

```
int error = g{1,2,3}_combine(ZIP_MAILER *mailer, /* 1D,2D, or 3D grid mailer */
                       void *buffer,             /* where result is accumulated */
                       Method *comb_method,      /* operator for combine */
                       int size,                 /* size of buffer items in bytes */
                       int items);               /* number of buffer items */
    error = g2_fanout(ZIP_MAILER *g2_mailer,
                      void **data,               /* data/result */
                      int *length,               /* data length */
                      int orig_p, int orig_q);   /* grid origin of data */
    error = g2_fanin(ZIP_MAILER *g2_mailer,
                     int dest_p, int dest_q,     /* destination on grid */
                     void *buffer, Method *comb_method, int size, int nitems);
```

Shorthands provide direct access to row and column children mailers:

```
g2_row_combine(g2_mailer, buffer, comb_method, size, items);
g2_col_combine(g2_mailer, buffer, comb_method, size, items);

g2_row_fanout(g2_mailer, &data, &length, orig_q);
g2_col_fanout(g2_mailer, &data, &length, orig_p);
```

and

```
g2_row_fanin(g2_mailer, dest_q, buffer, comb_method, size, items);
g2_col_fanin(g2_mailer, dest_p, buffer, comb_method, size, items);

g3_fanin(PQ_plane(g3_mailer), dest_p, dest_q, buffer, comb_method, size, items);
```

Since planes of G3-mailers are implemented via G2-mailers, plane macros map G3-mailers into specific G2-mailers. Similarly, the row/column macros above map to G1-grid calls, since rows and columns of G2-mailers are realized via G1-mailers.

4.4.5 Subgrids

Once a grid mailer has been established, it is possible to derive subgrid mailers by a cooperative call between all the participants in the original `g{1,2,3}_grid_open()`. In normal applications, this will result in a set of additional mailers in the Postmaster process, and one additional grid mailer (of the same dimensionality as its parent mailer) in each non-Postmaster process. This call allows subgrids to be aligned to the original grid in reasonably general ways, but requires a basic cartesian subgridding, in that each subgrid defined must be a rectangular collection of processes.

The Postmaster of the original mailer (often the host process, in current practice), initiates the subgrid open request as follows, for the G3 case:

```
    /* array of pointer to subgrid mailers: */
    ZIP_MAILER **subgrid_mailers =
        g3_subgrid_open(ZIP_MAILER *mailer, /* mailer already opened */
              /* for each (p,q,r) on original grid, marks its subgrid: */
              int (*select_fn)(int p, q, r, void *extra),
              void *select_extra;           /* extra data needed by select_fn() */
              int *nsubgrids);              /* the number of subgrids created   */
```

while each non-Postmaster process in the original `g3_grid_open()` does a second, standard `g3_grid_open()`:

```
    ZIP_MAILER *mailer = g3_grid_open(&P, &Q, &R, NULL);
```

Each subgrid so created gets its own unique contexts of communication.

4.4.6 Method Caching in Mailers

Zipcode currently provides a "method caching" mechanism that allows one to associate new specific methods for pre-defined collective and point-to-point operations with mailers, allowing mailers to be customized at runtime to take advantage of specific features inherent in a process group (eg, homogeneity, power of two, shared memory message passing, etc). This feature is less general than MPI's caching mechanism that allows arbitrary attributes to be attached to communicators, with full copy- and delete-callback facilities [14, Section 5]. We illustrate the usefulness of "method caching" in Figures 5, 6.

4.5 Invoices

In order to facilitate ease-of-use and to prepare for portability to heterogeneous parallel computers, *Zipcode* has been extended to provide a mechanism to pack and unpack buffers and letters. Buffers are unstructured arrays of data provided by the user; they are applicable with buffer-oriented collective operations. Letters are the unstructured arrays of data already mentioned previously in Section 4.4.2. Letters are tied to specific contexts and are dynamically allocated and freed by *Zipcode*. Further discussion on invoices and their interactions with letters and buffers is available in [34].

Pack (gather) and unpack (scatter) are implemented with the use of `Zip_Invoices`. The analogy is taken from invoices or packing slips used to specify the contents of a postal package. An invoice informs *Zipcode* what variables are to be associated with a communication operation or communication buffer. This invoice is subsequently used when `zip_pack()` (`zip_unpack()`) is called to copy items from the variables specified into (out of) the communication buffer space to be sent (received); this implements gather-send- and receive-scatter-style semantics. In a heterogeneous environment, pack/unpacking will allow data conversions to take place without user intervention. Users that code strictly with `zip_pack()`/`zip_unpack()` will have codes that are guaranteed to work in heterogeneous implementations of *Zipcode*. Performance of the existing layered implementation is described in [31, 34].

The `zip_new_invoice()` call creates new invoices:

```
int  = zip_new_invoice(Zip_Invoice **inv, const char *format, va_list ap);
```

`zip_new_invoice()` creates an invoice (`inv`), while taking a variable number of arguments, starting with a format string (`format`) similar to the commonly used `printf()` strings. The format string contains one or more conversion specifications. A conversion specification is introduced by a percent sign ('%') and is followed by

- A positive integer indicating the number of items to convert, or a '*' or '&' indicating argument-list specification of an integer expression or address (see below). If no integer is specified the default is one item.
- An optional stride factor indicated by a '.' followed by a positive integer indicating the stride. Optionally a '*' or '&' may be specified, signifying argument-list specification of an integer expression or address (see below). If no stride is specified the default is one.
- An optional '-' character indicating that the indicated space is to be reserved but not packed (ignore-space option).
- A character specifying an internal type or a string indicating a user type.

For both the number of items to convert or stride, '*' or '&' can replace the hard-coded integer. If '*' is used, then the next argument in the argument list is used as an integer expression specifying the size of the conversion (or stride). Both the number of items to convert or stride factor can be indirected by using '&' instead of an integer. The '&' indicates that a pointer to an integer should be stored, which will address the size of the invoice item (or stride) when it is packed. When '&' is used, the size is not evaluated immediately, but is deferred until the actual packing of the data occurs. '&'-indirection consequently allows variable-size invoices to be constructed at runtime; we call this feature *deferred sizing*. '*' allows the size of an invoice item (or stride) to be specified at runtime. Note that one must be cautious of the scope of C variables when using '&'. For example, it is erroneous to create an invoice in a subroutine that has a local variable as a stride factor and then attempt to pass this invoice out and use it elsewhere, since the stride factor points at a variable that no longer is in scope. Unpredictable, bad things will happen if this is attempted.

The simple data types that are supported are as follows: 'c' for character, 's' for short integer, 'i' for integer, 'l' for long integer, 'f' for single-precision floating point, 'd' for double-precision floating point. For each conversion specification, a pointer to an array of that type must be passed as an argument. User-defined types may be added to the system to ease the packing of complicated data structures. An extra field (for passing whatever the user wants) may be passed to the conversion routines by adding '(*)' to the end of the user-type name. The '-' character can be used to skip space so that one can selectively push/pull things out of a letter. This allows for unpacking part of a letter and then unpacking the rest based on the part unpacked.

The following code would pack variable `i` followed by elements $0, 2, 4, ..18$ of the `double_array`.

```
/* Example 1 */
ZIP_MAILER *mailer;
char *letter;
...

Zip_Invoice *invoice;
int i = 20;
double double_array[20];

zip_new_invoice(&invoice, "%i%10.2d", &i, double_array);
...

/* use the invoice (see below) */
letter = zip_malloc(mailer, zip_sizeof_invoice(mlr, invoice));
length = zip_pack(mailer, invoice, ZIP_LETTER, &letter, ZIP_IGNORE);

if(length == -1)      /* an error occurred */
   ...
```

The second example is a variant of the first. The first pack call is the same, while the second packs the first five elements of the double_array.

```
/* Example 2 */
int len = 10, stride = 2;
...
zip_new_invoice(&invoice, "%i%&.&d", &i, &len, &stride, double_array);

/* use the invoice */
letter = zip_malloc(mailer, zip_sizeof_invoice(mailer, invoice));
length = zip_pack(mailer, invoice, ZIP_LETTER, &letter, ZIP_IGNORE);
...
len = 5;  /* set the length and stride for this use of the invoice */
stride = 1;

/* use the invoice */
letter = zip_malloc(mailer, zip_sizeof_invoice(mailer, invoice));
length = zip_pack(mailer, invoice, ZIP_LETTER, &letter, ZIP_IGNORE);
```

If a user-defined type matrix has been added to the system to pack matrix structures, then the following example shows how matrix-type data can be used in an invoice declaration. See also below on how to add a user-defined type.

```
/* Example 3 */
struct matrix M;   /* some user-defined type */
int i;
Extra extra; /* contains some special info on packing a 'matrix'; */
             /* often this will not be needed, but this feature   */
             /* is provided for flexibility                       */

zip_new_invoice(&invoice, "%i%matrix(*)%20d", &i, &M, &extra, double_array);
```

At times it might be useful to know the size (in bytes) that is needed to hold the variables specified by an invoice. zip_sizeof_invoice returns the size (in bytes) that the invoice will occupy when packed. We have already used this facility in several examples above.

```
int zip_sizeof_invoice(ZIP_MAILER *mailer, Zip_Invoice *inv);
```

To delete an existing invoice use zip_free_invoice():

```
void zip_free_invoice(Zip_Invoice **inv);
```

This will free up the specified invoice and set *inv = NULL to help flag accidental access.

User-defined types for pack and unpack routines are defined using a registry mechanism provided by *Zipcode*.

```
int zip_register_invoice_type(char *name, Method *in, Method *out,
                              Method *len, Method *align);
```

In the above, name is the user-defined name for the auxiliary type. User-defined names follow the ANSI standard for C identifiers. They begin with a non-digit (characters 'A' through 'Z', 'a' through 'z', and the underscore '_'), followed by one or more non-digits or digits. User-defined-type names currently have global scope with potential for name conflicts. User-defined types cannot be the same as one of the built-in types specified above. The in, out, len and align are the Methods used to pack/unpack the user-defined type [31, 34].

The "out" (resp, "in") method performs any necessary data conversions when messages are sent (resp, received). The total size necessary to pack a user-defined type is computed by the "len" method. The "align" method returns the number of bytes that must be added to properly align a user-defined type.

Finally, the following call is used to remove a user-defined type from the system:

```
int zip_unregister_invoice_type(char *name);
```

zip_unregister_invoice_type deletes the entry for the named type, which cannot be used after this call has been made.

4.5.1 Packing and Unpacking

Packing is done when one wishes to copy the variables into the communications buffer space prior to transmission; to access the contents of a packed buffer, one must unpack it first.

```
int zip_pack(ZIP_MAILER *mailer, Zip_Invoice *inv, int buffer_type,
          char **ptr, int len);
```

This command packs the invoice. buffer_type is either 'ZIP_BUFFER' or 'ZIP_LETTER', indicating whether we are packing into a buffer (say for a combine or fanout) or a letter (for sends/receives).

If one is packing a buffer and has preallocated the buffer space, then len must be set to the size of this allocated buffer space. If the invoice is too large to fit in this buffer space an error occurs. By specifying *ptr = NULL and len = ZIP_IGNORE, the pack routine will allocate the space for the buffer based on the size of the invoice to be packed. Alternatively, if a pre-allocated letter is being packed, then pack will fill in the letter by using the invoice. If the letter provided is not large enough then an error will occur. If no pre-allocated letter is available, the pack routine can create one automatically, provided *ptr = NULL. Note that len is ignored when letters are involved, as the size of letters can be determined with Zip_length(); len should always be ZIP_IGNORE when packing letters. For either case, zip_pack() returns the number of bytes that the data from the invoice occupies in the communication space (letter or buffer).

To unpack a letter use

```
int zip_unpack(ZIP_MAILER *mailer, Zip_Invoice *inv, int buffer_type, char *ptr);
```

As in `zip_pack()`, `inv` is the invoice to unpack. The `buffer_type` parameter indicates the type of communication space being used; that is, whether we are unpacking a letter (`buffer_type = ZIP_LETTER`) or a buffer (`buffer_type = ZIP_BUFFER`). The parameter `ptr` is a pointer to the communication space. Unlike `zip_pack()`, we pass a pointer to the communication space to `zip_unpack()`, not a pointer to a pointer. The communication space must be freed by the caller after it is unpacked.

4.5.2 The Packed-Message Functions

As may be apparent, many packs are followed almost immediately by sends while corresponding receives are followed closely by unpacks. Not only is this somewhat notationally tedious, but it also limits the runtime optimizations that can be contemplated by future versions of *Zipcode*. To create a more flexible system with future high performance, *Zipcode* provides the capability to do both the pack and communications in a single call. For instance,

```
g3_pack_send(ZIP_MAILER *g3_mailer, int p, q, r, Zip_Invoice *invoice);
```

takes care of creating the letter, packing the invoice and sending it to the grid location specified by `{p,q,r}`. Whenever possible, use `pack_send`-style routines, as they will generally be more runtime optimizable than pack calls followed by send calls.

Packed collective operations are also provided. For those collective operations which require methods, *Zipcode* provides built-in methods that work over the elements of an invoice. There are currently twelve built-in methods available that perform operations such as addition (ZIP_ADD), logical 'and' (ZIP_LAND), and minimums and maximums (ZIP_MIN and ZIP_MAX). *Zipcode* also provides macros that create new user-defined methods. Here is the specific syntax of the grid pack_combines:

```
int g{1,2,3}_pack_combine(ZIP_MAILER *mailer, Zip_Invoice *invoice,
                Method *comb_method);
```

4.5.3 Lessons Learned

Though we defined invoices to improve the software-engineering aspects of message-passing programming, we have come to understand that abstractions like invoices are helpful for runtime optimization of message-passing. Basically (without overselling the concept), the user indicates "what" is to be communicated rather than "how," so it is possible at runtime for the system to make choices that reduce the number of copies of data, and that possibly use special gather/scatter hardware. For instance, for collective operations, we could use hardware such as the CM-5's control network in order to implement certain collective operations on small buffers. By contrast, on the Cray T3D, we could use remote memory access primitives for short messages and change to the "Block Transfer Engine" for longer messages, where the T3D's gather/scatter hardware's startup overhead is proportionally less significant.

To summarize, we have experience using invoices in a practical application, where we found that they reduced the time needed to formulate a message-passing application [34]. In [34], we offer evidence that a layered invoice implementation is not without overhead. Higher net efficiency requires a non-layered approach, and should make use of efficient hardware where available, accessed through a portable, efficient device mechanism if possible (see [19]). We have found that invoices are sufficiently flexible to tackle complicated tasks, but that the use of the string-based syntax is not particularly convenient for extremely large invoices.

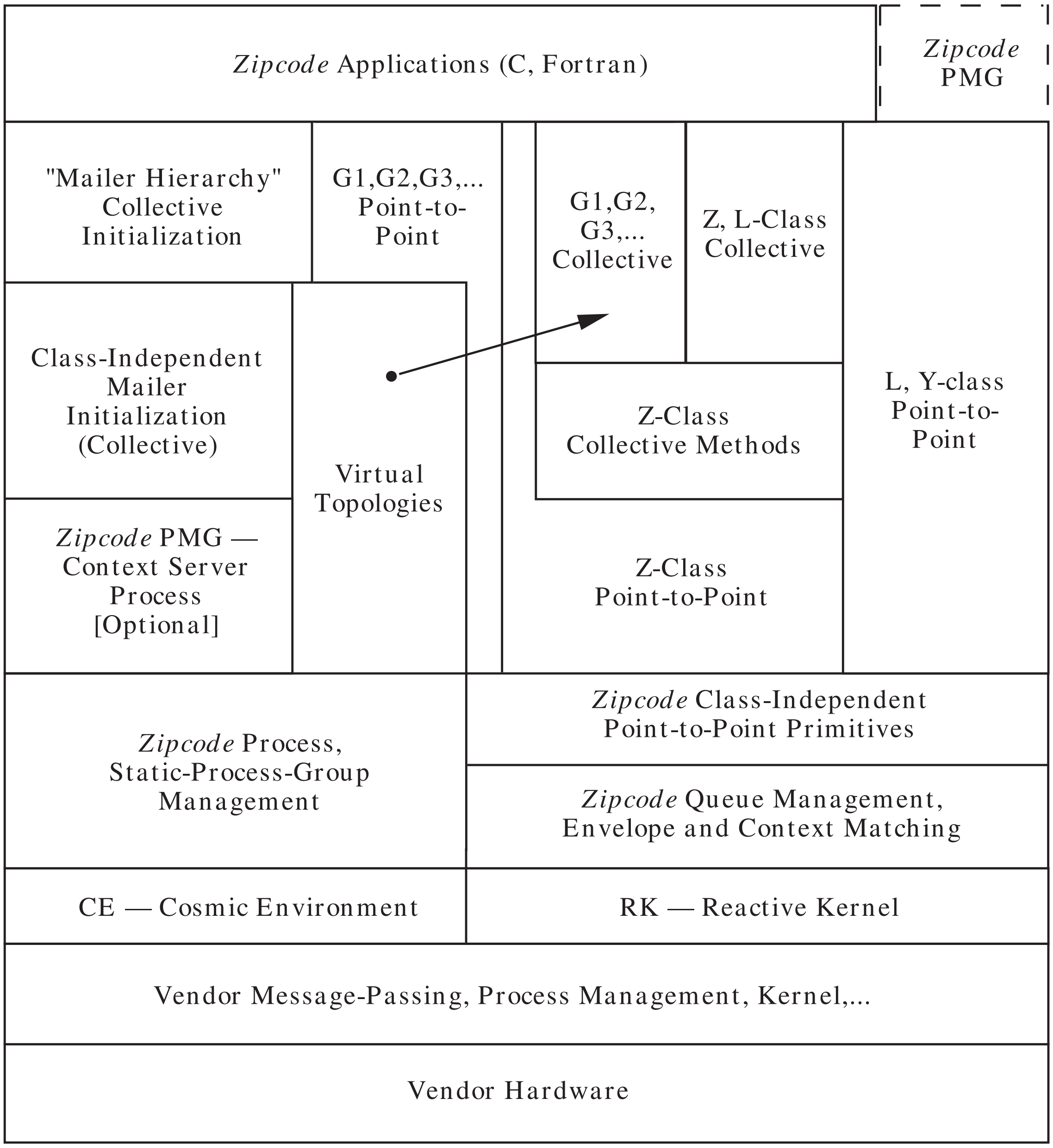

Figure 3: We depict the protocol stack for the current *Zipcode* implementation. Towards the left, the hierarchical layering of process management is shown. Towards the right, the hierarchical layering of message-passing services is shown. Currently, *Zipcode* uses the Cosmic Environment (CE) primitives as its interface to process management, and the Reactive Kernel (RK) primitives as its interface to message passing. To this date, ports of the relatively modest number of CE/RK primitives have been accomplished on a wide variety of multicomputers and shared-memory multiprocessors, as well as homogeneous networks of workstations. (The arrow indicates further layering dependence that we could not otherwise depict in two dimensions.)

5 Portability

5.1 Current Porting Strategy

Zipcode has to now relied on the basic process management (spawn/kill) and messaging services (x-primitives) of the *Reactive Kernel / Cosmic Environment*, or, more usually, our own emulations thereof [23, 24, 25]. 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 [4, 5]; integration of *Zipcode* with ELROS messaging capabilities is also being undertaken [8]; a direct TCP/IP port is also contemplated, 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 starting point, and is reasonable for multicomputers. 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 (or restricted), but acceptable portability is still maintained.

5.2 Future Scope of Portability

Zipcode has fully surpassed its original *Reactive Kernel / Cosmic Environment* platform [23, 24, 25]; it is now planned that *Zipcode* implementations will be based on one or more of the following in a given implementation:

- Hardware-based shared memory,
- Active-message strategies (*cf*, [38]),
- Lightweight layering on MPI implementations,
- Control-Network operations definable on process groups (subsets of processes),
- TCP/IP and/or UDP,
- High-speed network protocols (*e.g.*, ATM, Fiberchannel, FDDI).

Heterogeneous translation can be by one of several translation mechanisms, for instance: XDR [35], ELROS [7, 8], or other strategies (that appropriately balance the work of the sender and recipient in the translation process as a function of their computational speed for such translations). Because *invoice*s are persistent objects, the runtime cost of discovering and reorganizing data transmission to enhance vectorization is possibly feasible, as such costs can be amortized over many uses. This runtime optimization can be done transparently to the user, and differently in each mailer, according to the nature of the processes belonging to each mailer.

Importantly, when a code is moved to a system that does not have special features (*e.g.*, a purely message-passing system), the user code's calls to *Zipcode* will compile down to pure message-passing, whereas the calls compile down to faster schemes within special parts of non-uniform memory access hierarchies. Originally, the *CE/RK* primitives were the cheapest available primitives for system-level message-passing, and hence the most attractive to build higher-level services like *Zipcode*. Today, vendor operating systems are likely to provide additional services in the other categories mentioned above which, if used directly in applications, would prove unportable, unmanageable, or too low-level (like direct use of *CE/RK* primitives). If a user needs to optimize a code for a specific system, he or she works in terms of process groups, and contexts, to get desirable mappings from which *Zipcode* can effect runtime optimizations. These ideas are depicted in Figures 5, and 6.

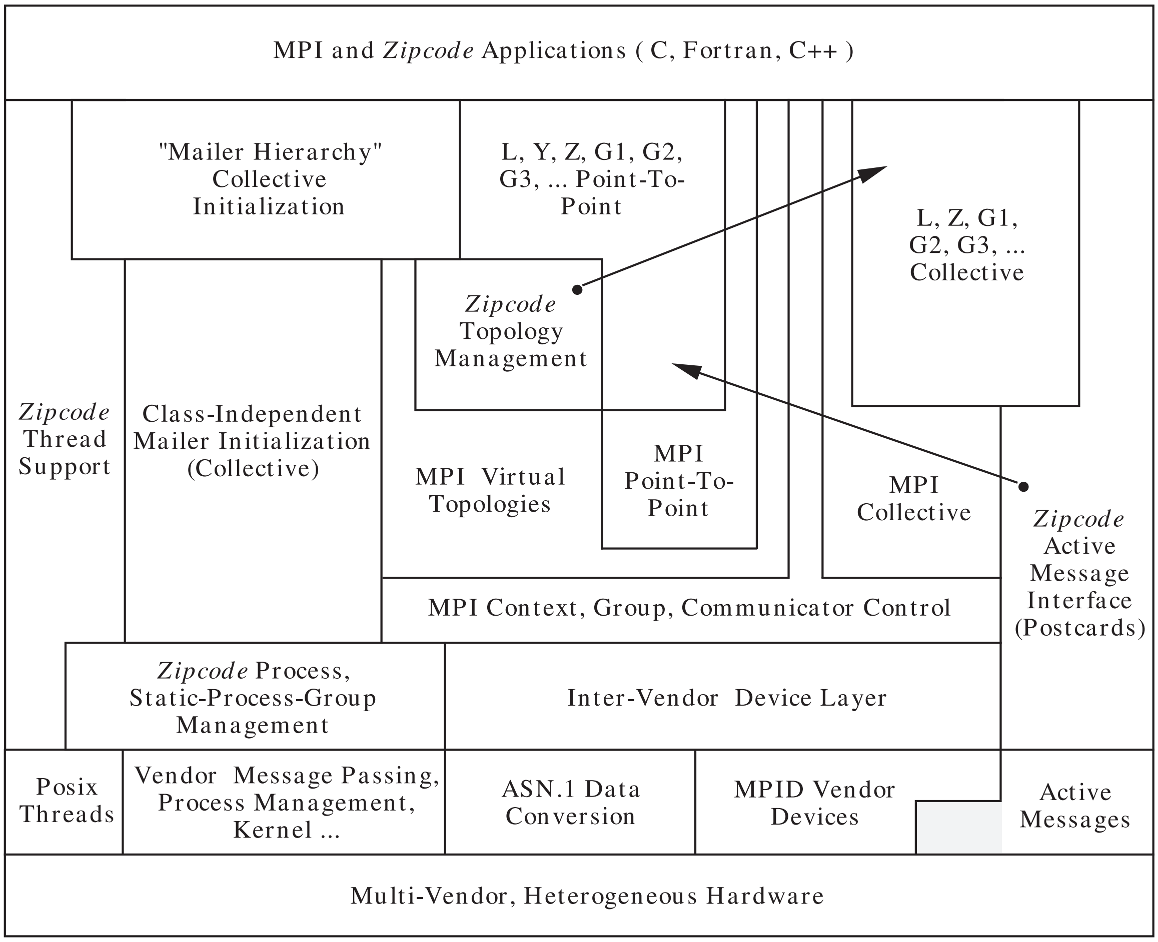

Figure 4: The advent of MPI, Active Messages, and the Posix standardization of threads [36] opens new opportunities for further *Zipcode* research. In particular, studying means to interface multiple MPI systems with threads and active messages alone poses worthwhile technical challenges. In the "new world order," MPI subsumes much of the layering of the original *Zipcode* protocol stack for message passing, leaving a "thin" interface to the user, and revealing higher performance than *Zipcode* could achieve previously. *Zipcode* will continue to provide the added value of virtual topology support supplementary to MPI, as well as process management (including threads) and active messages. This protocol stack would give *Zipcode* applications access both to the medium-latency, high-bandwidth MPI protocols, as well as the low-latency, low-bandwidth Active Message protocols. Finally, access to threads will help make *Zipcode*-based applications latency tolerant. (Arrows indicate further layering dependence that we could not otherwise depict in two dimensions.)

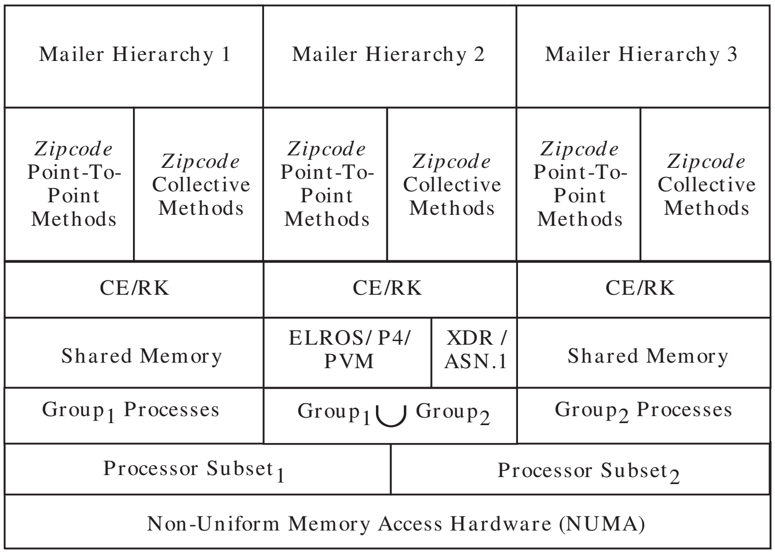

Figure 5: In a non-uniform memory access (NUMA) hardware environment, *Zipcode*'s mailers provide a useful abstraction. The figure depicts two groups of processes, each working within a special subset of a distributed memory hierarchy. The *Zipcode* mailer for each process group would utilize shared-memory-based primitives appropriate to that portion of the hardware. The *Zipcode* mailer for the union group would have to use (potentially slower) message-passing mechanisms. The software technology to achieve this type of runtime optimization is already in place in *Zipcode*, but we have not yet demonstrated a running system of this complexity.

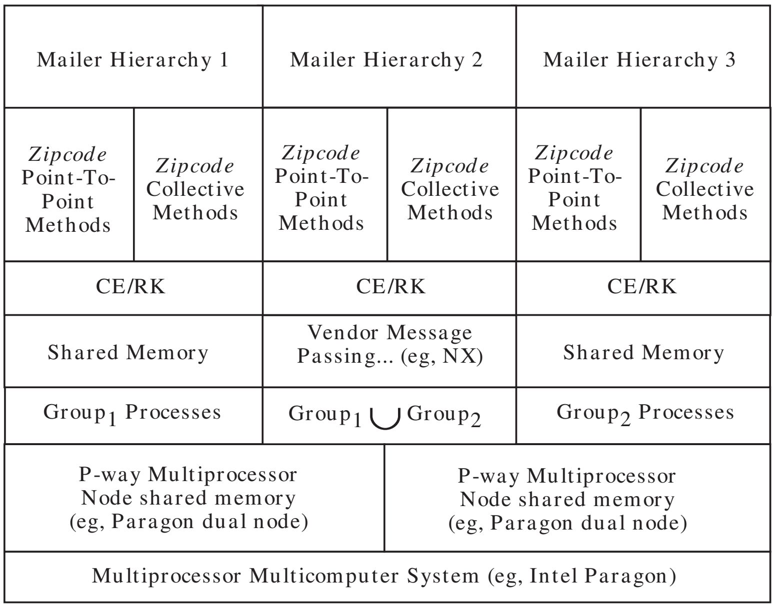

Figure 6: A non-uniform memory access environment (NUMA) can also occur within a single multicomputer. The figure depicts a special case of Figure 5 for the Intel Paragon system, assuming the node pairs are programmed symmetrically. For groups of processes (*e.g.*, pairs of processes) executing on a multiprocessor node, shared-memory-based primitives would be used to implement the *Zipcode* protocol stack. Mailers defined over those processes would avoid full-weight message passing of the mesh-connected multicomputer. Mailers with process groups including multiple multiprocessor nodes would use a full-weight message-passing protocol. We expect to test this technology in the coming months.

6 Future Work

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 the 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 MPI, lighter-weight context support will become a reality, and *Zipcode*-based applications will increase performance either by moving directly to MPI primitives, or by our planned light-weight port of *Zipcode* that will run on top of MPI. We view this evolution as highly satisfactory, as it has provided good portability and performance enhancement to our *Multicomputer Toolbox* libraries. Therefore, no *Zipcode*-based application or library will suffer in face of the standardization. Instead, they will benefit almost immediately by it, and be even more portable, as multiple vendors will eventually implement MPI, whereas we were obliged to undertake each *Zipcode* port as new hardware became available.

Because *Zipcode* provides process management (whereas MPI does not), we will continue actively to develop, distribute, and support *Zipcode* for the next several years, principally to investigate inter-vendor MPI support, active messages, and threads, all working together.

Furthermore, the heterogeneous environment has not been addressed or explored fully in practice, and MPI does not address dynamic process models. All of these factors suggest that new *Zipcode* features can potentially provide a continued useful role in providing further input to a future MPI effort, while remaining a "full" portability platform. Of course, we hope that future generations of MPI will include dynamic process support and control, and strict guidelines for inter-vendor MPI interoperability. Our views on future work are well-characterized in Figures 4, 5, 6.

7 Conclusions

Zipcode currently provides portable message-passing capability on a number of multicomputers. It also works on homogeneous networks of workstations, and, with ports in progress, will soon be supported in several ways on heterogeneous networks, and should be readily portable to future heterogeneous multicomputers. The key benefits of *Zipcode* are its ability to limit the scope of message passing activities over sets of processes designated by the user (process groups), to define separate contexts of communication so that libraries can be written readily, and to allow different notations (virtual topologies) of process naming. Tagged message passing with rank naming of processes is included as a particular case of the notations supported by *Zipcode*.

We see notational abstraction as helpful in dealing with issues of non-uniform memory access hierarchies and heterogeneity in multicomputers and distributed computers. Abstraction is a way to help *Zipcode* find additional runtime optimizations, rather than a tacit source of inefficiency.

In the emerging MPI standard of message passing, we see many of the unique features of *Zipcode* represented in it: contexts, process groups plus contexts, virtual topologies, and invoices/pack/unpack technology. As such, the concepts in *Zipcode* have proven to be successful models of what will be implemented as standard capabilities by vendors, thereby enabling library development. When MPI becomes pervasive, *Zipcode*, as it is today, must become less important as a direct tool to achieve performance on real multicomputers. *Zipcode* will continue to provide a vehicle for testing advanced concepts in message passing as it has successfully done over the past five years. The additional kinds of research issues involving inter-vendor MPI, threads, and active messages mean that *Zipcode* research will take a new turn in the future. We will concentrate on using MPI to achieve standard message passing, and explore the new areas of parallel threads and active messages, that are outside MPI.

In closing, we note that *Zipcode* 1.00 will be made available through anonymous ftp and **netlib**, approximately December 1, 1993 with updates thereafter.

8 Acknowledgements

We acknowledge Dr. Charles L. Seitz, and Dr. Wen-King Su, both of Myricom, Inc. We acknowledge the help of Dr. Charles H. Still, Lawrence Livermore National Laboratory, for help with several *Zipcode* ports. We express our sincere gratitute to the reviewers for their numerous suggestions and constructive criticisms of the first version of this article.

References

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

[2] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint; (also, Proc 7th IPPS, April 1993).

[3] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[4] 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.

[5] 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.

[6] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[7] M.L. Branstetter, J.A. Guse, and D.M. Nessett. ELROS – An Embedded Language for Remote Operations Service. Technical Report UCRL-JC-108862, Lawrence Livermore National Laboratory, November 1991.

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

[9] R. Butler and E. Lusk. User's guide to the P4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[10] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the parmacs message–passing library. *Parallel Computing*, 1994. in this issue.

[11] J.J. Dongarra and R.A. van de Geijn. Two dimensional basic linear algebra communication subprograms. LAPACK Working Note 37, Technical Report CS-91-138, University of Tennessee, 1991.

[12] Nathan E. Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. An initial implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[13] 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.

[14] 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**.

[15] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[16] D. Frye, R. Bryant, H. Ho, R. Lawrence, and M. Snir. An external user interface for scalable parallel systems. Technical report, IBM, May 1992.

[17] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[18] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A Users' Guide to PICL – A Portable Instrumented Communication Library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, May 1992.

[19] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical Report MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[20] Leslie Hart and Tom Henderson. Summary of GP5 Library Routines, August 26 1992.

[21] Parasoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[22] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[23] 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.

[24] 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).

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

[26] 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).

[27] 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.

[28] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[29] Anthony Skjellum and Alvin P. Leung. *Zipcode:* A Portable Multicomputer Communication Library atop the Reactive Kernel. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 767–776. IEEE Press, April 1990.

[30] 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.

[31] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Charles H. Still, Alvin P. Leung, and Manfred Morari. Zipcode: A Portable Communication Layer for High Performance Multicomputing. Technical Report UCRL-JC-106725 (revised 9/92, 11/93), Lawrence Livermore National Laboratory, March 1991. To appear in Concurrency: Practice & Experience.

[32] 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!*, chapter 16. Morgan Kaufman, 1994.

[33] 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.

[34] Steven G. Smith, Robert D. Falgout, Charles H. Still, and Anthony Skjellum. High-Level Message-Passing Constructs for Zipcode 1.0: Design and Implementation. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, 1993.

[35] (Unattributed). XDR: External Data Representation Standard. Technical Report RFC 1014, Sun Microsystems, June 1987.

[36] (Unattributed). Draft Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. Technical Report Work Item Number JTC 1.22.21.x., IEEE, April 1993. P1003.4a/D7.

[37] (Unattributed). Elan Widget Library. Technical report, Meiko, 1993.

[38] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. Technical Report UCB/CSD 92/#675, UC Berkeley, Computer Science, 1992.

[39] Tammy Welcome. Programming in LMPS. Technical Report UCRL-MA-107031, Lawrence Livermore National Laboratory, March 1992.