[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 *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 767–776. IEEE Press, April 1990.

[13] Anthony Skjellum, Steven G. Smith, and Charles H. Still. The Zipcode System Users' Guide — version 1.00. Technical report, Lawrence Livermore National Laboratory, October 1992.

[14] Anthony Skjellum, Steven G. Smith, 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), Lawrence Livermore National Laboratory, March 1991. To appear in Concurrency: Practice & Experience.

[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.

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

[17] 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.

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

5 Conclusions

Zipcode currently provides portable message-passing capability on a number of multicomputers. It also works on homogeneous networks of workstations, and will soon be supported on heterogeneous networks and for heterogeneous multicomputers, when such systems are created. The key benefits of *Zipcode* are its ability to work over process lists designated by the user, to define separate contexts of communication so that message-passing complexity can be managed, and to allow different notations of message-passing appropriate to the concurrent algorithms being implemented. Typed message passing is seen as a special case of the notations supported by *Zipcode*.

We see notational abstraction as helpful in dealing with issues of memory hierarchy and heterogeneity in multicomputers and distributed computers. Abstraction is a way to help *Zipcode* find additional optimizations, rather than a tacit source of inefficiency. We believe that *Zipcode* implementations will be competitive in performance to typed message systems whenever vendors make low-level access to their hardware and system calls available to us during our implementation phase.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] Anthony Skjellum. *Zipcode:* The Killer Micros' Amanuensis. In Eugene D. Brooks and Karen H. Warren, editors, *The 1991 MPCI Yearly Report: The Attack of the Killer Micros*, pages 139–155. Lawrence Livermore National Laboratory, March 1991. Available as LLNL Technical Report UCRL-ID-107022.

[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.

- Since Fortran is not case sensitive, `CXT_` is prepended to uppercase *Zipcode* functions to indicated context-dependent communication (that is, where a default mailer is used from a stack of active mailers).

A few other minor changes appear because of case-sensitivity issues. See [13] for the definitive list.

4 Details of Execution

In this section, we cover the initialization and termination protocols, and discuss how to get node processes spawned in the *Zipcode* model of multicomputer programming. Within this model, the user is not allowed to call *CE/RK* functions directly, except for the `print()` utility, that replaces `printf()` on the nodes.

4.1 Initialization / Termination

Each host program, and node program must call the appropriate initialization function to initialize *Zipcode* for their process:

```
int error = Zip_init(void);         /* assume default mode for initialization */
    error = Zip_global_init(void);  /* assume a simpler host-master model */
void zip_exit(void);                /* terminate Zipcode session */
```

4.2 Process Creation / Destruction

The basic mailer manipulation commands (such as `g1_grid_open()`) require the specification of process lists, currently as ordered pairs of nodes and process ID's, packaged within a `ZIP_ADDRESSEES` structure. For application convenience, we supply optional commands to support the creation of such collections. One common collection is a *cohort*, a set of processes with the same process ID, distributed across a number of nodes. A cohort is often used to formulate a single program, multiple data (SPMD) calculation.
Cohort list creation:

```
ZIP_ADDRESSEES *addressees =
    zip_new_cohort(int N,          /* number of processes involved */
                   int node_bias,  /* node number of zeroth entry in list */
                   int cohort_pid, /* process ID of entire collection
                                      of processes */
                   int host_flag); /* flags whether host process participates */
```

Additionally, we provide a *Zipcode*-level spawn mechanism:

```
int result = zip_spawn(
                char *prog_name,            /* ASCII name of program to spawn */
                ZIP_ADDRESSEES *addressees, /* addressee list to spawn */
                void *state,                /* future expansion */
                int pm_flag);               /* flags if program is spawned on
                                               zeroth addressee */
```

where `result` is non-zero on failure. Most implementations require that this spawning function be effected in the host process, though the original *CE/RK* system did not make this restriction. A compatible `zip_kill()` is also defined:

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

With the addition of these functions, *Zipcode* specifies an entire programming environment that can be completely divorced, if desired, from its original foundations in the *Reactive Kernel / Cosmic Environment (CE/RK)*. This is possible so long as one can emulate appropriate *CE/RK* functions for *Zipcode* to use. This has been accomplished in the release 2.0 of nCUBE's 6400 system software, for instance.

```
        CALL ZIP_SIZEOF_INV(mailer, invoice, length)
        CALL ZIP_MALLOC(mailer, length, letter)
        CALL ZIP_PACK(mailer, invoice, ZIP_LETTER, letter, ZIP_IGNORE, length)

    if(length .eq. -1) then
C       an error occurred
        ...
```

The above example packs the same invoice that Example 1 does in C. The last two arguments to `ZIP_INV_ADD_INT()` and `ZIP_INV_ADD_DBLE()` are the "ignore-space" and "deferred-sizing" logicals, respectively, to be explained via examples below. They also appear in the C syntax, but as part of the argument string via special characters.

```
C       Example 2F

        POINTER invoice
        integer i, len, stride
        DOUBLE double_array(20)

        i      = 20
        len    = 10
        stride =  2

        call ZIP_INV_NEW(invoice)
        call ZIP_INV_ADD_INT(invoice, i, 1, 1, .false., .false.)
C
C The .true. argument invokes deferred sizing of the data:
        call ZIP_INV_ADD_DBLE(invoice, double_array, len, stride, .false., .true.)

C       pack or unpack call is made...

        len    = 5
        stride = 1

C       pack or unpack call is made...
```

This example performs the same work as the C Example 2 did. To ignore space in a pack or unpack call, the ignore-space logical is set true. For instance:

```
       call ZIP_INV_ADD_INT(invoice, ZIP_IGNORE, 1, 1, .true., .false.)
```

To ignore space and use deferred-sizing evaluation, both flags are set true:

```
      call ZIP_INV_ADD_INT(invoice, ZIP_IGNORE, len, stride, .true., .true.)
```

Other *Zipcode* calls are very simliar in Fortran to the C versions. A pre-processor is used to create some definitions for use by the Fortran programer. Pointers are declared using the `POINTER` type. This is a variable of sufficiently large size to hold a C pointer on a given machine.

The following conventions are followed in the Fortran interface.

- No functions. This avoids some problems on various machines with returning C values to Fortran code. Return values from C routines are passed as an extra argument to the Fortran interface.
- All items that are pointers in C are declared in Fortran as type `POINTER`.

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.

3.3 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 optimizations that can be done by *Zipcode*. To create a more flexible system, *Zipcode* provides the capability to do both the pack and communications in a single call. For instance,

```
g3_Pack_Send(ZIP_MAILER *g3mailer, int d1, d2, d3, Zip_Invoice *invoice)
```

takes care of creating the letter, packing the invoice and sending it to grid location specified by {`d1,d2,d3`}. Whenever possible, use Pack_Send-style routines, as they will generally be more runtime optimizable than pack calls followed by send calls.

A packed combine is also provided. The associative-commutative, user-defined function is specified differently than the non-packing version. The user routines takes as arguments two pointers to buffer spaces that must be unpacked before they are allowed to be used. The structure of the invoices must be known ahead of time or it can be created using the partial unpacking to get type information and then unpacking the rest of the buffer. Here is the specific syntax for the G2, 2D-grid packed combine:

```
int g2_pack_combine(ZIP_MAILER *g2mlr, Zip_Invoice *invoice, void (*func()) )
```

3.4 Fortran Interface

The Fortran (F77) interface is provided, but certain features have necessarily been omitted (awaiting Fortran 90). The syntax is different since there are no pointers in F77. There are no user defined types provided in the Fortran interface as Fortran does not provide structures. Once Fortran 90 has been adopted, user-defined types will likely appear in the Fortran interface.

Since Fortran does not allow variable argument functions the construction of invoices differs from the C interface. An invoice is built up over several function calls, each one specifying the next field in the invoice.

```
C       Example 1F
        POINTER mailer
        POINTER letter
        ...
        POINTER invoice
        integer i, length
        double double_array(20)

        call ZIP_INV_NEW(invoice)
        call ZIP_INV_ADD_INT(invoice, i, 1, 1, .false, .false.)
        call ZIP_INV_ADD_DBLE(invoice, double_array, 10, 2, .false., .false.)

C       use the invoice
```

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)
```

The structure `Method` is a composite of a pointer-to-function, and additional state information for a function call. The details of `Method` declarations are beyond the scope of this presentation.

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 so beware of 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. They must have the following parameter lists

```
int in(ZIP_MAILER *mailer, void *src, void *dest, int num_items,
       int stride, Extra *extra)

int out(ZIP_MAILER *mailer, void *src, void *dest, int num_items,
        int stride, Extra *extra)

int align(ZIP_MAILER *mailer, void *dest, Extra *extra)

int len(ZIP_MAILER *mailer, int num_items, int stride, Extra *extra)
```

Here, `src` is a pointer to the items to be converted and the `dest` parameter is a pointer to the space where converted items are stored. `num_items` is the number of items to be converted and `stride` is a stride factor for striding through arrays. `extra` is used to pass any miscellaneous information that is needed by the conversion. The user can pass an extra using the '%user-type(*)' notation discussed in `zip_new_invoice`.

The `align` variable returns the number of bytes to be added to `dest` to align the value properly. `len` returns the total size of `num_items`, in bytes; the `in()` and `out()` functions perform the conversion.

Finally, to remove a user-defined type from the system use the

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

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

3.2 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

```
Zip_Invoice invoice;
int i = 20, len = 10, stride = 2;
double double_array[20];

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

/* use the invoice (see below) */
letter = zip_malloc(mlr, zip_sizeof_invoice(mlr, invoice));
length = zip_pack(mlr, 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 */
...
invoice = zip_new_invoice("%i%&.&d", &i, &len, &stride, double_array);

/* use the invoice */
letter = zip_malloc(mlr, zip_sizeof_invoice(mlr, invoice));
length = zip_pack(mlr, 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(mlr, zip_sizeof_invoice(mlr, invoice));
length = zip_pack(mlr, 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                       */

invoice = zip_new_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 in several examples above.

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

To delete an existing invoice when there is no more need for it use `zip_free_invoice()`:

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

`zip_new_invoice()` returns 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.

One must be cautious of the scope of C variables when using '&'. For example, it is improper 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 things will happen if this is attempted.

The single character types that are supported are as follows:

'c' character

's' short

'i' int

'l' long

'f' float

'd' double.

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 *mlr;
char *letter;
...
```

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

/* set variables (P,Q) to grid shape: */
void g2_PQ(ZIP_MAILER *mailer, int P, int Q);
void G2_PQ(int P, int Q);

/* set variables (p,q) to grid position: */
void g2_pq(ZIP_MAILER *mailer, int p, int q);
void G2_pq(int p, int q);
```

These are the preferred forms for accessing grid information from G2-Class mailers.

G3-Class mail is a 3D-grid-abstraction class. A $P \times Q \times R$ grid naming abstraction is attached to the process list, analogously to the G2-Class 2D-grid primitives. This class is interesting for problems where there are three logical "axes" of concurrency. Children of a G3 mailer are G2 mailers, representing planes within the three-dimensional grid. Operations analogous to the 2D case have been extended to the 3D grid case, and are not detailed here (see [13, 14]).

Shorthands provide access to the PQ-plane, QR-plane, and PR-plane children, to which G2 grid operations may be applied, as above. Both context independent and context dependent versions are provided:

```
ZIP_MAILER *mailer;  /* 3D grid mailer */
ZIP_MAILER *PQ_plane_mailer, QR_plane_mailer, PR_plane_mailer;

PQ_plane_mailer = g3_PQ_plane(mailer);
QR_plane_mailer = g3_QR_plane(mailer);
PR_plane_mailer = g3_PR_plane(mailer);
PQ_plane_mailer = G3_PQ_plane();
QR_plane_mailer = G3_QR_plane();
PR_plane_mailer = G3_PR_plane();
```

3 High-Level Primitives

In order to facilitate easier use and the possibility of heterogeneous parallel computers, *Zipcode* provides 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 *Zipcode* commands. Letters are unstructured arrays of data, provided by *Zipcode* based on user specification; they are tied to specific mail contexts and are dynamically allocated and freed.

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-on-send- and scatter-on-receive-type semantics. In a heterogeneous environment, pack/unpacking will allow data conversions to take place without user intervention. Users that code with `zip_pack()`/`zip_unpack()` will have codes that are guaranteed to work in heterogeneous implementations of *Zipcode*.

3.1 Invoices

The `zip_new_invoice()` call creates new invoices:

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

```
G2_col_combine(buffer, comb_fn, size, items);

g2_row_fanout(mailer, &data, &length, orig_p, orig_q);
g2_col_fanout(mailer, &data, &length, orig_p, orig_q);
G2_row_fanout(&data, &length, orig_p, orig_q);
G2_col_fanout(&data, &length, orig_p, orig_q);
```

and

```
g2_row_fanin(mailer, dest_q, buffer, comb_fn, size, items);
g2_col_fanin(mailer, dest_p, buffer, comb_fn, size, items);
G2_row_fanin(dest_q, buffer, comb_fn, size, items);
G2_col_fanin(dest_p, buffer, comb_fn, size, items);
```

The row/column instructions above compile to G1-grid calls, since rows and columns of G2 mailers are implemented via G1 mailers.

Additional convenience shorthands facililtate host-process interaction, which is normally not part of a process grid. Here are some of the context-independent versions:

```
int error = G2_fanout_Host(&data, &length);
    error = G2_fanin_Host(&data, comb_fn, size, items);
```

G2-Grid mailer creation:

```
    ZIP_MAILER *mailer = g2_grid_open(int *P, int *Q, ZIP_ADDRESSEES *addr);
```

Once a G2 grid mailer has been established, it is possible to derive subgrid mailers by a cooperative call between all the participants in the original `g2_grid_open()`. In normal applications, this will result in a set of additional mailers in the postmaster process, and one additional G2 grid mailer in each node 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), initiates the subgrid open request as follows:

```
    /* array of pointer to subgrid mailers: */
    ZIP_MAILER **subgrid_mailers =
        g2_subgrid_open(ZIP_MAILER *mailer, /* mailer already opened */
              /* for each (p,q) on original grid, marks its subgrid: */
              int (*select_fn)(int p, int q, void *extra),
              void *select_extra;           /* extra data needed by select_fn() */
              int *nsubgrids);              /* the number of subgrids created   */
```

while each process in the original `g2_grid_open()` does a second, standard `g2_grid_open()`:

```
    ZIP_MAILER *subgrid_mailer = g2_grid_open(&P, &Q, NULL);
```

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

Finally, it is often necessary to determine the grid shape $P \times Q$, as well as the current process' location on the grid (p, q), when using 2D logical grids. Often this information is housed only in the mailer (though some applications may choose to duplicate this information). The following context-independent and context-dependent (with respect to the mailer most recently stacked with `Zip_push()`) calls provide simple access to these four quantities, while hiding the structure of the G2-Class `extra` structure, something with which applications should not directly concern themselves:

G1-Class mail 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.

G2-Class mail is a 2D-grid-abstraction class. A $P \times Q$ grid naming abstraction is attached to the process list; each process is specified by a (p, q) pair (*e.g.*, in the PO Box). Through inheritance, row and column mailers are defined in each process as the appropriate subsets of the 2D grid. This class has received the most extensive use because of the natural application to linear algebra and related computations.

Class-specific primitives for G2-Class mail have been defined for both higher efficiency and better abstraction. Small-g calls require mailer specification while big-G calls do not, analogous to the `zip_`- and `Zip_`-type calls defined generically above.

Letter-Generating Primitives:

```
char *letter = g2_Recv(ZIP_MAILER *mailer, int p, int q); /* unblocked: */
     letter = g2_Recvb(ZIP_MAILER *mailer, int p, int q); /* blocked: */
     letter = G2_Recv(int p, int q);  /* unblocked */
     letter = G2_Recvb(int p, int q); /* blocked */
```

Letter-Consuming Primitives:

```
void g2_Send(ZIP_MAILER *mailer, char *letter, int p, int q);
void G2_Send(char *letter, int p, int q);
```

Global operations *combine*, *broadcast* (fanout) and *collapse* (fanin) are defined and have been highly tuned for this class (see schematics in [7]).

Combines and fanins are over arbitrary associative-commutative operators specified by `(*comb_fn)()`. Broadcasts share data of arbitrary length, assuming all participants know the source. Collapses combine information assuming all participants know the destination:

```
int error = g2_combine(ZIP_MAILER *mailer,      /* 2D grid mailer */
                       void *buffer,            /* where result is accumulated */
                       void (*comb_fn)(),       /* operator for combine */
                       int size                 /* size of buffer items in bytes */
                       int items);              /* number of buffer items */
    error = G2_combine(void *buffer, void (*comb_fn)(), int size, int items);

    error = g2_fanout(ZIP_MAILER *mailer,
                      void **data,               /* data/result */
                      int *length,               /* data length */
                      int orig_p, int orig_q);   /* grid origin of data */
    error = G2_fanout(void **data, int *length, int orig_p, int orig_q);

    error = g2_fanin(ZIP_MAILER *mailer,
                     int dest_p, int dest_q,     /* destination on grid */
                     void *buffer, void (*comb_fn)(), int size, int nitems);
    error = G2_fanin(int dest_p, int dest_q, void *buffer, void (*comb_fn)(),
                     int size, int nitems);
```

Shorthands provide direct access to row and column children mailers, tersely providing common communication patterns within the two-dimensional grid:

```
g2_row_combine(mailer, buffer, comb_fn, size, items);
g2_col_combine(mailer, buffer, comb_fn, size, items);
G2_row_combine(buffer, comb_fn, size, items);
```

2.5.2 Mailer Creation

Mailers are created through a loose synchronization between the members of the proposed mailer's process list. A single process creates the process list, placing itself first in the list, and initiates the "mailer-open" call with this process information; it's called the "Postmaster" for the mailer, as initiator. The other participants receive the process list as part of the synchronization procedure. A special reactive process, the "Postmaster General," maintains and distributes zipcodes as mailers are opened; essentially the zipcode count is a single location of shared memory. Class-independent mailer creation:

```
ZIP_MAILER *mailer = zip_open(ZIP_CLASS *class,  /* class spec. */
              ZIP_ADDRESSEES *addr,  /* addressee list */
              ZIP_MAILER *parent,    /* parent, if any */
              void *extra,           /* class extra data */
              int *copyflg,          /* copying flags for addressee lists */
              ZIP_ZIPSET *zipset,    /* optional zipcode specification */
              ZIP_INHERIT *inherit)  /* sets details of mailer creation */
```

Typical call:

```
ZIP_MAILER *mailer = zip_open(class, addr, NULL, NULL, NULL, NULL, NULL);
```

2.5.3 Pre-Defined Letter Classes

Y-Class mail is used mainly for *Zipcode* internal mechanisms. The PO Box information is a single short integer type. Global operations are not implemented for this class, because of its (intentional) simplicity.

Z-Class mail is a general purpose class. Process names are abstracted to a single integer (based on position in the process list); receipt-selectivity is based on that source name. Global operations are implemented for this class, with analogous calling sequences to the G2-Class 2D-grid global operations noted below.

XX-Class mail emulates the x-primitives of *CE/RK*. This allows systems developed directly on *CE/RK* to work compatibly in a large application code. Ideally, such a path would only be used as a temporary mechanism, because direct use of more expressive *Zipcode* classes seems appropriate and would be more efficient in the usual case.

L-Class mail is used to support emulation of typed message notations such as Intel's NX or PICL [4]. Process names are not abstracted, and receipt selectivity is based on the source name (as { `node,pid`} pair) plus a type.

We have been able to classify a number of message-passing systems in [14], though specific differences in sending and receiving strategies exist between common typed-message-passing systems. In *Zipcode*, we define the L-class, which provides for receipt selectivity based on message source in unabstracted {`node,pid`}-notation, and based on a long-integer type. This class can be used to define one or more contexts of typed message systems, that call the primitives described fully in [13]. However, and perhaps more interestingly, these L-class calls can be used to generate wrappers for all the major typed message-passing systems. In the *Zipcode* manual we illustrate how this is done by showing a few of the wrappers for the PICL, NX and Vertex system [13]. We also have a long-standing *Zipcode*-based emulation for the Livermore Message Passing System (LMPS) [18].

Furthermore, for each context a user declares, he or 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 typed subroutines or programs to be brought together and face-lifted easily to work together, without changing types, or seeing when/where the message passing resources might conflict. In short, this provides a general means to ensure typed-message registry, as contemplated in [5].

2.5 *Zipcode* Calls

2.5.1 *Zipcode* Class-Independent Calls

Zipcode maintains a message-passing function naming convention and style analogous to *CE/RK*. For all classes, the same calls are valid for allocation, sending and receiving letters. Specific classes may define additional calls to increase the convenience of use (see G2- and G3-class calls further below). Calls with the `zip_` prefix require specification of the mailer relative to which a letter is to be created, sent or received. Calls prefixed with `Zip_` depend on the current mailer context established by `Zip_push()/Zip_pop()` calls. As such, they omit mailer arguments. The `zip_mail() and Zip_mail()` calls transmit to all addressees of a mailer (analogous to *CE/RK*'s `xmsend()`).
Context-setting Primitives:

```
int error = Zip_push(ZIP_MAILER *mailer);
int error = Zip_pop(void);
```

Letter-Generating Primitives:

```
char *letter = zip_malloc(ZIP_MAILER *mailer, int length);
      letter = zip_recv(ZIP_MAILER *mailer);   /* unblocked */
      letter = zip_recvb(ZIP_MAILER *mailer);  /* blocked */

      letter = Zip_malloc(int length);
      letter = Zip_recv(void);      /* unblocked */
      letter = Zip_recvb(void);     /* blocked */
```

Letter-Consuming Primitives:

```
void zip_send(ZIP_MAILER *mailer, char *letter, int node, int pid);
void zip_msend(ZIP_MAILER *mailer, char *letter, int count, int *proc_list);
void zip_free(ZIP_MAILER *mailer, char *letter);

void Zip_send(char *letter, int node, int pid);
void Zip_msend(char *letter, int count, int *proc_list);
void Zip_free(char *letter);
```

Abstraction to process-list addressees:

```
void zip_mail(ZIP_MAILER *mailer, char *letter);
void Zip_mail(char *letter);
```

Variations of the basic `zip_send()/Zip_send()` and `zip_mail()/Zip_mail()` macros are provided for determining the disposition of the letter's PO Box information. The three versions alternatively use a default value for the PO Box, accept an argument as a pointer to the contents of the PO Box to be used, or assume the PO Box is preset correctly in the letter's envelope. `zip_mail()/Zip_mail()` applied to appropriately inherited child mailers, allows specification of arbitrary, user-defined recipients subsets of the original mailer's addressees. Similarly, variations of `zip_recv()/Zip_recv()` and `Zip_recv()/Zip_recvb()` calls are provided to control receipt selectivity based on a default PO Box, a PO Box specified as an extra argument, or bypassing PO Box-based selectivity entirely, thereby accepting the next message in the correct zipcode context. Lower-level calls allow replacement of the receipt-selectivity method (the delivery function), in order to allow flexible user-defined interpretations of the PO Box data in an incoming letter. The latter feature can be used to define "wildcard" message receipt selectivity, for example.

Any host/node data-format conversions to the cover information are automatically performed without any user intervention. This feature causes additional load only in the host process.

- `freemc(void)` deallocates the allocated multicomputer,
- `spawn(char *prog, int node, int pid, char *state)` spawns processes on one or more nodes,
- `ckill(int node, int pid)` kills processes on one or more nodes.
- `cosmic_init(void)` starts the environment in a process,
- `cosmic_exit(void)` ends the environment in a process,
- `mynode(void)` returns the current process' logical node number,
- `mypid(void)` returns the current process' identification number (pid),
- `nnodes(void)` returns the number of nodes in the multicomputer allocation,
- `print(char *fmt, ...)` resembles the standard C function `printf()` except that the output is preceded by {`node,pid`} identification, and terminated by a newline, with the output buffer automatically flushed,
- `clock()` returns the running nodal clock value in microseconds.

2.4 *RK* Calls

The *RK* calls required by *Zipcode* are as follows:

- `msg = xmalloc(int length)` allocates a message buffer of `length` bytes and returns a pointer `msg` to it,
- `xfree(char *msg)` deallocates the message buffer pointed to by `msg`,
- `xlength(char *msg)` determines the length (in bytes) of a previously allocated or received message `msg`,
- `msg = xrecv()` receives a message without blocking, returning a pointer to the message if a message can be received and NULL if there is no message queued to the calling process,
- `msg = xrecvb()` blocks until a message can be received and returns a pointer to the message,
- `xsend(char *msg, int node, int pid)` sends a message pointed to by `msg` to the process `pid` on node `node`,
- `xmsend(char *msg, int count, int *dest)` sends a message `msg` to multiple destinations specified by the integer array `dest` of length `2*count` in the form { `node0, pid0, node1, pid1, ...`} .

It is important to note that `xsend()` and `xmsend()` deallocate the message buffer after sending the message; they are semantically analogous to `xfree()`. The receive functions `xrecv()` and `xrecvb()` are semantically analogous to `xmalloc()`.

Zipcode-based programs are not to call any of these *CE* or *RK* functions directly. Both message passing and environment control are represented in *Zipcode*.

2 Low-Level Primitives

2.1 *CE/RK* Overview

To appreciate the model upon which current *Zipcode* implementations are built, one needs to understand the scope and expressivity of the low-level *CE/RK* system.

2.2 Interface with the *CE/RK* system

Implementations of *Zipcode* to-date interface to primitives of the *CE/RK*, a portable, light-weight multicomputer node operating system, which provides untyped blocked and unblocked message passing in a uniform host/node model, including type conversion primitives for heterogeneous host-node communications. Presently, the *Reactive Kernel* is implemented for Intel iPSC/1, iPSC/2, Sequent Symmetry, Symult S2010, Intel iPSC860 Gamma prototype multicomputers, with emulations provided for the Intel Delta prototype, Thinking Machines CM-5, and nCUBE/2 6400 machines. Furthermore, Intel will provide the *CE/RK* primitives at the lowest-level (read: highest performance) on its Paragon system. *CE/RK* is also emulated on shared-memory computers such as the BBN TC2000 as well as networks of homogeneous NFS-connected workstations (*e.g.*, Sun clusters). We see *CE/RK* primitives as a logical, flexible platform for our work, and for other message passing developers, and upon which higher-level layers such as *Zipcode* can be ported. Because most typed message-passing systems with restrictive typing semantics do not provide quite enough receipt selectivity directly to support *Zipcode*, we find it often best to implement untyped primitives as the interface to which *Zipcode* works. The *CE/RK* emulations, built most often on vendor primitives, make use of any available typing for bookkeeping purposes, and allow users of a specific vendor system to mix vendor-specific message passing with *CE/RK*- or *Zipcode*-based messaging.

One should view the *CE/RK* system as the default message-space management system for *Zipcode* (in C++ parlance, default constructor/new, destructor/free mechanisms), with the understanding that future implementations of *Zipcode* may rather use more primitive calls (*e.g.*, packet protocols or active messages) to gain even greater performance (alternatively, if Paragon or similar implementations are very fast, such shortcuts will have commensurately less impact on performance). Via the shortcut approach, *Zipcode* analogs of *CE/RK* calls will become the lowest level interface of message passing in the system, and become a machine dependent layer.

2.3 *CE* Programs

The *Cosmic Environment* provides control for concurrent computation through a "cube dæmon." This resource manager allows multiple real and emulated concurrent computers to be space-shared [6]. The following functions are provided, and we emulate these on systems that provide analogous host functionality (this emulation can be done efficiently on the nCUBE/2, almost trivially on the Gamma, and not at all on the Delta and CM-5):

- `getmc` gets multicomputer resource by type and size for space-sharing (also called `getcube`)
- `freemc` frees multicomputer resource currently in use (also called `freecube`)
- `spawn` spawns one or more node processes to a previously allocated multicomputer partition
- `ckill` kills one or more extant processes on a previously allocated multicomputer partition.

2.3.1 *CE* Calls

To support *Zipcode*, we normally emulate the *CE* functions below. Again, some of these functions, particularly `getmc()`, `freemc()`, `spawn()` and `ckill()`, are not available on all implementations (for instance, the Delta), and are restricted to the host program:

- `getmc(char *computer_name, int nnodes)` allocates a multicomputer,

than instructions on how to do the desired operations, we are able to discover optimizations. Low-level notations cannot hope to achieve this type of optimization, because they do not expose the semantic information in their instructions, nor work over process-lists, for which special properties may be asserted (except with extensive compile-time analysis).

This evolutionary process implies that *Zipcode* has surpassed its original *Reactive Kernel / Cosmic Environment* platform; 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 (with and/or without an intervening *CE/RK* layer),
- Active-message strategies (*cf*, [17]),
- Pure Message Passing (with and/or without an intervening *CE/RK* layer),
- Control-Network operations definable on process lists (subsets of processes or processors).

Heterogeneous translation can be by one of several translation mechanisms, for instance: XDR [16], ELROS [1, 2], or other strategies (that appropriately balance the work of the sender and recipient in the translation process as a function of their computational bandwidth for such translations). Because *invoice*s are persistent objects, the possibility of nodal vectorization of translated objects is possible, using ELROS or other machine-specific strategies (perhaps user-defined); XDR is not currently amenable to vectorization. Such translation strategies will also be held transparent to the user, except when the user chooses to intervene, by providing a sub-method that implements part of an *invoice* translation. With this approach, we can take advantage of the architectures presented at runtime, on a context-by-context basis.

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 hierarchies. This multi-faceted approach to implementing *Zipcode* follows its original design philosophy; originally, the *CE/RK* primitives upon which *Zipcode* is based 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 lists, and contexts, to get desirable mappings from which *Zipcode* can effect runtime optimizations.

The *CE/RK* primitives (originally central to *Zipcode*) manage memory as well as message-passing operations. This is an important feature, carried into the *Zipcode* system, which is helpful in reducing the number of copies needed to pass a message from sender to recipient in basic message (hence, less wasted bandwidth). In *CE/RK* the system provides message space, which is freed upon transmission, and allocated upon receipt. This approach removes the need for complicated strategies involving asynchronous sends, in which the user has to poll to see when his or her buffer is once again usable. Since the majority of transmissions in realistic applications involve a gather before send (and scatter on receive), rather than block-data transmission, these semantics provide, on-the-whole, good notational and performance benefits, while retaining simplicity. *Zipcode* extends the concept of the *CE/RK*-managed messages to include buffered messages (for global operations), and synchronizations. These three varieties of primitives make different assumptions about how memory is allocated (and by whom), and are implemented with the most efficient available system calls in a given *Zipcode* implementation. In all cases, actual memory allocation can be effected using light-weight allocation procedures in efficient implementations, rather than heavy-weight mallocs. Therefore, the dynamic nature of the allocations need not imply significant performance penalties.

When moving *Zipcode* to a new system, the *CE/RK* layer will normally be the first interface provided, with additional interfaces provided if the hardware's special properties so warrant. In this way, user codes and libraries will come up to speed quickly, yet attain better performance as the *Zipcode* port is optimized for the new system. We see this as a desirable mode of operation, with the highest initial return on investment.

system. Registry is needed so that diverse codes can be brought together and made to work without the possibility of message-passing conflicts, and without the need to globalize the semantic and syntactic issues of message passing contained in each separate piece of code. For instance, the use and support of independently conceived concurrent libraries requires such context registry. As applications mature, more contexts are likely to be needed, especially if diverse libraries are linked into the system, or a number of process structures are needed to represent various phases of a calculation. In purely message-passing instances within *Zipcode*, contexts control the flow of messages through a global messaging resource. In more complex hierarchies, contexts will manage channel and/or shared-memory blocks in the user program, while the notation remains message-passing-like to the user. This evolution is transparent to the user.

Concurrent mathematical libraries are well supported by the definition of multi-dimensional, logical-process-grid primitives, as provided by *Zipcode*; one-, two- and three-dimensional grids are currently supported (grid *mail classes*). Grids are used to assign machine-independent naming to the processes participating in a calculation, with a shape chosen by the user. Such grids form the basis for higher-level data structures that describe how matrices and vectors are shared across a set of processes, but these descriptors are external to *Zipcode*. New grids may be aligned to existing grids to provide nesting, partititioning, and other desired subsetting of process grids, all done in the machine-independent notation of the parent grid.

Mail classes (such as new grid structures) may be added statically to the system; because code cannot move with data in extant multicomputers, mail classes have to be enumerated at compile-time. Because we at present retain a C implementation, rather than C++, the library must currently be modified explicitly to add new classes of mail, rather than by inheritance. Fortunately, the predefined classes (grids, typed-messages) address a number of the situations we have encountered thus far in practical applications. Soon, we plan to add tree-style mail classes. Non-mathematically oriented users may conceive of mail classes that we have not as yet imagined, and which might be application specific.

Recently, we have evolved the *Zipcode* system to provide higher-level application interfaces to the basic message-passing contexts and classes of mail. These interfaces allow us to unify the notions of heterogeneity and memory hierarchy in a single framework, on a context-by-context basis. For instance, we view a homogeneous collection of multicomputer nodes as a particular type of memory hierarchy. We see this unification of heterogeneity and memory hierarchy in our notation as an important conceptual advance, both for distributed- and concurrent-computing applications of *Zipcode*. Mainly, heterogeneity impacts transmission bandwidth, and should not have to be treated as a separate feature in data transmission, nor should it be explicitly visible in user-defined application code or algorithms, except perhaps in highly restricted method definitions, for performance' sake.

For instance, the notations currently provided by *Zipcode* support writing application programs so that the same message-passing code can map reasonably well to heterogeneous architectures, to architectures with shared memory between subsets of nodes, and to architectures which support active-message strategies. Furthermore, it should be possible to cache limited inter-node channel resources within the library, transparent to the user. This is possible because the gather-send and scatter-receive notations remove message formatting from the user's control. We provide general gather/scatter specifications through persistent *invoice* data types. This notation is available both to C and Fortran programmers. As a side-effect, we provide a clean interface for message-passing in the Fortran environment. If compilers support code inlining and other optimizations, we are convinced that overheads can be drastically reduced for systems with lighter communication overheads than heretofore developed. Cheap dynamic allocation mechanisms also help in this regard, and are easily attainable. In all cases, the user will have to map the process lists to processors to take advantage of the hierarchies, but this can be done systematically using *Zipcode*.

We define operations on a context-by-context basis (methods), so that the methods implementing send, receive, combine, broadcast, and so on, are potentially different for each context, reflecting optimizations appropriate to given parts of a hierarchy (homogeneity, power-of-two, flat shared memory, *etc*). We have to rely on the user to map the problem to take advantage of such special contexts, but we provide a straightforward mechanism to take advantage of hierarchy, through the gather/scatter notation. When compilers provide inlining, we will see significant improvements in performance for lower-latency realizations of the system. Higher-level notation, and context-by-context method definitions are key to optimizing for memory hierarchy and heterogeneity. Because the user provides us with information on the desired operations, rather

The *Zipcode* Message-Passing System*

Anthony Skjellum, Steven G. Smith, Charles H. Still

Lawrence Livermore National Laboratory
Numerical Mathematics Group

Alvin P. Leung

Syracuse University Northeast Parallel Architectures Center

Manfred Morari

California Institute of Technology
Chemical Engineering

September 15, 1992

1 Overview of *Zipcode*

Zipcode is a message-passing system developed originally by the first author, beginning at Caltech in the Summer of 1988 and continuing to-date [8, 12, 14, 15]. *Zipcode* was created to address features and issues absent in then-existing message-passing systems (most of these issues and features remain largely unaddressed in other systems). Currently, *Zipcode* continues to serve as a vehicle to demonstrate high-level message-passing research concepts and, more importantly, to provide the basis for supporting vendor-independent scalable concurrent libraries; notably, the *Multicomputer Toolbox* [3, 10, 9, 11]. The basic assertion of *Zipcode* is that carefully managed, expressive message-passing is an effective way to program multicomputers and distributed computers, while low-level message passing is admittedly both error-prone and difficult.

The purpose of *Zipcode* is to manage the message-passing process within parallel codes in an open-ended way. This is done so that large-scale software can be constructed in a multicomputer application, with reduced likelihood that software so constructed will conflict in its dynamic resource use, thereby avoiding potentially hard-to-resolve, source-level conflicts. Furthermore, the message-passing notations provided are to reflect the algorithms and data organizations of the concurrent algorithms, rather than pre-defined typing (tagging) strategies. Typing, while generic and easy-to-understand, proves too low-level to support manageable application development well. Notational abstractions provide a means for the user to incorporate information through which the *Zipcode* library may make optimizations when a code runs on systems with specific hardware features. Abstraction is therefore seen as a means to higher performance, and notation is seen as a means towards more understandable, easier-to-develop and maintain concurrent software. Context registry (see below) provides a "social contract" within which multiple libraries and codes can co-exist reasonably.

The main type of registry is communication *context* control; the main process data structure is the process list (a collective of processes that are to communicate). These constructs are handled dynamically by the

*Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.