AIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202

Centimeter

Inches

MANUFACTURED TO AIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

Conf-9310220-6

UCRL-JC-115603
PREPRINT

High-Level Message-Passing Constructs for Zipcode 1.0: Design and Implementation

S.G. Smith
R.D. Falgout
C.H. Still
A. Skjellum

This paper was prepared for submittal to the
1993 Scalable Parallel Libraries Conference
Starville, MS
October 6-8, 1993

October 1993

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

MASTER
DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED
js

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

High-Level Message-Passing Constructs for Zipcode 1.0: Design and Implementation*

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

Numerical Mathematics Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

Anthony Skjellum
Department of Computer Science
Mississippi State University
Mississippi State, MS 39762-5623

Abstract

Zipcode is a message passing system that was initially designed for multicomputers and homogeneous networks of computers. This paper describes Zipcode "invoices," which raise the message-passing interface of Zipcode to a higher level of abstraction. The "gather-send" and "receive-scatter" semantics enable heterogeneous communication. The higher level of abstraction also simplifies message passing and reveals more optimizations. We explain the utility of these features and give examples of the calling sequences that implement them. All of these features are seen as enablers for parallel library development and large applications.

1 Introduction To Zipcode

Zipcode is a message-passing system designed and developed originally by Anthony Skjellum, beginning at Caltech in July, 1988 and continuing to-date [4, 5, 6, 3]. It was created to address issues and provide features absent in then-existing message-passing systems, many of these feature have been included in MPI [2]. Zipcode continues to serve as a vehicle to demonstrate high-level message-passing research concepts. Central to the development of Zipcode is the view that message-passing is an effective way to program multicomputers and distributed computers.

The primary design goal of Zipcode is to manage message-passing within parallel applications and libraries. Communication contexts partition the

message-passing space into distinct spaces to provide "safe" environments for message-passing. Messages in a given context are guaranteed not to interfere with those in another context. This enables the development of large applications and libraries without fear of message interference between applications and libraries. Collective operations are implemented based on communication contexts rather than being global in scope, as is the case with most existing systems. In addition, Zipcode provides virtual topologies, which enable the construction of user-specified message-passing notations. These notations can be designed to reflect the algorithms and data organization of the application rather than having the application map to a single message-passing notation. For example, Zipcode includes support for logical process grids of one, two and three dimensions. Virtual topologies facilitate manageable application development.

For actual message passing operations, the high-level notation (*e.g.*, pack-and-send-style operations) explained here provide a means for the user to incorporate information through which the Zipcode library may make optimizations. Abstraction is therefore seen as a means to portable performance. Higher-level notation is also seen as a means towards more understandable, easier-to-develop (and maintain) explicitly concurrent programs. This high-level mechanism is part of a recent trend in our improvement to the system. Originally, users managed their own *letters*, including the format of data within the letter. Now, the preferred form of Zipcode programming totally hides letter formatting from the user. As such, it is also capable of encapsulating heterogeneity. This highly desirable feature makes it possible for correctly written Zipcode programs to move to a heterogeneous environment.

The work described here was finished before the

*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.

MPI standardization process was heavily active. As such it represents a usable system for the development of software today. The authors fully support the MPI standardization effort and will be moving to this standard. When MPI becomes pervasive, *Zipcode* will become less important as a direct tool to achieve performance on real multicomputers, but will continue to provide a vehicle for testing advanced concepts in message passing, as it has successfully done over the past five years. For reasons of development latency (MPI is very large, and vendors have limited resources), we will continue to actively develop, distribute, and support *Zipcode*, for the next one to two years, and perhaps longer.

2 Heterogeneity

The move towards higher speed networks and faster workstations has focused large amounts of attention on grouping workstations and super-computers together to form distributed multicomputers. We were eager to be able to take advantage of these developments, however, *Zipcode* did not include support for heterogeneous machines. *Zipcode* was based on the traditional "buffered" message passing semantics where a user allocates a buffer and then fills in the buffer for sending. These primitive semantics were expanded by systems (like PVM [1]) to hide any knowledge of the communications buffer from the user. Instead the message-passing interface focused on what user-space variables were to be communicated. This enables "type" information to be easily passed to the message-passing system in order to facilitate any data-conversion that must be done. Data-conversion includes such issues as byte-order and size differences between processes.

In *Zipcode* we developed the concept of "invoices" to achieve this abstraction. Invoices are based on an analogy with an invoice (packing slip) that is commonly included when shipping a package. An invoice indicates the variables a user wishes to communicate and thus eliminates user control over communication buffers. Invoices will be described more thoroughly in the next section.

In order to enable data-conversion, a design decision must be made as to which processes are responsible for doing the actual conversion, if needed. In *Zipcode* we did not enable user control of this feature. It was felt that allowing user-control would encourage the writing of non-portable code (e.g. moving the code from one group of workstations to another). *Zipcode*

Figure 1: Conversion in *Zipcode* is based on communication contexts

is thus responsible for determining when data conversion is to be done. There are two extreme approaches that could be taken. One is to have all communication take place in some predetermined global storage representation (e.g. XDR). This has performance implications when the machine representation differs from this global representation on the majority of workstations. Another extreme approach would be to have storage representation information passed in the message and have the receiving process do the conversion. This implies that all processes have the capability to convert from every other storage representation as well as enlarging the length of messages.

Zipcode takes an intermediate approach and bases conversion on the context of communication. When a context is created, *Zipcode* determines which storage representation to use for that context. Processes that are part of the context are responsible for converting to/from this context storage representation. For an example, see Figure 1. If machine 2 sends a message to machine 1 in context 1, machine 2 has the responsibility to convert the outgoing message to context 1 storage (storage type A). Similarly, if machine 2 is sending a message to machine 1 in context 2, machine 1 is responsible for converting from the storage representation for context 2 (type B storage).

When the initial grouping of machines is done, *Zipcode* can attempt to group machines with the same storage representations together thereby reducing the number of conversions that need to be done. Contexts provide a natural and convenient way for users to convey information about which processes will be working together in an algorithm. Currently *Zipcode* does not fully support the heterogeneous model described.

3 Invoices

As indicated previously, invoices are *Zipcodes* way of having the user specify the variables that they wish to communicate. This higher-level interface enables several things to happen. First, the message-passing interface becomes easier to use. The user no longer has to perform the simple task of data motion from user variables to communication buffers. On systems that have native support for gather-scatter semantics, invoices provide a simple interface to take advantage of this feature. Second, as was described in the previous section, heterogeneity is enable since *Zipcode* is informed of the types of items that are being communicated. The last feature that invoices enable is "optimization hiding". This feature will be discussed later. This section describes the primitives which are used to control invoices and how to use them.

3.1 Invoice Creation

Invoice creation is the most difficult part of their use. The actual communication primitives are very similar to the buffered approach. Part of the difficulty of invoice construction arises from the many options that were included. While a simpler interface is possible we wanted to develop a system which was flexible enough to enable efficient implementations on existing and future platforms. The implementation that was developed was for proof-of-principle and as such is layered on top of existing *Zipcode* constructs. This approach, while allowing for quick development, is not the most efficient way to implement the primitives and does not take advantage of any hardware specific features on any of the platforms currently supported by *Zipcode*.

The `zip_new_invoice()` call creates new invoices:

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

`zip_new_invoice()` constructs 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 length (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).
- An optional '@' character indicating that the variable is overlaid (see below).
- A character specifying an internal type or a string indicating a user type.

For both the length or stride, '*' or '&' can replace the hard-coded integer in the format string. If '*' is used, then the next argument on the argument list is used as an integer expression specifying the length (or stride). '*' allows the size of an invoice item (or stride) to be specified at invoice construction time. Both the length or stride factor can be indirected by using '&' instead of an integer. The '&' indicates that a pointer to an integer follows on the argument list, this integer will be examined for the length (or stride) when the invoice is used. When '&' is used, the length (or stride) 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. Note that 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.

The built-in types are:

c	character
s	short
i	int
l	long
f	float
d	double

For each conversion specification in the format string, a pointer to an array of that type must be passed as an argument. This array is where *Zipcode* will pack/unpack variables. If the '@' overlay operator option is used then a pointer to a pointer to an array must be passed. The overlay operator is used to prevent an extra copy from being made on systems with communications buffers. The overlay op-

erator indicates that *Zipcode* is responsible for allocating and deallocating storage for that array. On systems which use communications buffers, *Zipcode* “overlays” the user variable on top of the communications buffer. This enables the “trick” of manipulating values inside of a communications buffer. Without the overlay operator a copy is implied in all cases which would introduce greater overhead. On systems that allow gather/scatter communication this notation is not needed since data can be moved from the network directly into scattered user-space. This feature should be used only for temporary variables that will be discarded. The communications buffer space can not be freed until the user releases this space with the `zip_clear()` function.

Some examples of invoice creation are:

```
Zip_Invoice *invoice;
int i[10];
double d[10];
```

```
zip_new_invoice(&invoice, "%10i %5.2d",
i, d);
```

This specifies that the user wishes to send all the elements of the integer array *i* and the even elements of the double array *d*.

```
Zip_Invoice *invoice;
int i[10];
double d[10];
int length;
```

```
length = 10;
zip_new_invoice(&invoice, "%*i %5.2d",
length, i, d);
```

This invoice contains the same information as the previous example but uses the ‘*’ operator to control the length of the integer array.

```
Zip_Invoice *invoice;
int i[10];
double d[10];
int length;
```

```
zip_new_invoice(&invoice, "%&i %5.2d",
&length, i, d);
length = 10;
/* first use of invoice */
length = 5;
/* second use of invoice */
```

In this example, the deferred sizing feature is demonstrated. In the first use of the invoice the entire array

Figure 2: Overlay example for an invoice “10@i %5.2d”

i would be packed; in the second use only the first five elements of *i* would be packed.

```
Zip_Invoice *invoice;
int *i;
double d[10];
```

```
zip_new_invoice(&invoice, "%10@i %5.2d",
&i, d);
```

The example above demonstrates the use of the overlaid feature. In this case the array *i* is overlaid. Figure 2 indicates what would happen if this invoice were used in a receive operation on a system that used contiguous communications buffers. The values for array *d* would be copied from the communications buffer. The pointer for array *i* would be directed towards the communications buffer to avoid a copy.

3.2 User-defined Types

User-defined types may be added to the system to ease the packing of complicated data structures. A new type is registered with the `zip_register_invoice_type()` function. This call has the form:

```
zip_register_invoice_type(char *name,
int (*in)(),
int (*out)(),
int (*sizeof)(),
int (*align)(),
int (*new_in)(),
int (*new_out)());
```

where *name* is the name to be used in the conversion specification in a `zip_new_invoice()` call. The

functions are user-defined functions called by *Zipcode* to perform data motion to/from a communications buffer, where `in()` converts from a buffer to a local structure, `out()` converts from a local structure to a buffer, `align()` returns the number of bytes needed to properly align the new type, `sizeof()` returns the size (in bytes) of the structure, `new_in()` creates a new structure based on a buffer and `new_out()` creates new structure overlaid on a buffer.

The following is a sample implementation for a simple matrix structure. The `out_matrix()` function copies from a structure to a flat buffer space. The function calls invoked are the data-conversion functions associated with the context specified in the `mlr` structure.

```
typedef struct
{
    int m,n;
    double **array;
} matrix;

out_matrix(mlr, src, dest, len, stride,
          extra)
ZIP_MAILER *mlr;
matrix *src;
char *dest;
int len, stride;
Extra *extra;
{
    ZIP_CALL_INT_OUT(mlr, &(amp;src->m),
                    dest, 1, 1);
    dest += ZIP_CALL_INT_SIZEOF(mlr,
                                &(amp;src->m), dest, 1, 1);

    ZIP_CALL_INT_OUT(mlr, &(amp;src->n),
                    dest, 1, 1);
    dest += ZIP_CALL_INT_SIZEOF(mlr,
                                &(amp;src->n), dest, 1, 1);

    dest += ZIP_CALL_DOUBLE_ALIGN(mlr,
                                *src->array, dest,
                                (src->m)*(src->n), 1);
    ZIP_CALL_DOUBLE_OUT(mlr, *src->array,
                        dest, (src->m)*(src->n), 1);

    return 0;
}
```

Once a type has been registered it can be used in an invoice just like the built-in types. For example,

```
Zip_Invoice *invoice;
matrix mat;
```

```
matrix *mat_ptr;
```

```
zip_register_invoice_type("matrix",
                          in_matrix, out_matrix,
                          sizeof_matrix, align_matrix,
                          new_convert_in_matrix,
                          new_out_matrix);
```

```
zip_new_invoice(&invoice, "%matrix", &mat);
zip_new_invoice(&invoice, "%0matrix",
               &mat_ptr);
```

When a type is no longer needed it can be unregistered with

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

to free any storage that was allocated for that type.

Registered types have a special operator `(*)` that can be applied to them. This enables passing of information to the conversion operations. These are received through the `extra` parameter (see the `out_matrix()` example given above). The information could be used to specify that a submatrix is to be communicated:

```
Zip_Invoice *invoice;
matrix mat;
struct
{
    int start_x, end_x;
    int start_y, end_y;
} sub_matrix;
```

```
zip_new_invoice(&invoice, "%matrix(*)",
               &mat, &sub_matrix);
```

The creator of the new invoice type is responsible for defining the meaning of any extra arguments that are needed for that type.

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

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

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

3.3 Invoice Communication Primitives

The invoice communication primitives are very similar to existing *Zipcode* communication primitives. In most cases the only distinguishing difference is the communication buffer parameter is replaced with an invoice and any length information is omitted (this is contained in the invoice).

Point to point communication is achieved through the use of the `pack_send()` and `pack_recv()` primitives. Since each virtual topology has its own set of communication primitives, we present examples from the one dimensional grid case. The point-to-point communication primitives are:

```
g1_pack_send(ZIP_MAILER *mlr,
             Zip_Invoice *invoice, int destination);
```

Send a package to destination.

```
g1_pack_recv(ZIP_MAILER *mlr,
             int source, Zip_Invoice *invoice);
```

Receive a package from source, non-blocking version.

```
g1_pack_recvb(ZIP_MAILER *mlr,
             int source, Zip_Invoice *invoice);
```

Receive a package from source, blocking version.

```
g1_pack_mrecvb(ZIP_MAILER *mailer,
               Y_COVER *(*spdlvry)(),
               int *total_to_receive,
               int n, int *d1, int *max_count,
               int *recv_count, Zip_Invoice **invoices);
```

Receive one or more packages from one or more sources. This command is blocking.

The following example sends an integer and double to grid process 1 and then receives an integer and a double back from grid process 1.

```
int i=1;
double d=1.0;
zip_new_invoice(&invoice, "%i%d", &i, &d);
g1_pack_send(mailer, invoice, 1);
g1_pack_recvb(mailer, 1, invoice);
```

The overlay operator requires that *Zipcode* control the allocation and deallocation of space for overlaid variables. `zip_create()` signals that the user is ready to use the overlaid variables in a specific invoice and is used before sending an invoice. Following a call to `zip_create()` the user can access overlaid variables. Once the invoice is sent the user is not allowed to touch the overlaid variables. On the receiving side, a user can access overlaid variables when an invoice has been received. After the user is done using the overlaid variables they must signal that they are finished with the `zip_clear()` command. Once `zip_clear()` has been invoked on an invoice, it is illegal to access any overlaid variables in that invoice. `zip_create()` and `zip_clear()` are not needed for invoices that do not contain overlaid variables. The following code sample

shows the use of overlaid variables; it performs the same communication as the previous example. Note that `zip_create()` requires a mailer in order to determine any data-conversion requirements. Also note that access to the overlaid variables only occurs between `zip_create()` and `g1_pack_send()` on sending and between `g1_pack_recv()` and `zip_clear()` on receiving.

```
ZIP_MAILER *mailer;
Zip_Invoice *invoice;
double *d;
int *i;

zip_new_invoice(&invoice, "%i%d",
               &i, &d);

zip_create(mailer, invoice);
*i = 1;
*d = 1.0;
g1_pack_send(mailer, invoice, 1);

g1_pack_recv(mailer, 1, invoice);
printf("d=%d%lf\n", *i, *d);
zip_clear(invoice);
```

The collective operations in *Zipcode* work over a context of communication. *Zipcode* provides the standard set of collective operations: combine, fanin (reduction), fanout (broadcast) and parallel prefix (scan). Combine, fanin and prefix take an associative commutative operation as an argument. *Zipcode* provides several built-in operations and also allows for user-defined operations. The built-in operations are:

Addition	Multiplication
Maximum	Minimum
Absolute Maximum	Absolute Minimum
Binary Or	Binary Xor
Binary And	Logical Or
Logical Xor	Logical And

The built-in operations work on invoices that contain only built-in types. The one dimensional grid primitives are:

```
g1_pack_fanout(ZIP_MAILER *mailer,
               Zip_Invoice *invoice, int source);
g1_pack_combine(ZIP_MAILER *mailer,
               Zip_Invoice *invoice, int (*func)());
g1_pack_fanin(ZIP_MAILER *mailer,
              int destination, Zip_Invoice
               *invoice, int (*func)());
g1_pack_prefix(ZIP_MAILER *mailer,
               Zip_Invoice *invoice,
               int (*func)(), int scan_direction);
```

The following example demonstrates the use of collective operations on an invoice containing an integer and a double.

```
int i;
double d;

zip_new_invoice(&invoice, "%i%d", &i, &d);

g1_pack_fanout(mailer, invoice, 0);
g1_pack_combine(mailer, invoice, ZIP_ADD);
g1_pack_fanin(mailer, 0, invoice, ZIP_MIN);
g1_pack_prefix(mailer, invoice, ZIP_ADD,
               ZIP_FORWARD);
```

3.4 Pack and Unpack Primitives

Several operations are provided to unpack a buffer directly. These are used in defining user-defined collective operations. In addition these operations provide some capabilities that are not present in pack-and-send type operations. Using these operations and the buffered *Zipcode* primitives, it is possible to pack/unpack a variable based on a type flag in the message itself. For example, an application might want to receive a message containing either shorts or integers. By passing the type in the message, one invoice could unpack the type (short or integer) and another invoice could extract the appropriate data. The skip space option in invoice construction can be used to facilitate this type of operation. However, this feature disallows some of optimizations that can normally be done with pack-and-send primitives, so it should be used only when needed.

```
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()`, here we pass a pointer to the communication space, not a pointer to a pointer. The communication space must be freed by the caller after it is unpacked.

The following is an example of how the pack/unpack primitives would be used. This example is a user-defined operation for a collective operation. It sums the elements of a simple matrix.

```
int add_matrices(buf_a, buf_b, extra)
char *buf_a, *buf_b;
Extra *extra;
{
    matrix *a *b;
    Zip_invoice *invoice_a, *invoice_b;

    zip_new_invoice(&invoice_a,
                   "%0matrix", &a);
    zip_new_invoice(&invoice_b,
                   "%0matrix", &b);

    Zip_unpack(invoice_a, ZIP_BUFFER, buf_a);
    Zip_unpack(invoice_b, ZIP_BUFFER, buf_b);

    for(i = 0; i < a -> m; i++)
        for( j = 0; j < a -> n; j++)
            a -> array[i][j] += b -> array[i][j];

    length = Zip_sizeof_invoice(invoice_a);
    Zip_pack(invoice_a, ZIP_BUFFER, &buf_a,
             length);

    zip_free_invoice(invoice_a);
    zip_free_invoice(invoice_b);
```

}

At times it is 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 the example above.

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

4 Optimization

From the outset of the design of the invoice construct, the idea of achieving optimization was a goal. With the advent of scatter-gather capabilities, *Zipcode* needed to be evolved to enable their use. For this reason invoices were constructed with the idea the users should indicate what they want rather than how it should be implemented. This methodology will enable *Zipcode* to take advantage of special hardware features on platforms that have them. Importantly, when a code is moved to a system that does not have special features (e.g. a purely message-passing system), *Zipcode* will compile down to pure message-passing. This multi-faceted approach to implementing *Zipcode* follows its original design philosophy. 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, which, if used directly in applications, would prove unportable, unmanageable, or too low-level. By using a high level approach *Zipcode* can take advantage of these features without users modifying their software. Low-level notations cannot hope to achieve this type of optimization, because they do not expose enough semantic information in their instructions.

The following are examples of the kinds of optimization strategies which were envisioned. For point-to-point communications different strategies could be employed for different sized messages. For example, on a Cray T3D an implementation might take advantage of light-weight shared memory primitives for small messages while invoking the block-transfer engine for larger messages. Many vendors are providing network capabilities to perform collective reductions and broadcasts. *Zipcode* implementations could make heuristic runtime decisions on when to use such features.

With vendor implementations of MPI, lighter-weight context support will become a reality, and *Zip-*

Figure 3: Comparison of point-to-point communication

Figure 4: Comparison of point-to-point communication overlaid

code-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 will provide portability and performance to our *Multicomputer Toolbox* libraries. Therefore, no *Zipcode*-based application or library will suffer in face of the standardization, but will instead benefit almost immediately by it.

5 Timing Results

In order to determine what overhead was added by this higher level interface, a comparison of sev-

Figure 5: Comparison of combine

eral operations was done with the old style *Zipcode* calls. The current implementation, as was indicated earlier, was layered on top of existing *Zipcode* primitives. This implies that the results shown should be considered worse-case; a more sophisticated implementation should never perform worse, and, in most cases, perform significantly better. The tests were run on an nCUBE/2 and use a flat communication buffer scheme. For point-to-point communication timings a "round-trip" time was taken for sending a message that contained a double array. This involved packing the communications buffer, sending to an adjacent node, unpacking on that node, repacking, sending back to the initial node and unpacking the message. This represent moving data from user-space to user-space which is what an application or library writer cares about. For the non-overlaid test a copy was done in both cases; in the invoice case this is done by the `pack_send()` routine while in the non-invoice case it was done manually. All of the results presented here are of relative differences between the invoice method and the old buffered method.

The results for the for the non-overlaid test are shown in Figure 3. The overhead in this test is constant and becomes insignificant for large messages. For the overlaid test only pointer motion is needed (no copy is made). The results of this test are shown in Figure 4. The overhead in this test was not constant (it grows with the size of the message being passed). This is the result of an extra memory allocation done in the invoice version that can be avoided when the user has direct control of the communications buffer. The final timing result is for a combine. In the invoice case the built-in addition operation is used while in

the non-invoice case a hand-written addition routine is used. The differences (see Figure 5) arise from extra memory allocation and deallocation, due to the implementation. If the invoice routines were not layered on top of existing *Zipcode* primitives, the extra memory management could be eliminated and the overhead would be constant yielding better performance.

6 Conclusions

The invoice mechanism described in this paper is currently in use in a ground water flow application. The use of invoices decreased the development time by automating the data motion tasks associated with a communications buffer implementation. The user-defined type feature proved valuable in isolating code changes when structures were modified, thereby increasing productivity. In some sense the user-defined types enable an object-oriented approach by defining communication member functions for data-objects. The real impact on performance is hard to determine since there is no identical code to compare against but we see performance that is on-par with codes that have similar computation to communication ratios.

We see the notational abstraction of invoices as helpful in dealing with heterogeneity in multicomputers and distributed computers. Abstraction is also 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 tagged message systems whenever vendors make low-level access to their hardware and system calls available to us during our implementation phase.

Acknowledgements

We thank Nathan E. Doss, of the Engineering Research Center for Computational Field Simulation, for help with on-going improvements to the latest release of *Zipcode* (version 1.00). The authors also wish to acknowledge the MPCRL at Sandia National Laboratories for access to their 1,024-node nCUBE/2.

References

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

- [2] Scott Berryman, James Cownie, Jack Dongarra, Al Geist, Bill Gropp, Rolf Hempel, Bob Knighten, Rusty Lusk, Steve Otto, Tony Skjellum, Marc Snir, David Walker, and Steve Zenith. Draft document of the MPI standard. Available on `netlib`, May 1993.
- [3] Anthony Skjellum. Scalable Libraries in a Heterogeneous Environment. In G. Fox et al, editor, *High Performance Distributed Computing II (HPDC2)*, pages 13–20. IEEE Press, July 1993.
- [4] 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.
- [5] 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*.
- [6] 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.

DATE

FILMED

8/16/94

END