Integrating Zipcode and PVM: Towards a Higher-Level Message-Passing Environment*

Li-wei H. Lehman
NSF Engineering Research Center for Computational Field Simulation
Mississippi State University
Mississippi State, Mississippi, 39762

December 10, 1993

Abstract

This paper describes the architecture and implementation of an integrated message-passing environment consisting of Zipcode and PVM. Zipcode is a high-level message-passing system for multicomputers and homogeneous networks of computers. PVM is a relatively low-level message-passing system designed for multicomputers and heterogeneous networks of computers. Although PVM provides a workable and easy-to-use message-passing system, it does not have some of the high-level constructs, such as communication contexts, required to develop reliable and scalable parallel libraries and large-scale distributed software. By porting Zipcode's high-level constructs on top of PVM, the integrated environment facilitates existing PVM applications to gradually migrate from a low-level to a high-level message-passing paradigm. In the integrated environment described here, Zipcode is ported on top of PVM via an intermediate layer, the emulated Cosmic Environment/Reactive Kernel (CE/RK) layer. Such an integrated environment allows programmers to utilize high-level Zipcode constructs as well as low-level PVM calls. Implementation details of the CE/RK primitives are presented. Planned future enhancements include performance improvement of the integrated system as well as adding heterogeneous environment support.

1 Introduction

In this paper, we describe the architecture and implementation of an integrated message-passing environment consisting of Zipcode and PVM (Parallel Virtual Machine). Zipcode is a high-level message-passing system designed to support parallel libraries and large-scale distributed software on multicomputers and homogeneous networks of computers. Many of Zipcode's key features also appear in the newly defined MPI draft standard [1, 6, 7]. PVM is a relatively low-level message-passing system designed for multicomputers and heterogeneous networks of computers [2, 4].

Although PVM provides a workable and easy-to-use message-passing system, it does not have some of the high-level constructs, such as communication contexts, required to develop reliable and scalable parallel libraries and large-scale distributed software. By porting Zipcode's high-level constructs on top of PVM, the integrated environment facilitates existing PVM applications to gradually migrate from a low-level to a high-level message-passing paradigm. In the integrated environment described here, Zipcode is ported on top of PVM via an intermediate layer, the emulated Cosmic Environment/Reactive Kernel (CE/RK) layer. Such an integrated environment allows programmers to utilize high-level Zipcode constructs as well as low-level PVM calls.

The integration process can be implemented in three phases. First, the two systems are integrated in a homogeneous environment. Zipcode primitives at the CE/RK layer are implemented using PVM calls. The main concern in this phase is to achieve a smooth integration in which PVM and Zipcode calls can coexist without breaking the integrated message-passing environment. Performance issues are not major concerns in this phase. In the second phase, certain Zipcode primitives will bypass PVM calls and interface directly to the TCP/IP layer in order to improve performance. The final phase involves extending results from previous phases to support a heterogeneous com-

*Revised slightly, September 23, 1994. This work was supported by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University. Technical Report No. MSSU-EIRS-ERC-94-2.

puting environment.

This paper discusses the first-phase integration of Zipcode and PVM, and leaves the second and third phases for future development. It first presents a comparative overview of Zipcode and PVM. It then examines the architecture of the integrated system, and discusses how Zipcode, the CE/RK layer, and PVM work together to form a coherent message-passing and process-management environment. Next, implementation details are presented. Finally, we summarize the results of the integration in our conclusions.

2 A Comparative Overview of Zipcode and PVM

This section gives a comparative overview of Zipcode and PVM. Main features from both systems are examined in the context of a high-level vs. a low-level message-passing environment.

2.1 Zipcode

Zipcode's distinctive features are its support for high-level constructs such as static process groups, communication contexts, and virtual topologies [7, 8]. It supports both point-to-point and collective operations. It also supports a general receipt selectivity scheme based on mail classes. These features are explained in more detail as follows:

- **Static Process Groups.** In Zipcode, process groups are static: once created, they cannot change size. Such a process group is an ordered collection of processes, each of which can be uniquely identified by its rank within a group. When a process group is created, an "addressee list" in the form of (node, pid) pairs is created to store the "addresses" of all members in the group. Static process groups provide stable scopes for implementing safe message-passing via communication contexts. They also define the scope for collective operations.

- **Communication Contexts.** In Zipcode, contexts are provided to restrict the scope of message-passing. They are essential for the development of parallel libraries and large-scale distributed software. Contexts allow each library to operate in its own safe communication space without being interfered with by messages from other libraries.

- **Mailers.** A mailer in Zipcode is equivalent to a communicator in MPI. A mailer data structure is used to form a safe communication context on top of a process group. It contains data such as context-id, a list of all participating members of the mailer, receipt selectivity information, a set of point-to-point message passing methods, and other class-specific data for that mailer. There are several system-defined mail classes, each of which represents an abstraction of a mailer type. Examples of Zipcode mail classes include the Y-class, Z-class, L-class, and the grid classes (G1, G2, and G3).

- **Virtual Topologies.** Virtual topologies are abstract names used to identify processes in a process group. They provide an intuitive way to reference processes using names corresponding to their application-specific roles in a process group. They provide a way to avoid working with hardware dependent names such as (node, pid) pairs. Virtual topologies also appear in MPI.

- **Collective Operations.** Zipcode supports various types of collective operations to perform message-passing among members of a mailer. Examples of collective operations include broadcast (one to all), combine (all to all), and collapse (all to one). These also appear in MPI.

- **Receipt Selectivity Based on Mail Classes.** Zipcode supports various receipt selectivity schemes based on its mail classes. For example, G1, G2, G3, and Z classes have receipt selectivity based on message sources without tags. The Y-class supports a tag-based scheme, whereas the L-class supports receipt selectivity based on both message tags and sources.

2.2 PVM

PVM supports basic process control as well as dynamic process groups [2]. It supports message-passing between a source and one or more destinations. PVM does not support communication contexts to protect message scopes. PVM relies on message tags and sender sources to select messages. In buffer management, it provides typed and untyped pack and unpack operations. PVM supports heterogeneous data format conversion during pack and unpack using XDR. Some of its features are discussed below:

- **Dynamic Process Groups.** In PVM, process groups are dynamic objects with changing participants. A process may join or leave a process group at any time during the computation without informing other processes [2]. Dynamic process groups provide more flexibility in group formation; however, they require extra communication overhead for processes to maintain global group state information. In addition, they discourage the implementation of safe communication contexts based on process groups by introducing unpredictable participating group members.

- **Unique Integer Task Identifiers.** Each PVM process has a unique integer task identifier, called its tid. A tid uniquely identifies a process within a user's entire PVM virtual machine environment. It is assigned by the local PVM daemon process.
- **Collective Operations.** PVM does not have significant support for collective operations [3, 4, 7]. In PVM 3.2, multicast (one to many) operation is supported in addition to the basic point-to-point send and receive. A broadcast operation based on a dynamic process group is supported to broadcast a message to all tasks in a specified group. Furthermore, since all PVM group operations work through the group server, they are slow and unscalable.
- **Receipt Selectivity.** In PVM 2.4, receipt-selectivity is completely tag-based. In PVM 3.2, messages can be selected based on the message tag as well as the sender source. Wild-carding of tags and sender sources is allowed. Even though PVM supports both tag-based and source-based receipt selectivity, messages could still potentially be received out of scope because messages are not protected by communication contexts.

2.3 Conclusions to the Comparative Overview

To conclude this section, Zipcode provides a set of high-level process management constructs and messaging services which PVM does not support. These high-level constructs and services are important in developing parallel libraries and large-scale distributed software [1, 6, 7, 8]. PVM, however, provides a set of low-level process-management and message-passing services upon which Zipcode can be added as a higher layer. Although it will not provide optimal performance, it is possible to port Zipcode on top of PVM, forming an integrated environment where programmers can access services from both systems. As mentioned above, performance issues will be addressed in the next phase integration, which is out of the scope of this paper. This paper mainly concerns itself with how these two systems are put together to form a coherent environment.

3 Architecture of the Integrated System

To integrate the two systems, Zipcode is ported on top of PVM using an intermediate interface layer, that is, the Cosmic Environment/Reactive Kernel (CE/RK) layer. Figure 1 shows a hierarchical view of this layered architecture. The Zipcode layer mainly provides management services of high-level constructs such as static process groups, communication contexts, and virtual topologies. The CE/RK layer provides Zipcode an interface to the lower-level process management and messaging services provided by PVM. It also maintains a HOST/NODE model and performs basic buffer management. One of its most important functions is to ensure coherency between the Zipcode and the PVM environment. PVM is the layer which performs the actual process creation and termination, and send and receive operations. It also directly manages the underlying "virtual machine" resources, including the host configuration information and network socket connections.

3.1 Zipcode and the CE/RK Layer

CE/RK is a lightweight multicomputer nodal operating system with support for message passing services [5, 8]. CE/RK supports a simple HOST/NODE model. A HOST program is the initial process which starts the environment and spawns the NODE processes. In CE/RK, a process in a process group is identified by an ordered (node, pid) pair. The node is the logical node number of the processor where the process is running on. The pid is an integer which uniquely identifies a process on a node. Besides process management services, CE/RK also supports basic buffer management and untyped blocked and unblocked message passing.

CE/RK was designed to be lightweight and portable so that higher level system such as Zipcode can be layered on top of it [7, 8]. In fact, Zipcode was designed to rely only on basic services provided by the CE/RK primitives. The CE/RK HOST/NODE model and (node, pid) pairs form the underlying basis of Zipcode's process model. Using this layering approach, Zipcode has been ported on many platforms

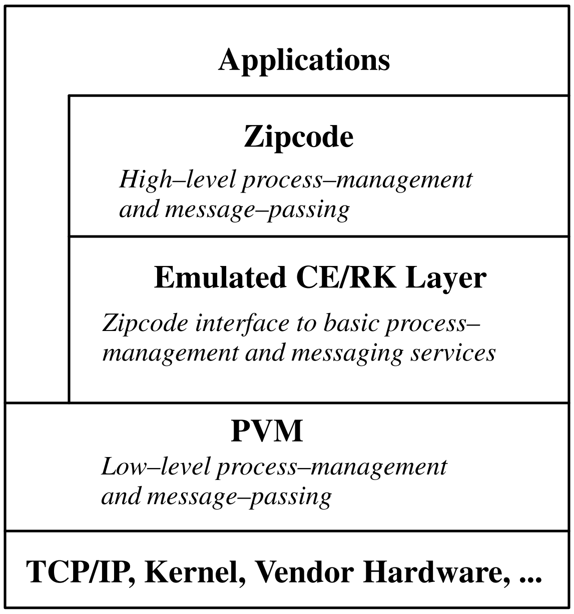

Figure 1: A Hierarchical View of The Integrated System

including Symult S2010, nCUBE/2, iPSC/860, Delta, Paragon, BBN TC2000, CM-5 scalar machine, Sun workstations, and RS/6000 networks [7].

3.2 Internals of the CE/RK Layer

The CE/RK layer is the interface mechanism through which Zipcode and PVM are integrated into a coherent environment. Implementation of the emulated CE/RK primitives will be discussed in the next section. In this section, we present the internals of the integrated system. In particular, we discuss how the CE/RK layer maintains a coherent process model and message-passing environment. To simplify the environment, we have assumed an SPMD and one process per node model.

The CE/RK layer has two main functions. First, it provides Zipcode basic process management and messaging services. Second, it provides necessary mapping between the Zipcode and PVM resources to ensure consistency of the underlying process model and messaging environment. The CE/RK layer performs the following tasks to fulfill these two functions.

Environment Configuration

The CE/RK layer maintains basic configuration information about the underlying environment, including number of nodes in the system, host names of all host computers, and the location of each process. Since Zipcode has a high-level view of the hardware resources in the environment, whereas PVM has a relatively low-level view of it, the CE/RK layer needs to maintain necessary mapping between the two systems. For example, a mapping between the logical node number and its correspondent host machine name is maintained by the CE/RK layer.

Process Names Mapping

The CE/RK layer provides mapping between the PVM tids and the Zipcode addressee list, which is an array of (node, pid) pairs. Zipcode allows users to specify a "recommended" addressee list when a process group is created. Zipcode assumes that the underlying CE/RK system may or may not overwrite the user specified addressee list. PVM, on the other hand, enforces system-defined process identifiers (i.e., the tids). In this implementation, we make the Zipcode layer pid equal to the PVM layer tid to solve the mapping problem. The CE/RK layer is responsible for passing the actual tid/pid of the spawned process to the Zipcode layer.

HOST/NODE Process Model

The CE/RK layer supports the basic HOST/NODE model which Zipcode is based upon. A HOST program is responsible for initiating the environment and spawning other processes. It is usually the same process as the Post Master, or the group leader, in a Zipcode environment. The CE/RK layer ensures that the initial communication between a HOST program and its NODE programs is established before it passes the process group back to the Zipcode layer. The initial communication between the HOST and the NODE program informs the latter of all necessary information about itself and the group. The information required by a NODE process during the initial communication may include the actual pid of the HOST program, pids of all group members, its own pid, total number of nodes in the environment, and the logical node number of the process.

In this implementation, we rely on the Zipcode layer to disseminate some of the above information to the NODE programs in order to save message bandwidth. In particular, we rely on Zipcode to disseminate the actual addressee list, or the (node, pid) pairs, to all the members of the group. We can do so because Zipcode disseminates group information as part of its mailer creation process. To further save message bandwidth, we also eliminate information on the

NODE program's tid and total number of nodes from the initial HOST-NODE communication. We can do so because each NODE program can easily obtain the information from simple PVM calls. Therefore, the only information needed to be passed from HOST to NODE in the CE/RK layer is the actual HOST pid and the logical node number of the NODE program.

Basic Buffer Management

The Zipcode layer relies on the CE/RK layer to allocate and deallocate message buffer space. A message in Zipcode is called a letter. A letter has a layered structure, with the letter body (user data) preceded by Zipcode header information which contains the PO BOX information, mail class information, the context-id, and so forth. The CE/RK layer allocates message buffer space upon Zipcode's request, but makes sure that the message is of length (Zipcode specified buffer length + CE/RK message header length). This is so that CE/RK can store its own layer-specific information into its own message header. Currently, the header only stores the length of the buffer space. After the buffer space is allocated, the CE/RK layer passes a pointer to the requested buffer space to the Zipcode layer.

One complication occurs when integrating the above buffer management scheme with the PVM's scheme. The Zipcode layer requires direct access to the message buffer space to support the layered message header scheme. However, PVM buffer space is not directly accessible to the users. PVM provides calls to create message buffers and returns an assigned logical buffer id. PVM also provides calls to pack and unpack data for the users into and from a buffer based on the buffer id. To solve this problem, the CE/RK layer is required to explicitly pack and unpack buffer content from Zipcode buffer to the PVM buffer, and vice versa, when send and receive operations are performed. After each send, the CE/RK layer deallocates the Zipcode layer buffer. Before each receive, it allocates a Zipcode layer buffer to store the received message.

Basic Process Management and Messaging Services

The CE/RK layer provides basic services for process creation and termination, and untyped blocked and unblocked message passing. All these services can be provided by direct PVM calls after appropriate mapping between Zipcode and PVM resources. An example of this type of mapping is between the Zipcode level HOST (node, pid), which usually has constant values (-1, 0) in most Zipcode implementations, and the PVM level HOST tid. This mapping is necessary when the PVM send operation is used to send messages to the HOST program.

4 Implementation Details of the CE/RK Primitives

This section presents the CE/RK primitives and implementation details for each primitive. The CE/RK primitives are implemented using PVM 3.2 calls whenever possible. Some of the CE/RK primitives, such as getmc(), freemc(), spawn(), and xspawn(), can only be invoked by the HOST program. Certain functions are not implemented because they are not central to the operations of the integrated environment at this point. The following primitives have been implemented and partially tested on Sun workstations networks running PVM 3.2 under SunOS 4.1.3.

CE Calls

- int getmc(char `*comp_name`, int nnodes, char *hostfile) allocates a multicomputer environment. In this implementation, it starts PVM daemons on all machines in the specified *hostfile*. It also calls `pvm_config()` to get configuration information about the allocated PVM environment. Finally, it stores a data structure to map a processor's logical node number to its host name.
- `int freemc(void)` deallocates the multicomputer environment. In this implementation, it kills all PVM daemons on all hosts.
- `int cosmic_init(void)` enrolls the process into the environment. For the HOST program, `pvm_mytid()` is called to enroll itself into the PVM environment. For NODE programs, the function performs a blocking receive to get a message from the HOST. The message contains the HOST's actual pid/tid and the logical node number of the NODE process.
- `int cosmic_exit(void)` unenrolls the process from the environment. It calls `pvm_exit()` to unenroll the process from the PVM environment.
- int spawn(char *prog, int node, int pid, char *state) spawns a process on a node. This function is implemented using `pvm_spawn()`, which

returns the PVM tid of the spawned process. The actual tid is passed back to the Zipcode layer using a shared variable between the Zipcode layer and the CE/RK layer.

- int xspawn(char *prog, int cnt, int *nodelist, char *state) spawns *prog* on *cnt* number of nodes as specified in the *nodelist*, which is in the form of an array of (node, pid) pairs. The implementation of this call is similar to the previous function. The actual tids of the spawned processes can be passed back to the Zipcode layer by writing the tids directly to the *nodelist*. This can be done because the caller of this function passes a pointer to the addressee list instead of the addressee list values.

- `int ckill(int node, int pid)` kills a process on one or more nodes. This call currently is not implemented.

- `int mynode(void)` returns calling process's logical node number. For the HOST process, this function returns the constant HOST, which is -1 in this implementation.

- `int mypid(void)` returns current process's pid. For the HOST program, it returns the constant defined for HOST pid, which is 0 in this implementation. For a NODE program, it returns the actual PVM tid of the process.

- `int nnodes(void)` returns the number of nodes in the environment. The function calls `pvm_config()` to get the total number of hosts, N, in the environment. It returns (N-1) to exclude the utility node where the HOST program is on.

- `void print(char *fmt, ...)` is similar to C function `printf()` except that the output is preceded by (node, pid) pairs. This function is currently not implemented.

- `unsigned long clock()` returns the current nodal clock value in microseconds. This function is currently not implemented.

RK Calls

- `char *xmalloc(int len)` allocates a message buffer of length *len*. This function calls the C function `malloc()` to allocate a message buffer with length len + `RK_MSG_HDR_LEN`, where `RK_MSG_HDR_LEN` is the RK level message header length. It stores the requested message length *len* into the header. It then returns a pointer to the user buffer space, which starts at the first byte following the RK header.

- `void xfree(char *msg)` deallocates the message buffer. This function locates the real message pointer by decrementing *msg* by RK message header length in bytes. It then calls the C function `free()` to deallocate the entire message buffer space, including its RK header space.

- `int xlength(char *msg)` returns the length of a previously allocated message. In this implementation, it returns the user buffer length, which is stored in the message header.

- `char xrecv()` receives a message without blocking. It returns the message pointer if a message is in the queue, otherwise it returns NULL. This function is implemented using `pvm_nrecv()`. It first creates a message buffer using `xmalloc()`. After the message has been received, it uses `pvm_upkbyte()` to unpack the data into the allocated space.

- `char xrecvb()` blocks until a message can be received. Implementation of this function is similar to `xrecv()` except that it uses `pvm_recv()` blocking receive call.

- `int xsend(char *msg, int node, int pid)` sends a message pointed to by *msg* to the process *pid* on node *node*. This function first checks to see if *node* equals to the constant HOST. If so, it substitutes *pid* with the actual HOST pid/tid. It then uses `pvm_initsend()`, `pvm_pkbyte()`, and `pvm_send()` to complete the send operation. Finally, it deallocates the message space by calling `xfree()`.

- `xmsend(char *msg, int count, int *dest)` sends a message *msg* to multiple destinations specified by the integer array *dest* in the form of an array of (node, pid) pairs. Implementation of this function is similar to the previous function except that it uses PVM multicast operation to send to multiple destinations.

5 Summary and Conclusions

In this paper, the architecture and implementation of an integrated message-passing system based on Zipcode and PVM are presented. It explains how the

CE/RK layer primitives are implemented to provide a coherent integrated message-passing environment. Although the integrated system at its current state does not provide optimal performance, it does provide a convenient environment where both low-level and high-level constructs can coexist.

We view Zipcode as an example system of a high-level message-passing paradigm, which MPI will eventually support. We also recognize that PVM provides an easy-to-use system with a large users base. The PVM system is a reasonable starting point for distributed software development; however, it is not sufficient to support development of reliable and scalable parallel libraries and large-scale distributed software. We have ported Zipcode on top of PVM, mainly to provide an integrated environment where existing PVM applications can gradually migrate to a higher-level message-passing paradigm. Planned future enhancements include performance improvement of the integrated system as well as adding heterogeneous environment support.

Acknowledgements

The work presented in this paper was directed by Dr. Anthony Skjellum. His guidance is appreciated. The author would also like to acknowledge Nathan Doss for his help, and Ron Brightwell and Charles Still for their previous work in porting Zipcode.

References

[1] Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).

[2] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 USER'S GUIDE AND REFERENCE MANUAL. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.

[3] Brian Grant and Anthony Skjellum. The PVM Systems: An In-Depth Analysis and Documenting Study - Concise Edition. Technical Report UCRL-JC-112016, Lawrence Livermore National Laboratory, 1992.

[4] Brian Grant, Anthony Skjellum, and Lance Burton. The PVM Systems: An In-Depth Analysis and Documenting Study – The Full Report. In preparation, August 1993.

[5] Charles L. Seitz, Jakov Seizovic, and Wen-King Su. The C Programmer's Abbreviated Guide to Multicomputer Programming. Technical Report Caltech-CS-TR-88-1, California Institute of Technology, 1989.

[6] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. October 1993.

[7] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20:565–596, April 1994.

[8] Anthony Skjellum, Steven G. Smith, Charles H. Still, Alvin P. Leung, and Manfred Morari. The Zipcode Message-Passing System. Technical Report UCRL-JC-112022, Lawrence Livermore National Laboratory, 1992.