Zipcode: A Portable Multicomputer Communication Library atop the Reactive Kernel

A. Skjellum
A.P. Leung
M. Moriari

CRPC-TR90029
1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Zipcode:
A Portable Multicomputer Communication Library atop the *Reactive Kernel*

CRPC-90-3

Anthony Skjellum* Alvin P. Leung
Manfred Morari

California Institute of Technology
Chemical Engineering; mail code 210-41
Pasadena, California 91125

Abstract

Sophisticated multicomputer applications require efficient, flexible, convenient underlying communication primitives. In the work described here, *Zipcode*, a new, portable communication library, has been designed, developed, articulated and evaluated. The primary goals were: high efficiency compared to lowest-level primitives, user-definable message receipt selectivity, as well as abstraction of collections of processes and message selectivity to allow multiple, independently conceived libraries to work together without conflict.

Zipcode works atop the Caltech *Reactive Kernel*, a portable, minimalistic multicomputer node operating system. Presently, the *Reactive Kernel* is implemented for Intel iPSC/1, iPSC/2, and Symult s2010 multicomputers and emulated on shared-memory computers as well as networks of Sun workstations. Consequently, *Zipcode* addresses an equally wide audience, and can plausibly be run in other environments.

*Present address: Lawrence Livermore National Laboratory, Numerical Mathematics Group, 7000 East Avenue, L-316, PO Box 808, Livermore, CA 94551. e-mail: *tony@helios.llnl.gov.* (415)422-1161. FAX: (415)423-2993.

Introduction

Wide experience with first-generation point-to-point multicomputer node operating systems (such as Intel's NX) demonstrates the inadequacy of basic typed message systems for large applications. That is, simple message typing does not provide enough degrees-of-freedom or notational elegance in message receipt selectivity for most situations. As is widely implemented in practical codes, an additional (typically one-shot) message-passing layer and queueing mechanism cover the naked primitives, providing additional flexibility at the application level. The overhead of an application-oriented layer can be made acceptably light, as we indicate below. However, the overheads associated with the underlying typed primitives are viewed excessive in that little or no value is attributable to the hard-wired typing provided by the node operating system itself.

The Caltech *Reactive Kernel* (*RK*), by Seitz and co-workers, was designed with this theme in mind [5,6,9]. These primitives provide *no* message typing at all; they are of high-efficiency, but too low-level for direct application use. For determinism, pairwise message ordering is preserved. Multiple processes per node are supported, with correctness independent of process placement, subject to finite storage limitations. There is no intra-node shared memory. Finally, no explicit notion of the underlying communication network is enforced on the application (*e.g.,* binary n-cube-oriented limitations/strategies); process placement remains, however, at the discretion of the application.

Application-oriented layers are created to specialize and abstract from the *RK* level on a case-by-case basis; the layers' functionality and, hence, overhead are chosen by the application programmer as part of the overall software design process. The easy-to-understand, concise set of primitives in *RK* is easily ported and, alternatively, readily emulated. Consequently, applications based on *RK* stand an excellent chance of surviving changes of node architecture and communication network. Furthermore, as discussed below, these primitives provide a rational basis for programming medium-grain, shared-memory multiprocessors as well.

Unfortunately, individuality in design of the application message-passing layer leads not only to repetition of effort but also to portability problems between programmers and projects, just as

incompatible vendor operating systems do between diverse multicomputers. These effects are fundamentally unacceptable, because we intend to create high-performance, portable multicomputer codes with potentially long lifetimes. Furthermore, we want to create substantial libraries that can be used together in a single program without the chance of message-passing conflicts because of differing assumptions between those libraries, or with/within the application code itself. Consequently, it is desirable to define a single, encompassing application message-passing layer with high efficiency, portability and extensibility, that will be used well by a wide range of applications. These are the main goals of *Zipcode*.

The *Zipcode* philosophy is as follows. First, only the application can properly define the nature, style, and extent of message-passing receipt selectivity. There are arbitrarily many such patterns of selectivity – they cannot be foreseen or implemented by the node operating system *a priori*. Consequently, any node operating system that *types* messages is, in general, too restrictive, molds message-passing style and notation unnecessarily, and imposes overhead to overcome such built-in restrictions. Second, there may be arbitrarily many contexts of communication within a given multicomputer application which, for correctness, cannot clash; no node operating system of which we are presently aware supports multiple contexts. Third, the best node operating system is the one that constrains the application least, both in function and overhead. Thus far, *RK* has proven the most elegant underpinning because it imposes essentially no arbitrary restrictions on the communication process, and is not ridden with features of dubious value but noticable cost.

Zipcode design features can be summarized as follows:

- Operates on process lists as the fundamental communication object with no predilection toward hypercubes, gray codes, or powers of two.
- Uses message classes to decide how process lists are to be abstracted.
- Uses message contexts to decide in part on receipt selectivity.
- Uses message classes to decide in part on receipt selectivity.
- Inheritance techniques are used to derive additional message contexts.
- Five standard pre-defined message classes are provided, including grids.
- "Global operations" – *combine* and *broadcast* – are defined for several of the standard message classes and are extensible to new classes.

- C macros are applied widely to avoid excess overheads.
- Message-debugging capabilities are inherent in message classes.
- The number of classes and contexts are definable and extensible at run time.
- Applications can set the current context and utilize terse, readable program notation for message transmissions.

In our empirical experience, carefully coded *ad hoc* message layers imply a 10-15% overhead in message startup cost compared to bare primitives. We observe comparable overheads for the *Zipcode* system (about 20%). As a function of its design, rejection of a message during message selection is nearly as cheap as in *ad hoc* layers. Message acceptance cost is, however, a function of the complexity of the message class being requested. Queueing of received, undelivered messages is discussed; the present use of a single queue is justified and alternatives are mentioned.

Zipcode has been run extensively on the Intel iPSC/2 and Symult s2010 systems, and on networks of Sun workstations. Performance results (single message transmissions and global operations) are quoted as a function of message length for the Symult s2010 implementation.

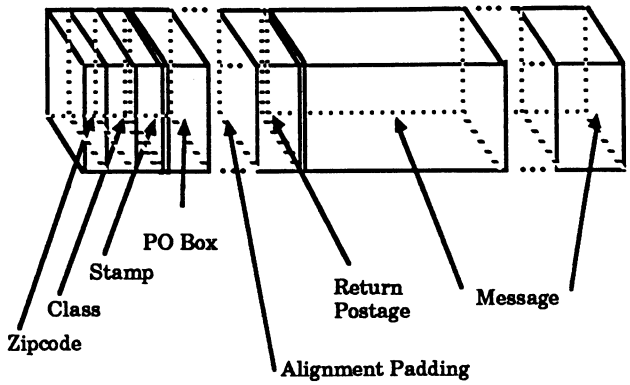
Nearly 60,000 lines of successful application code have already been developed relying on *Zipcode*. Use of the layer as a pedestal for portable scientific/engineering numerical tools is in progress. Thoughts on this and future planned improvements are mentioned in closing.

Design Discussion

In second-generation multicomputers, improvements in routing technology allow programmers sensibly to ignore the underlying communication network and conceive of the computers as nodes on a completely connected graph with uniform transmission costs. As Athas and Seitz point out [1], this approximation holds well for small- to medium-sized multicomputers employing their cut-through, wormhole routing technology. Figure 1. illustrates performance of application-level primitives on the Symult s2010, which incorporates this routing technology.

Within the loose framework of communicating sequential processes [2], two programming paradigms

Figure 0. Schematic of a *Zipcode* Letter.



Letter:	*Zipcode* format of a message
Zipcode:	Integer indicating the letter context
Class:	Integer indicating style of selectivity
PO Box:	Class-dependent structure for selectivity
Envelope:	Preamble structure containing PO Box, class, zipcode
Cover:	The envelope with alignment padding
Stamp:	Length in bytes of the cover

are commonly used: reactive programming, where processes progress asynchronously with computational decisions driven by the number and variety of messages received, and loosely synchronous programming, where processes progress with intermittent, pre-specified synchronizations. *Zipcode* supports both styles of programming by building on unblocked and blocked *RK* primitives, respectively.

Type *vs.* Class *vs.* Context

A message class is a set of rules and a data specification used for defining message receipt selectivity, and for discriminating correctly among incoming messages. A hypothetical class of messages (call it 'A') might be "messages chosen based on their source, where the source is to be specified by node number and process ID." Given this message class, it's possible to look for a message from one or more acceptable sources, rejecting all others to a queue for future retrieval. For example, we could request "the next message from (node 1, process 0)" or, equally well, "the next message

from (node 1 or 2, process 0)." That we can discriminate on source implies that the message must include, however transparently, its source information: in this case, two integers. In *Zipcode*, we call this the "PO Box" data.

The message class just discussed would *not* allow discrimination based on the particular aspect of the process that sent a message, nor on the particular contents of a message. These possible deficiencies can be handled in distinct ways. On one hand, we could define a more powerful class (denote 'B'), increasing the contents of its PO Box compared to the 'A' class: "messages chosen based on their source, plus an integer type." Given such a class, messages could be tagged appropriately by the sender to indicate their contents and/or intended use. If we really want to indicate the contents of the message by type, this is probably the most convenient approach. If, however, analogous parts of the communicating processes produce messages that they want to keep exclusively among themselves, addressing their messages in a narrow sense, it is more convenient to define a message context. We call the integer that specifies context the "zipcode" because it states conceptually "where" the message is to go within its destination process(es), but not in detail.

A message context is like a message type, but stronger – knowing the context implies knowing who can participate in the transmission process. So, for example, we could pose receipt selectivity as "Accept a class 'B' message from (node 0, process 0) in context 6 (or zipcode 6)," where "6" indicates the specific phase of the computation for which the message is intended (such as a linear-algebra subroutine operating on a set of related matrices using processes in a particular logical configuration). So far, context is just an extra integer added for greater flexibility. However, it leads immediately to further interesting capabilities. As stated, being part of a message context implies knowing the participants: in the simplest instance, an explicit list of the participating processes. A message class can specify indentifying information in PO Boxes in a number of ways, and we could imagine altering the semantics of the PO Box to exploit this extra information. First, we could assign an abstract name to each process in the process list. A class 'A' message could be changed to have its receipt selectivity be "messages specified by their context and position (index) of the source process in that context's process list." A request could be "accept an 'A' class message in context 6 from abstract process name 30." Once we abstract the basis of receipt selectivity,

context and class information together uniquely identify the message(s) we want to accept; each is insufficient alone.

We need a terse, flexible notation, and message structure to permit multiple contexts and classes to work together. Figure 0. illustrates the structure of a *Zipcode* "letter" – a message, plus enabling information: the variable-length envelope/cover including its zipcode, PO Box, and other needed structural data. The postal analogy in *Zipcode* carries quite far because a process creates and mails a letter, first by grabbing and filling out a blank message, then by addressing its envelope, and finally, by posting the entire object. Starting from a list of addressees, a class, and a zipcode context, a canonical data object, a *mailer*, is constructed by *Zipcode* calls. A mailer is the object used when creating, receiving, or posting letters within the system. From it, further contexts of communication may be created via inheritance routines (by correctly deriving a communicating subset of processes and making a new process list for them).

"No Class" Systems

Typical node operating systems are "no class" systems. Specifically, they are systems where the only explicit *class* is "messages identified by a single integer," and *types* are instantiations of that integer. Types are most often bound at compile time by applications, and diverse applications usually attach distinct semantic connotations to the same integer types, implying source-level conflicts. Furthermore, all messages are in the same context, so there is no way to distinguish messages intended for one phase of a process over another, to avoid such conflicts, except by the types themselves.

Broadcast and combine operations require extension of typing for their deterministic implementation. It's necessary to discriminate among messages based on their source. Consequently, typed message systems must include extra header information invisible to the user, at least in those messages destined to participate in a global operation – multiple classes, though invisible and inaccessible, play a role even in these systems.

Reactive Kernel Primitives

For the purpose of this discussion, we need to define six of the *RK* primitives, fitting neatly into two categories: message-generating (*i.e.*, allocate, receive) and message-consuming (*i.e.*, free, send), as follows:

```
char *msg;
int length, node, pid;
int count, *proc_list;
```

Message-Generating Primitives:

```
msg = xmalloc(length);
msg = xrecv();     /* unblocked */
msg = xrecvb();    /* blocked */
```

Message-Consuming Primitives:

```
xsend(msg, node, pid);
xmsend(msg, count, proc_list);
xfree(msg);
```

Basically, messages are created by `xmalloc()`, sent via `xsend()` or `xmsend()` (multiple destinations), and received via `xrecv()` (unblocked) or `xrecvb()` (blocked). Sending a message is equivalent to an `xfree()` with the side-effect that the message is mailed to the specified destination(s). This represents the complete message-passing notation of *RK*.

Zipcode Class-Independent Calls

Zipcode maintains the same basic naming convention and style as *RK*. For all classes, the same calls are used for allocation, sending and receiving letters. Specific classes may define additional

calls to increase the convenience of use (see G2-Class calls further below). Small-y calls require specification of the mailer relative to which a letter is to be created, sent or received. Big-Y calls depend on the current mailer context established by `Ypush()/Ypop()` calls. As such, they omit mailer arguments. The `[yY]mail()` calls transmit to all addressees of a mailer.

```
char *letter;
ZIP_MAILER *mailer;
```

Context-setting Primitives:

```
Ypush(mailer);
Ypop();
```

Letter-Generating Primitives:

```
letter = ymalloc(mailer, length);
letter = yrecv(mailer);   /* unblocked */
letter = yrecvb(mailer);  /* blocked */

letter = Ymalloc(length);
letter = Yrecv();       /* unblocked */
letter = Yrecvb();      /* blocked */
```

Letter-Consuming Primitives:

```
ysend(mailer, letter, node, pid);
ymsend(mailer, letter, count, proc_list);
yfree(letter);

Ysend(letter, node, pid);
```

```
Ymsend(letter, count, proc_list);
Yfree(letter);
```

Abstraction to process-list addressees:

```
ymail(mailer, letter);
Ymail(letter);
```

Variations of the basic `[yY]send()` and `[yY]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. `[yY]mail()` applied to appropriately inherited child mailers, allows specification of arbitrary, user-defined subsets of recipients of the original mailer's addressees.

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.

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, places 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; /* mailer pointer */
```

```
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 */
short int *zipcode; /* zipcode, if known */
ZIP_MAILER *(**inherit)(); /* overrides for inheritance */

mailer = yopen(class, addr, extra, parent, copyflg, zipcode, inherit);
```

Typical call:

```
mailer = yopen(class, addr, NULL, NULL, NULL, NULL, NULL);
```

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 cannot be 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.

G1-Class mail is a 1D-grid-abstraction class, similar to Z-Class mail.

G2-Class mail is a 2D-grid-abstraction class. A PxQ 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 [7].

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 y- and Y-type calls defined generically above.

```
int p, q; /* source or destination */
```

Letter-Generating Primitives:

```
letter = g2Recv(mailer,p, q); /* unblocked */
letter = g2Recvb(mailer, p, q); /* blocked */

letter = G2Recv(p, q);    /* unblocked */
letter = G2Recvb(p, q);   /* blocked */
```

Letter-Consuming Primitives:

```
g2Send(mailer, letter, p, q);
G2Send(letter, p, q);
```

Global operations *combine* and *broadcast* (fanout) are defined and have been highly tuned for this class. Combines are over arbitrary associative-commutative operators specified by `(*comb_fn)()`. Broadcasts share data of arbitrary length, assuming all participants know the source:

```
void (*comb_fn)(); /* operation */
void *buffer;      /* data/result */
int size, items;   /* data specifications */
```

```
g2_combine(mailer, buffer, comb_fn, size, items);
G2_combine(buffer, comb_fn, size, items);

void *data;         /* data/result */
int length;         /* length of data */
int orig_p, orig_q; /* origin */

g2_fanout(mailer, &data, &length, orig_p, orig_q);
G2_fanout(&data, &length, orig_p, orig_q);
```

G2-Grid mailer creation:

```
int P, Q; /* grid shape */

mailer = g2_grid_open(P, Q, addr, zipcode);
```

A much more general version, `_g2_grid_open()`, (analogous to `yopen()`) is also available.

G3-Class mail is a 3D-grid-abstraction class. A $PxQxR$ grid naming abstraction is attached to the process list, analogously to the G2-Class 2D-grid primitives. This class should prove very useful in defining operations such as matrix-matrix multiplications in an unrestrictive setting.

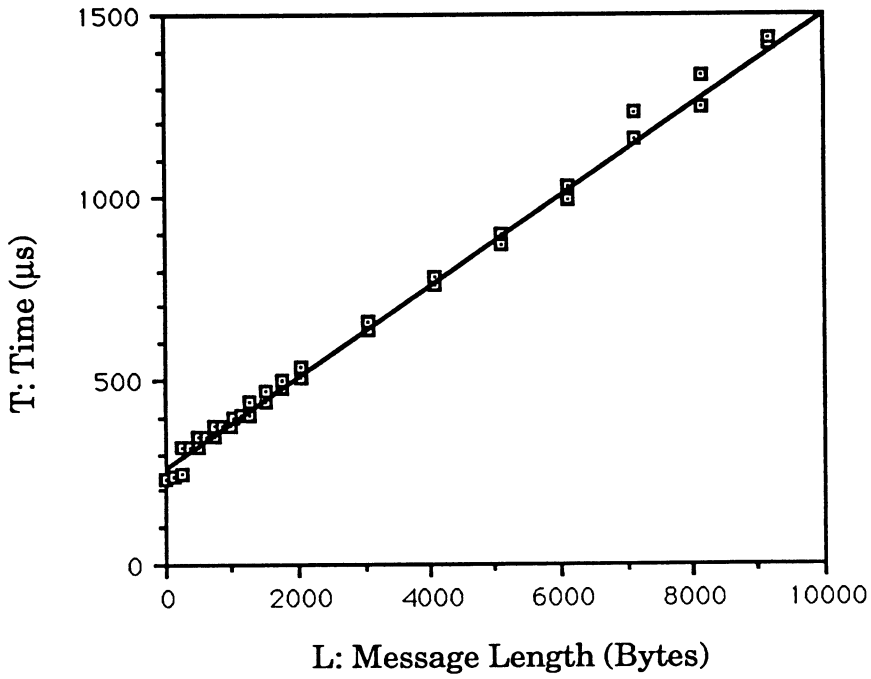
The *Zipcode* Queue

Message selectivity implies that some messages will have to be stored on a queue that the *Zipcode* layer must maintain; there is no push-back mechanism in *RK*. In our experience, multicomputer codes do not accumulate very many messages on the queue; typically not more than five. We have therefore chosen the simplest possible queueing arrangement: a linked list with linear access from oldest to newest. Hashing by zipcode and/or class could also be implemented, but thus far appears superfluous.

Performance

We quantify performance in three categories: single transmission timings, broadcast operations, and combine operations, which we consider in turn. For each case, we have restricted our attention to lengths that are even, to avoid severe penalities from data copying (*i.e.*, `bcopy()`) operations that are incurred for odd-length messages.

Figure 1. Graph of 2D-Grid Primitive transmission timings on a 16-node Symult s2010.



A fit yields: $T = 260.25 + 0.12660L\mu s$, where T is time in μs, and L is the message length in bytes. An underlying *RK* transmission costs approximately $T = 220.0 + 0.1L\mu s$, unoptimized (*vs.* $T = 200.0 + 0.1L\mu s$ optimized).

Single Transmissions

Single-transmission performance is measured using a quiescent ensemble, through which a single *Zipcode* letter is passed around many, many times among random destinations. The performance

illustrated in Figure 1. is for a 16-node machine, where G2-Class 2D-grid primitives were employed. There is a stair-stepping cost increase as a function of length. This is expected because of 256-byte pages used by *RK* to pass messages. Based on a least-squares fit of the data, we conclude that a reasonably conservative measure for the startup cost of G2-Class primitives is $260.25\mu s$ compared to about $220\mu s$ for the bare *RK* primitives. With optimized compilation, *RK* startup time drops to about $200\mu s$; this savings would be reflected directly in reduced *Zipcode* startup time. Furthermore, no optimizations, either by register keyword usage or optimized compilation have yet been employed on the *Zipcode* layer. Such optimizations are expected further to improve performance, perhaps as much as $10\mu s$ for the Symult implementation.

From this performance, we can estimate the systemic granularity of the Symult s2010, at the application level. Defining the granularity as T_{comm}/T_{calc}, we report $T_{comm}/T_{calc} \approx 46$, with $T_{comm} = 260.25\mu s$, and $T_{calc} \equiv 5.57\mu s$. T_{calc} is the highly optimized time for the double-precision floating point operation $a = a + b * c$ (*vs.* $13.785\mu s$, unoptimized). See also [7].

Global Operations

There are two global operations *broadcast* (fanout) and *combine* (recursive doubling) [8]. They are extensible to all classes whose receipt selectivity includes source information.

Broadcast is a one-to-all concurrent fanout operation. This has been implemented so that the originating process sends $\lceil \log_2 N \rceil$ letters, for N participants; completion is in $\lceil \log_2 N \rceil$ time. Other tree approaches are possible, and have potential merit in load-balancing situations. The key feature of *broadcast* is its lesser performance penalty for non-powers of two vis a vis *combine*, so it should be used whereever possible. Figures 2a., 2b., illustrate performance for the G2-Class 2D-grid primitives. They are only slightly cheaper than the *combine* primitives (Figure 3.) for powers of two. For non-powers of two, the difference is more dramatic (see [7] for further discussion). A least-squares fit of the timing data for lengths from $4 \ldots 10,084$, representative of performance for all nodes counts from $N = 2 \ldots 128$, is $T = (4.1926 \times 10^2 + 4.0138 \times 10^{-1} L)\log_2 N + (3.5611 \times$

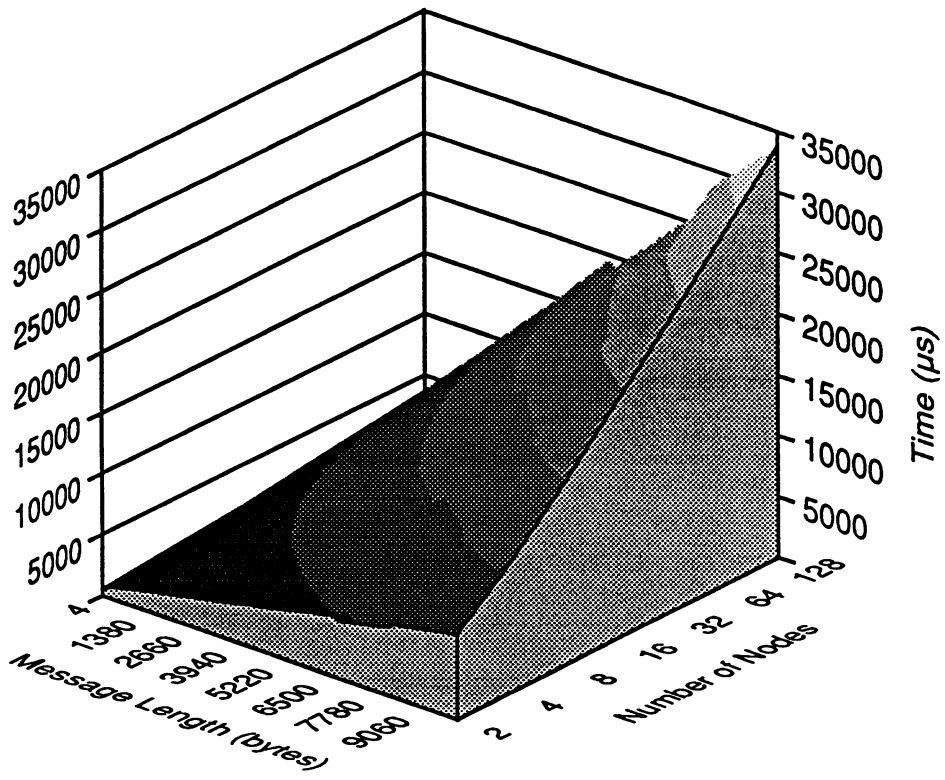


Figure 2a. Graph of 2D-Grid Broadcast Primitive Timings on a Symult s2010.

$10^2 + 1.4140 \times 10^{-1} L)\mu s$ where T is the time in μs, and L is the length in bytes. Finally, a linear transmission regime for small N has been implemented but is not reflected here. It produces lower overhead when $N \leq 4$.

Combine is the usual associative-commutative global operation, completed in logarithmic time in the number of processes. Figure 3. illustrates performance for powers of two, for the G2-Class 2D-grid primitive case. Non-powers of two are substantially more expensive; in the worst case, roughly twice the cost of *combine* for the next highest power of two. A least-squares fit of the timing data for lengths from $4 \ldots 10,084$, valid for power-of-two nodes $N = 2 \ldots 128$, is

$T = (6.0766 \times 10^2 + 4.3976 \times 10^{-1} L)\log_2 N + (2.9994 \times 10^2 + 2.7555 \times 10^{-1} L)\mu s$ where, again, T is the time in μs, and L is the length in bytes.

Both the *broadcast* and *combine* primitives exemplify the high-frequency stepping characteristic, which results from the 256-byte pages used for message transmission by *RK*. At each page boundary, a small additional startup cost is incurred. Furthermore, both operations illustrate a "trough" of improved performance, beginning at lengths somewhat beyond 5,000 bytes, and ending at roughly 8,192 bytes. This trough is thought to be a memory-allocation effect within *RK*; memory pages are managed and dispensed at the lowest level in 8,192 byte (8K) pages.

"Virtual Distributed Memory"

In some circles, it's popular to try to hide distributed memory characteristics by introducing a notion of "global virtual shared memory" that constructs, in principle, a shared-memory paradigm for multicomputing. This follows the tacit assumption that multicomputers are hard to program, while multiprocessors are easy to program, and that shared-memory ideas should be spread to the multicomputer regime insofar as possible, thereby reducing the effort inherent in multicomputer computation. Lacking evidence to suggest efficient realizations of this scheme are possible, we suggest the diametric opposite – "virtual distributed memory." We consider the distributed-memory paradigm to be the more practical model for concurrent computation on medium-grain multiple-instruction, multiple-data multicomputers and multiprocessors alike. We define uniform message-passing primitives for multicomputers and multiprocessors, and achieve portability and high performance for both classes of machines, encapsulating any special features of the memory hierarchy in higher-level data distributions. Data distribution is handled at the application-level, rather than directly and unportably in the communications layer. Applications are written for correctness independent of data distribution, with performance depending heavily on the appropriate data-distribution(s) (*e.g.*, scatter distribution *vs.* linear distribution in multicomputer linear algebra computations). The effects of locality of data are still left as tuning parameters for the

application programmer, but systematically so.

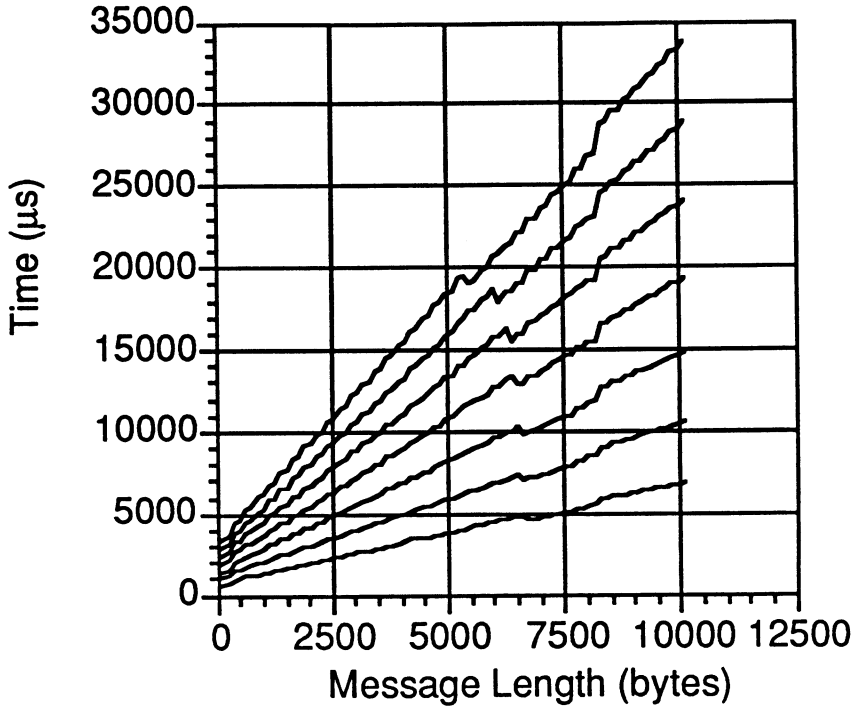
We contend that this approach not only promotes portability, but also rationalizes medium-grain multiprocessor programming, while promoting modular, object-oriented algorithms. Instead of hiding bottlenecks and unscalabilities in the form of shared-memory hotspots and critical sections, the "virtual distributed memory" approach – multiprocessor support for communicating sequential processes – makes explicit the synchronizations, and data dependencies that render multiprocessor code quite challenging to debug or extend to many processors, if not to develop at the outset.

RK ports readily to multiprocessor environments or can be emulated. We are aware of a six-processor Sequent Symmetry implementation by Hamrén and Mattisson, achieving message-startup times of $250\mu s$, competitive with the Symult s2010 multicomputer at roughly $200 - 220\mu s$ [3, 5]; they indicate no explicit per-byte message transmission costs because global memory pointers are used to emulate message passing. This performance results with one process per processor, with much lower performance evident with multiple processes per processor. The Sequent *RK* implementation is based on Unix System V shared-memory primitives and should itself port to other archetypical multiprocessors (*e.g.*, BBN Butterflies, multi-headed Crays). Given the *RK* underpinning, *Zipcode* and the whole body of *Zipcode*-compatible codes port immediately to such multiprocessor environments also. A full discussion of this class of implementations with ported *RK* / *Zipcode* performance will be addressed in a future paper.

Conclusions, Future Work

In typical multicomputer programs, a layer of communication primitives is constructed above those provided by the operating system. Early point-to-point node operating systems, such as Intel's NX, pre-defined the style and abstraction of message typing. (The decision that messages are typed per se is already a strong assumption.) Consequently, application programs were forced either to conform to the pre-defined style, or to ignore the typing feature, and add additional typing overhead of their own. The Caltech *Reactive Kernel* (*RK*) was designed with this experience in mind, and

Figure 2b. Graph of 2D-Grid Broadcast Primitive Timings on a Symult s2010.



Times quoted for 2, 4, 8, 16, 32, 64, and 128 node configurations. Linear-linear graph exemplifies low- and high-frequency behavior.

overcomes the design flaw simply by omitting low-level typing altogether. *RK* consequently presents a set of message primitives that must be augmented for any non-trivial application. Application programs define *ad hoc* extensions to pattern message passing according to their needs, yet such layers often imply incompatibility between any two application programs or subroutine libraries. The key design principle underlying *Zipcode* is that a single, extensible layer above *RK* is suitable for the vast majority of multicomputer applications, thereby avoiding fundamental incompatibilities before they arise, and also eliminating duplication of effort in application-level message-passing design.

We foresee *RK* as the low-level portability standard for multicomputers and multiprocessors in the 1990's, much as Unix is projected to become the operating system standard of 1990's personal com-

puters, workstations and supercomputers alike. The flexible features of *Zipcode* make it a suitable basis for many application codes and libraries, promoting both portability, and codes of complexity whereever *RK* is implemented or emulated. *Zipcode*, as a portability pedestal for multicomputer applications, encapsulates the interprocessor hardware characteristics, while encouraging the development of codes whose correctness is independent of data distribution. Data distributions can subsequently be used to tune for high performance in a hardware- and application-conscious way.

The key features of *Zipcode* are: its design for extensibility, allowing the definition of many classes of communication and hence message receipt selectivity; support for abstraction of process lists into convenient working groups for communication; the ability to define many non-interfering communication contexts based on process lists with instantiation at runtime rather than compile-time; and the derivation of additional communication contexts through inheritance. Use of *Zipcode* implies acceptable overhead compared to the pervasive one-shot message-passing layers of most multicomputer applications. We asserted at the outset of this work that message-passing generality could be achieved with very little additional overhead compared to one-shot layers. This has subsequently been achieved in *Zipcode*.

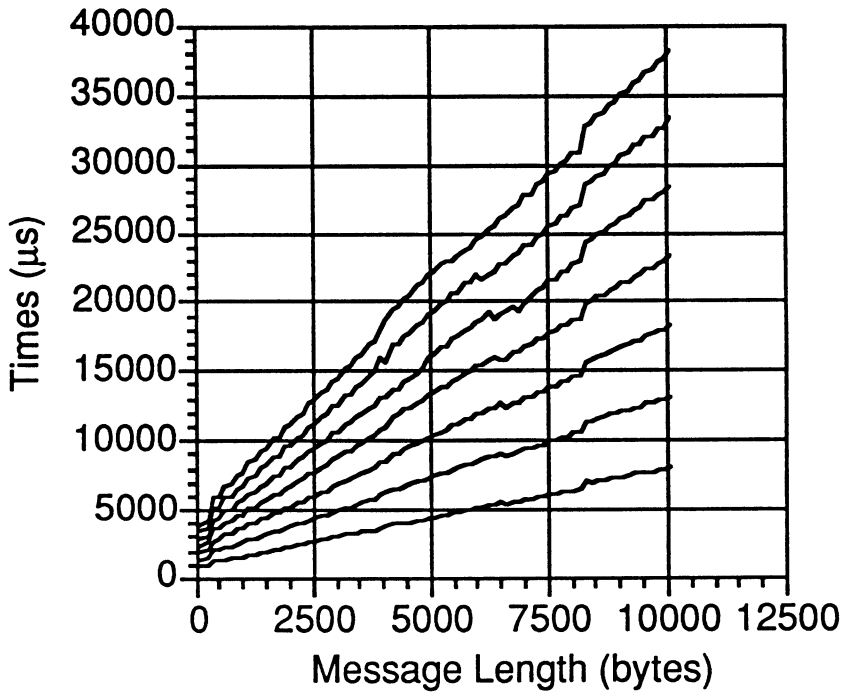
For the future, we foresee several classes of improvements and a wider range of implementations, both for new and extant multicomputers, and for medium-grain multiprocessors, as noted above. We foresee the creation of a slightly more extensive pool of general-purpose message classes, based on user feedback. We expect to extend grid-based primitives to provide grid-to-grid data transformations. In the area of debugging, we intend more dramatic growth. We expect to introduce more sophisticated macros and function calls to allow for automated detection of many communication-related errors, as well as better monitoring of the *Zipcode* queue. We do not plan to replace the queueing mechanism at present, but we do expect to make small definitional changes to allow the queueing mechanism to be application re-defined.

Experience with *Zipcode* suggests ways to extend *RK* for overall higher performance of the application. In particular, implementation of *broadcast* and *combine* by *RK* can be posed in a completely general way, consistent with its unrestrictive philosophy; however, such implementations could take

advantage of important hardware optimizations and produce much faster primitives overall. The extant *Zipcode* calls would layer transparently above such new primitives (see [7]).

A numerical toolbox consisting of *Zipcode*-based applications is under construction and refinement. The advantages of the *Zipcode* basis will include portability and compatibility between a number of numerical libraries from several sources, working primarily, at present, with G2-Class 2D-grid primitives. This too will be the subject of a future paper.

Figure 3. Graph of 2D-Grid Combine Primitive Timings on a Symult s2010.



Times quoted for 2, 4, 8, 16, 32, 64, and 128 node configurations.

Acknowledgements

The authors wish to acknowledge both the initial and recent suggestions of Lena Peterson and Sven Mattisson, as well as early input from Eric Van de Velde, all of whose previous experiences with and examples of "one-shot" message-passing layers provided valuable motivations for this work. We wish also to acknowledge Wen-King Su for his insightful advice over the last eightteen months, and Prof. Charles L. Seitz who offered helpful suggestions and encouragement.

The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050. The second author (then at the University of California, Santa Cruz; presently at Syracuse University) received support for his 1989 Caltech Summer Undergraduate Research Fellowship (SURF) under the same grants, and wishes to thank the Caltech SURF program for the opportunity to pursue the research discussed in part here.

Zipcode was developed using machine resources made available by the Caltech Computer Science Submicron System Architecture Project and the Caltech Concurrent Supercomputer Facilities (CCSF). Caltech/Rice CRPC provides support, in turn, to CCSF.

This paper, as a student contribution, won the student-contest prize for best contribution in the "operating systems" area at the *Fifth Distributed Memory Computing Conference* (DMCC5), Charleston, April 1990.

References

[1] Athas W. C., and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, August 1988, pp. 9-24.

[2] Hoare, C. A. R, "Communicating Sequential Processes," *CACM* **21**(8), August 1978, pp. 666-677.

[3] Mattisson, S. — Personal communication, April 1990.

[4] Seitz, C. L, "The Cosmic Cube," *CACM*, **28**(1), January 1985, pp. 22-33.

[5] Seitz, C. L., *et al.*, *The C Programmer's Abbreviated Guide to Multicomputer Programming,* Caltech Computer Science, Report **Caltech-CS-TR-88-1**, January 1988.

[6] Seizovic, J., *The Reactive Kernel*, California Institute of Technology, Computer Science, Report **Caltech-CS-TR-88-10**, 1988.

[7] Skjellum, A., *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Differential-Algebraic Process Systems in Chemical Engineering*, Ph.D. Dissertation, California Institute of Technology, Chemical Engineering, 1990. Report CRPC–90–4.

[8] Stone, H. S., *High-Performance Computer Architecture*, Addison-Wesley, 1987.

[9] Su, Wen-King, *Reactive-Process Programming and Distributed Discrete-Event Simulation*, Ph.D. Dissertation, California Institute of Technology, Computer Science, 1989, Report **Caltech-CS-TR-89-11**.