Migrating from PVM to MPI, part I: The *Unify* System*

Paula L. Vaughan[†]
Anthony Skjellum[†‡§]
Donna S. Reese[†§]
Fei-Chen Cheng[§]

NSF Engineering Research Center for Computational Field Simulation[†]&
Department of Computer Science[§]
Mississippi State University
Mississippi State, MS 39759

July 15, 1994

Abstract

This paper presents a new kind of portability system, *Unify*, which modifies the PVM message passing system to provide (currrently a subset of) the Message Passing Interface (MPI) standard notation for message passing. *Unify* is designed to reduce the effort of learning MPI while providing a sensible means to make use of MPI libraries and MPI calls while applications continue to run in the PVM environment. We are convinced that this strategy will reduce the costs of porting completely to MPI, while providing a gradual environment within which to evolve. Furthermore, it will permit immediate use of MPI-based parallel libraries in applications, even those that use PVM for user code.

We describe several paradigms for supporting MPI and PVM message passing notations in a single environment, and note related work on MPI and PVM implementations. We show the design options that existed within our chosen paradigm (which is an MPI interface added to the base PVM system), and why we chose that particular approach. We indicate the total evolution path of porting a PVM application to MPI with the help of porting libraries. Finally, we indicate our current directions and planned future work.

*Submitted to *Frontiers '95*, The Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia.

‡Corresponding author. e-mail: *tony@cs.msstate.edu*; phone: 601-325-8435.

1 Introduction

Parallel and cluster-based message passing has evolved to a standard in MPI [9], the Message Passing Interface. However, many codes are still using popular, but non-standard, notations, since the standard is new. Tools that ease transition to the MPI standard will save money, provide for a larger software industry, and accelerate acceptance of parallel and/or cluster computing. This paper describes one such tool for PVM-to-MPI migration, *Unify* [5]. A PVM-to-MPI migration tool is seen as valuable, since PVM [11] is used in a large fraction of recent codes.

1.1 Comparing PVM and MPI

1.1.1 PVM

PVM emerged as one of the most popular cluster message-passing systems in 1992, though PVM did not originally work on nodes of multicomputers (more recently, multicomputer vendors have offered both layered and native versions of PVM for multicomputer message passing). PVM's model and semantics closely resembles those of systems such as PICL [12] and NX/2 [17] from the mid to late 1980's. PVM's main contribution was its broad interoperability across workstation platforms. Another attraction was the portability of its TCP- and XDR-based implementation. However, its programming paradigm offered no significant advantage over the paradigms of the earlier systems.

1.1.2 MPI

By way of contrast, the MPI standardization effort had its advanced features rooted in other systems, particularly PARMACS [16], Zipcode [19], and Chimp [7, 8], which included higher-level concepts in their programming models. Formalization of message-passing models, notions of correctness in loosely synchronous programming, and considerations of safety had become concerns of programmers building significant applications with message passing. These concerns were ignored by the early systems and also by PVM (*e.g.*, groups, contexts, communicators, topologies). Furthermore, MPI was to incorporate concepts that would deliver higher performance to the user; these concepts were the product of experience with higher-level message systems (*e.g.*, datatypes, powerful collective operations).

The challenge for MPI is to overcome the inertia of existing codes written in other, less-powerful, message passing notations, particularly PVM, P4, and NX. *Unify* is one strategy to help overcome that inertia.

1.2 The *Unify* Strategy

The *Unify* system is an implementation of MPI built on top of PVM. To avoid any impact on current PVM applications, minor internal modifications were made to the PVM system rather than asking for significant user effort. The result is a dual-API system where users may invoke functions from both MPI and PVM in concert. The goals of *Unify* are to help users of PVM migrate to MPI, lower the learning curve and entry cost for experimenting with and learning MPI, support third-party libraries in the PVM environment during the period of migration, and provide a testbed for new MPI concepts (such as dynamic process creation and intercommunicator collective operations).

1.3 Framework of Paper

This paper will first discuss the challenges of system migration, especially as they apply to the PVM-to-MPI evolution. Three possible migration paradigms that allow a dual-API feature are outlined. The paradigm we have chosen is explained in more detail. We discuss the design strategy of the *Unify* system in an overview and in detail. We explain the state of the current *Unify* implementation and provide a list of current MPI functions supported. Finally, future work in this area is outlined, particularly in relation to another of the migration paradigms discussed here.

2 Evolving from PVM to MPI

2.1 Paradigms

There are three paradigms that would provide a dual-API to aid in application migration from PVM to MPI. They are as follows: building the MPI protocols over the PVM system, building PVM protocols over the MPI system, or building PVM and MPI together on top of a portable device such as an ADI (Abstract Device Interface)

[13]. Each of these paradigms has its advantages and its own possible place in the process of evolving from PVM to MPI.

The first approach, building the MPI protocols and API on top of PVM, is in a sense abstracting message passing to a level higher and using PVM as the portable device driver: MPI is added to PVM calls in the API and programs run in the PVM environment, and performance of the system differs insignificantly from that of PVM by itself. By using MPI point-to-point operations under the MPI collective routines and reasonably efficient algorithms [1] the collective MPI operations may be more efficient than corresponding PVM calls. This approach describes the planned strategy of our current system, *Unify*, though we have not yet optimized collective operations. Furthermore, we have not yet introduced the full multiple-program, multiple-data (MPMD) model into *Unify*, but rather support the single-program, multiple-data (SPMD) model at present; this extension requires study of multiple, MPI worlds, and will be considered later.

We consider the next approach, building the PVM protocols and API on top of MPI, to be the next stage of the evolution process. By building a version of PVM that will run on top of any version of MPI, we achieve a portable and more efficient system. PVM programs can run in the MPI environment and gain the higher performance of that environment. We undertake this approach as our next step (we call this work the *Simplify* system [20]).

Building PVM and MPI together may be a more purist approach, but does not have any cost/time advantages over the other two approaches. Actually, it has cost penalties: we would lose the advantage of using either of the systems' existing run-time environments, and there would be a duplication of effort by developing each system from scratch. This approach would violate our goal of having a timely tool to help the application community move to the new standard now, before further inertia develops.

2.2 Issues in Porting from PVM to MPI

There are difficulties inherent in porting an application from one message-passing system to another. Most migrations require a good knowledge of both systems and the corresponding functionality of each. When the migration is undertaken, the application must normally be fully converted to the new system before the revised application may be tested. Generally, no remnants of the old system may remain.

An initial step in system migration is analyzing the possible mappings of functions in one system to the other.

Some functions from the old system (PVM) map one-to-one to the new system (MPI). In this case only the syntax of functions in each system must be learned. In the PVM-to-MPI case, point-to-point, environmental management and some collective functions will fall into this one-to-one mapping. Other functions require a greater semantic knowledge of MPI. These include, among others, communicator functions, and most collective operations.

Unify's initial implementation is a subset of MPI. As we evolve *Unify* to the full standard specification, features of MPI will always be presented in order of increasing complexity (or less wide applicability) to users, because this presentation is easier to grasp. The initial version contains simple instructions, facilitating more focused learning, compared to the MPI standard document [9]. The incremental implementation and documentation of *Unify* presents MPI features in a manner by which the users may comfortably extrapolate PVM knowledge to the generalized MPI framework.

The dual-API feature of *Unify* supports the gradual conversion from PVM to MPI. Communication functions of both MPI and PVM may co-exist within the same application program and/or libraries. As the user gains famliarity and confidence with the functions of MPI, these functions may be added to the application being converted. This creates a feasible migration path from one system to the other.

2.3 Stages of Evolution

When evolving an application from PVM to MPI, in a way that retains a working application throughout the transition, several logical stages can be identified. An unaltered version of the PVM application code is ported into the *Unify* system by recompiling using the *Unify* version of PVM header files, and the patched version of PVM [5]. Because of the *Unify* implementation strategy, the application experiences negligible performance changes[1]. After a time, the program evolves to use some MPI calls, and possibly MPI libraries. Possibly, the program is changed to add MPI calls and drop PVM calls, correspondingly. In any event, the programmer gains familiarity with MPI. After the code has been used for a time in this fashion, the desire for the higher performance of a full, vendor-level MPI implementation arises, but PVM calls remain in part or all of the code. The code is consequently moved to the *Simplify* system, running on top of an MPI implementation. Changes to message passing code need not be

[1] Our slight change to the PVM protocol adds a trivial amount of additional latency to messages, and an additional integer comparison when messages are selected for receipt.

made at this time, but changes must be made to process-management code, since the code has been moved out of the PVM process-management environment; normally, such alterations do not constitute a big effort (except if an MPMD application is being ported). Performance increases to reflect the underlying MPI implementation, because the PVM wrappers are particularly lightweight in the *Simplify* strategy. Furthermore, since *Simplify* will be compatible with all MPI implementations, the code receives another incremental gain in portability.

3 Related Work

Related work, of two forms, is described next. First we describe other MPI implementations that are currently more complete than *Unify*, but do not address gradual portability from PVM. Then, we touch on LAM MPI, which does provide both a PVM and MPI interface.

3.1 Other MPI Implementations

There are three other MPI implementations worthy of noting here. MPICH [6] from Argonne National Laboratories and Mississippi State University is a full MPI implementation, portable to all parallel machines and workstations running Chameleon [15], p4 [3], or PVM [11]. Its implementation is the basis of IBM's MPI (MPI-F) [10] and Intel's MPI, and is also used by Meiko. The LAM MPI [2] implementation from the Ohio Supercomputing Center also supports the full MPI standard, and it includes PVM compatibility, having a PVM interface in addition to the MPI personality, all running on native communication. LAM MPI is the closest product to *Unify*; all the other products provide an MPI environment, not a dual-API environment. *Unify* differs from LAM in the following ways:

- Use of PVM environment to minimize disruption of user efforts.
- Patches PVM with richer protocol for safe MPI communication.
- Leverages PVM improvements without maintaining separate implementation (using patch).

Chimp MPI from the Edinburgh Parallel Computing Centre has a goal of creating a portable programmable and efficient library for their cluster message-passing applications [7, 8]. Chimp MPI is the basis of Cray's MPI.

3.2 Other Relevant Libraries

Gropp and Lusk have developed MPE, Multiprocessor Extensions for graphics and profiling for the MPI environment [14]. Furthermore Viswanathan et al have developed intercommunicator extensions for MPI that provide collective intercommunicator operations and topologies; they are also exploring the use of intercommunicators for dynamic process worlds [18]. We intend to provide support for the use of MPE and MPIX with *Unify* as soon as our system supports a larger subset of MPI. Furthermore, we intend to test out the dynamic process concepts of MPIX initially using *Unify*.

4 Design Strategy

The *Unify* system design strategy is to begin with a subset of MPI. This subset should retain the full functionality of PVM plus some communicator-management functions of MPI. The subset has been selected to be small enough to test the concept of PVM and MPI dual-API system, but large enough to be useful in upgrading and replacing current PVM applications and libraries.

Two methods for representing communication contexts were evaluated for the design of *Unify* [4]. The first embeds the communication context within PVM types. Contexts would be automatically combined with user-specified MPI tags. PVM types would therefore have a dual purpose. The upper half-word would represent communication context, and the lower half-word would remain for MPI tags. This strategy limits the MPI tag range from 0 to 65,535. This requires minor modification of PVM code, but would not allow wild card receives to be used in either PVM or MPI. The second approach, which is more faithful to both PVM and MPI API's, is to modify PVM source code to carry communication contexts through its primitives, as a separate word and to pick a default context value for PVM communication (which is never used by MPI). This approach does not limit the tag range, nor does it limit the use of wildcards. We have therefore chosen the second approach.

By modifying the source code of selected PVM routines to accept the MPI communication context word, our design upgrades the PVM envelope and its receipt selectivity. Modified PVM primitives have been renamed and macros are used to preserve the conventional PVM calling sequence. This provides the dual-API (PVM and MPI together) by-product. Importantly, no requeueing of messages is needed because of the design.

Minor internal modifications have been made to PVM. A communicator context field has been added to the parameter list of selected PVM communication functions. The changes to the PVM source code are implemented with "patch." This eases the conversion to newer versions of PVM. Vendors with their own implementations of PVM also gain information through the patch files to convert their versions to *Unify* compatibility.

The MPI communicator data structure contains a process group, context information, and MPI internal information. The process group in a communicator is a list of group member processes in rank order. *Unify*'s group implementation is a list of PVM tids (task ids) [11]. The communicator context information isolates the communication within a group in this communicator from other communicators. A figure depicting the structure of an MPI communicator in *Unify* and its connection to PVM tids is Figure 1.

5 Current Implementation

The implementation of *Unify* at this writing is version 0.9.2. It features point-to-point communication, collective-communication, some environmental-management, and C and FORTRAN bindings. Table 1 lists a subset of MPI with which a great many parallel applications, and libraries may be implemented. They are all available wihin this version of *Unify*. The other functions provided by *Unify* are listed in Table 2. PVM releases are tracked and reverse engineered to work with *Unify*. Four versions of PVM are supported at the writing of this paper (3.2.0, 3.2.6, 3.3.0, and 3.3.2). The changes that have been made to each of these versions have been nearly identical to one another except for source line numbers; thus, tracking the progress of PVM, even without any feedback to its developers, has proven tractable.

Unify point-to-point communication operations call the corresponding modified PVM function. Table 3 lists the PVM functions modified for *Unify* implementation. The modified PVM function calls include context information to isolate communication from other communicators and from user-called PVM functions. Source code for `MPI_Send` and `MPI_Recv` as implemented in the *Unify* system is depicted in Figures 5 and 6. Collective communication operations in *Unify* call MPI point-to-point functions. These operations use reasonably efficient, but not elaborate, algorithms at present. The MPI operations are not layered on top of PVM collective operations, because not all of the operations map one-to-one, and many of the operations may be implemented more efficiently and reliably using MPI point-to-point as a basis (because additional overhead to assure reliability of the MPI collective calls might be needed with layering, and general overlapping groups need to be supported).

The program example in Figure 2 is a simple routine in which both MPI and PVM functions have been called. This program divides MPI_COMM_WORLD into three communicators using the MPI_Comm_split function with the *color* option set to *rank* mod 3. Then it passes information in a ring within each of the new communicators using PVM point-to-point communication. The final action is to use the MPI reduction function, MPI_Allreduce, to sum the ranks of the processes within the new communicators. Only two functions unique to the *Unify* system have been developed to support dual-API. `unify_rank_to_tid` allows the program to access PVM tids by providing the MPI rank, and `unify_tid_to_rank` provides the inverse function. The C interface bindings for these functions is depicted in Figure 4. Figure 3 is an example of a program with the same functionality as the program shown in Figure 2, but using only MPI functions.

6 Future Work

Several efforts are planned for the future. One of our first goals is to incorporate intercommunicator capability into *Unify*, and then to experiment with dynamic process worlds. This will be a beneficial study prior to the beginning of a second round of MPI standardization, later in 1994.

We seek an upgrade path that will leverage the entire MPICH implementation, rather than trying to replicate it. Our goal in the pre-version-1 *Unify* releases is to prove the concept and to get user feedback. However, we will be striving to reuse code in the MPICH implementation starting later in the year. Currently, there is no code overlap between the systems. This transition will not be totally trivial, because MPICH is based on an abstract device interface strategy. That means that we could utilize collective communications in MPICH that are based on point-to-point operations; we could also easily utilize caching, and intercommunicator capability. However, our point-to-point routines will have to continue to differ from that of MPICH.

Faithfully supporting the entire MPI specification without further modifications to PVM is impossible. Specifically, PVM does not support overlapped communication and computation as assumed by calls like `MPI_Isend` and `MPI_Irecv`. Furthermore, we would need to make much more elaborate modifications actually to increase performance of the system, such as by further enhancing PVM's protocols and reducing the number of copies and buffering associated with the PVM calls we currently use[2]. Such efforts are beyond the scope of this project; instead, we plan to support the *Simplify* system as the logical next step in the evolution of applications to MPI. We view the combined value of a working, complete *Unify* system, and a fast *Simplify* interface, as the right investment to provide portability and application evolution to MPI.

However, we see at least one area in which the *Unify* implementation of MPI will be able to get higher performance than the corresponding program written in PVM. This pertains to the management of heterogeneity in communicators (see Figure 7). In the communicator data structure, information (an integer flag) could be cached concerning whether a groups is heterogeneous in data format or not. That could be passed down to avoid data conversion at the PVM send/receive level, thereby increasing bandwidth. This short-cut applies equivalently to MPI collective operations. It should further be noted that the homogeneity of subgroups could be determined whenever they are bound into communicators, thus providing higher performance message passing whenever a homogeneous sub-group is involved[3]. Because PVM lacks the concepts of static group and communicator, the user must manage all such interactions with the system.

7 Conclusions

In this paper, we have introduced a new message-passing implementation of MPI; it is currently a workable subset implementation that works in the PVM message-passing environment. *Unify* works by removing the semantic bottleneck in PVM's point-to-point communication protocol adding a context word that can be used to manage safer communication in an MPI application programmer interface (API). This new system is a dual-API environment, that allows a user to continue to use unmodified PVM calls, while libraries or other user code make use of MPI calls. We see this environment as an excellent vehicle for advancing applications to MPI, while retaining a working code during the transition phase. Furthermore, the programmer makes only trivial upfront investments to utilize *Unify* instead of the vanilla PVM 3.x environment. As an added benefit, *Unify* will

[2] Some versions of PVM use less buffering and copying, but these are not available as source code, so they cannot readily be used with the *Unify* concept.

[3] Internally, we would have to store the architectures of particular processes, to allow this property to be detected. This is clearly possible in a system built on top of PVM.

guarantee that parallel libraries developed in MPI will be available to PVM users who compile their codes with the *Unify* patches.

We have demonstrated a working *Unify* system, together with example programs of MPI under the system, and a dual-API capability where both MPI and PVM calls are used together. Currently, there are still gaps in our implementation: we do not support intercommunicators, datatypes, caching, topologies, and parts of the environment capabilities of MPI. Furthermore, PVM programs run only in the single-program-multiple-data mode when supporting the current version of *Unify*. However, this paper demonstrates that the concept of an MPI interface inside PVM works and has merit; our future work will incrementally add more of the MPI specification to *Unify*.

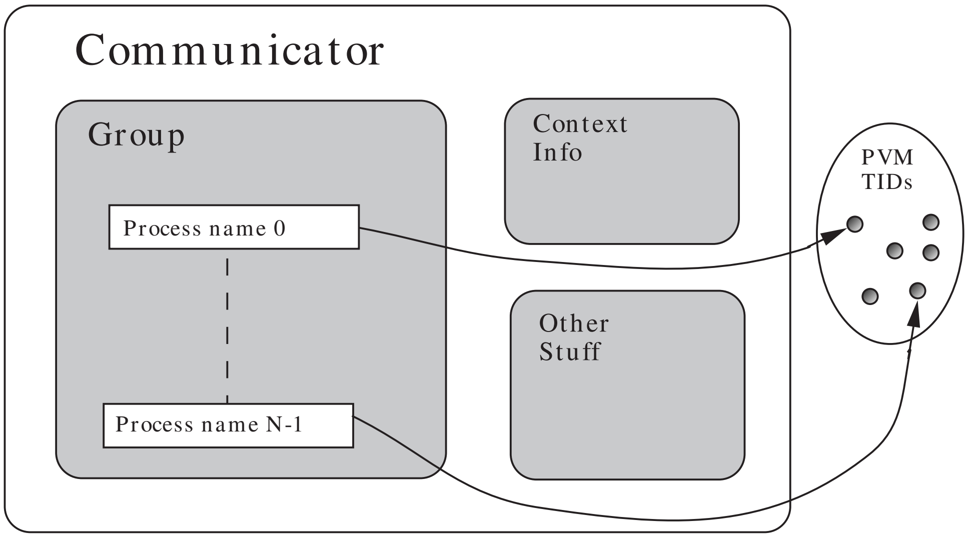

Figure 1: A communicator consists of a group, context information, and information internal to MPI.

Function	What it does
MPI_INIT	Initialize MPI
MPI_COMM_SIZE	Find out how many processes there are
MPI_COMM_RANK	Find out which process I am
MPI_SEND	Send a message
MPI_RECV	Receive a message
MPI_FINALIZE	Terminate MPI
MPI_BCAST	One-to-all broadcast
MPI_REDUCE	All-to-one reduction operation
MPI_ALLREDUCE	All-to-all reduction operation
MPI_COMM_DUP	Make a copy of a communicator
MPI_COMM_SPLIT	Partition a communicator into others
MPI_COMM_FREE	Communicator deletion operation
MPI_COMM_GROUP	Group constructor from within communicators
MPI_GROUP_EXCL	Group constructor by deletion of processes
MPI_GROUP_FREE	Group deletion operation

Table 1: Fifteen of the most commonly used MPI functions — all available in *Unify*.

Function	What it does
MPI_INITIALIZED	Has MPI_INIT been executed?
MPI_BARRIER	Synchronization function
MPI_GROUP_SIZE	How many processes are in a group?
MPI_GROUP_RANK	What is my rank within the group?
MPI_GET_PROCESSOR_NAME	Get the processor name
MPI_ERROR_STRING	Get error string from error code
MPI_GET_COUNT	Get the length of the message
MPI_ISEND	Asynchronous send
MPI_IRECV	Asynchronous receive
MPI_WAIT	Wait until request is complete
MPI_TEST	See if request is complete
MPI_PROBE	Check for incoming messages
MPI_IPROBE	Asynchronous probe
MPI_SENDRECV	Combined send and receive message
MPI_GATHER	All-to-one function
MPI_SCATTER	One-to-all function
MPI_ALLGATHER	All-to-all gather
MPI_ALLTOALL	All-to-all gather-scatter
MPI_COMM_CREATE	Creation routine for new communicators
MPI_GROUP_TRANSLATE_RANKS	Determine process ranks
MPI_GROUP_UNION	Union two groups
MPI_GROUP_INTERSECTION	Intersect two groups
MPI_GROUP_DIFFERENCE	Find difference between two groups
MPI_GROUP_INCL	Add processes to a group

Table 2: Other MPI functions provided with *Unify*.

```
#include "mpi.h"
#include "pvm3.h"
int main( int argc, char **argv )
{
    int       rank, size,gresult,color,key,pvmdata;
    int       destination, source,desttid,sourtid;
    int       greensize, greenrank;
    MPI_Comm  green_comm;
    /* initialize MPI */
    MPI_Init( &argc, &argv );
    /* find out who you are */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    /* find out how big the process world is */
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    /* prepare to split MPI_COMM_WORLD */
    gresult = 0;
    /* split MPI_COMM_WORLD using rank(mod 3) as       */
    /* the color and using a key which will reverse    */
    /* rank orders                                     */
    color = rank % 3;
    key =  100 - rank;
    MPI_Comm_split( MPI_COMM_WORLD, color, key, &green_comm );
    /* perform some communication with PVM directly   */
    /* using the communicator from the previous split */
    /* find out the size of and your rank in green communicator */
    MPI_Comm_size(green_comm, &greensize);
    MPI_Comm_rank(green_comm, &greenrank);
    /* compute sources and destinations */
    source = greenrank - 1;
    if (source == -1) source = greensize - 1;
    destination = greenrank + 1;
    if (destination == greensize) destination = 0;
    /* find the tids of the destination and source */
    unify_rank_to_tid(green_comm, destination, &desttid);
    unify_rank_to_tid(green_comm, source, &sourtid);
    /* compute and exchange data */
    pvmdata = rank * 30 - 23;
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&pvmdata, 1, 1);
    pvm_send(desttid, 0);
    pvm_recv(sourtid, 0);
    pvm_upkint(&pvmdata, 1, 1);
    /* add the ranks of the members of this communicator */
    MPI_Allreduce( &rank, &gresult, 1, MPI_INT, MPI_SUM, green_comm);
    /* clean up MPI */
    MPI_Finalize();
}
```

Figure 2: Example of a Dual-API (MPI and PVM) Program.

```
#include "mpi.h"
int main( int argc, char **argv )
{
    int       rank, size,gresult,color,key,data;
    int       destination, source,desttid,sourtid,greensize,greenrank;
    MPI_Comm         green_comm;
    MPI_Status       status;
    /* initialize MPI */
    MPI_Init( &argc, &argv );
    /* find out who you are */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    /* find out how big the process world is */
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    /* prepare to split MPI_COMM_WORLD */
    gresult = 0;
    /* split MPI_COMM_WORLD using rank(mod 3) as       */
    /* the color and using a key which will reverse    */
    /* rank orders                                     */
    color = rank % 3;
    key =  100 - rank;
    MPI_Comm_split( MPI_COMM_WORLD, color, key, &green_comm );
    /* find out the size of the green communicator */
    MPI_Comm_size(green_comm, &greensize);
    /* find out your rank within the green communicator */
    MPI_Comm_rank(green_comm, &greenrank);
    /* compute sources and destinations */
    source = greenrank - 1;
    if (source == -1) source = greensize - 1;
    destination = greenrank + 1;
    if (destination == greensize) destination = 0;
    /* do some MPI point to point communication */
    /* compute some data to exchange */
    data = rank * 30 - 23;
    /* send the data */
    MPI_Send(&data, 1, MPI_INT, destination, 2002, green_comm);
    MPI_Send(&data, 1, MPI_INT, destination, 2002, green_comm);
    /* receive the data */
    MPI_Recv(&data, 1, MPI_INT, source, 2002, green_comm, &status);
    /* add the ranks of the members of this communicator */
    MPI_Allreduce( &rank, &gresult, 1, MPI_INT, MPI_SUM, green_comm);
    /* clean up MPI */
    MPI_Finalize();
}
```

Figure 3: Example of a Single API (MPI only) Program.

```
int unify_rank_to_tid(MPI_Comm comm,
                      int      rank,
                      int      *tid)

int unify_tid_to_rank(MPI_Comm comm,
                      int      tid,
                      int      *rank)
```

Figure 4: C bindings of two new functions unique to the *Unify* system.

```
int MPI_Send(buf, count, datatype, dest, tag, comm)
void* buf;
int count;
MPI_Datatype datatype;
int dest;
int tag;
MPI_Comm comm;
{
int bufid, info;
int tid, context, mytag;
        /* check if send to a dummy rank, yes->return */
        if (dest == MPI_PROCNULL)
           return(1);
        /* get destination tid */
        tid = comm->group->tid[dest];
        /* use XDR encoding scheme */
        bufid = pvm_initsend(PvmDataDefault);
        /* packing data */
        unify_pack_data(buf, count, datatype);
        /* send msg */
        info = mvp_send(tid, tag, comm->context->pp_cnt);
}
```

Figure 5: *Unify* implementation of `MPI_Send` function.

PVM Function	***Unify* call**
pvm_mcast(tids, count, code)	mvp_mcast(tids, count, code, PVM_DEFLT_CTXT)
pvm_notify(what, code, count, vals)	mvp_notify(what code, count, vals, PVM_DEFLT_CTXT)
pvm_nrecv(tid, code)	mvp_nrecv(tid, code, PVM_DEFLT_CTXT)
pvm_probe(tid, code)	mvp_probe(tid, code, PVM_DEFLT_CTXT)
pvm_recv(tid, code)	mvp_recv(tid, code, PVM_DEFLT_CTXT)
pvm_send(tid, code)	mvp_send(tid, code, PVM_DEFLT_CTXT)
pvm_trecv(tid, code)	mvp_trecv(tid, code, timeval, PVM_DEFLT_CTXT)
def_match(mid,tid,code)	def_match(mid, tid, code, context)
mroute(mid, dtid, code, tmout)	mroute(mid, dtid, code, tmout, context)

Table 3: PVM functions modified for *Unify* implementation.

```
int MPI_Recv(buf, count, datatype, source, tag, comm, status)
void* buf;
int count;
MPI_Datatype datatype;
int source;
int tag;
MPI_Comm comm;
MPI_Status *status;
{
int bufid, tid, bytes;
int msgtag, context, mytag;
int *temp;
int size, i;
        /* check if receive from a dummy rank */
        if (source == MPI_PROCNULL)
        {
           status->MPI_SOURCE = MPI_PROCNULL;
           status->MPI_TAG = MPI_ANY_TAG;
           status->count = 0;
           return(1);
        }
        /* get source tid */
        if (source == MPI_ANY_SOURCE)
            tid = MPI_ANY_SOURCE;
        else
            tid = comm->group->tid[source];
        bufid = mvp_recv(tid, tag, comm->context->pp_cnt);
        /* unpacking data */
        unify_unpack_data(buf, count, datatype);
        MPI_Comm_size(comm, &size);
        /* modify status record */
        pvm_bufinfo(bufid, &bytes, &msgtag, &tid);
        for (i=0; i<size; i++)
           if(comm->group->tid[i] == tid)
              status->MPI_SOURCE = i;
        status->MPI_TAG = msgtag;
        status->count = bytes;
}
```

Figure 6: *Unify* implementation of `MPI_Recv` function.

```
typedef struct unify_group {
        int     *tid;
        int     count;  /* size count */
        int     ref_cnt;/* Increment one when MPI_Comm_create, MPI_Comm_dup
                           Decrement one when MPI_Group_free, MPI_Comm_free */
} *MPI_Group;

typedef struct unify_context {
        int     pp_cnt;   /* for point to point communication */
        int     gp_cnt;   /* for collective communication     */
} *MPI_Context;

typedef struct unify_comm {
        MPI_Group       group;
        struct unify_comm  *comm_coll;
        int             ref_count;
        MPI_Context     context;
        MPI_Errhandler  error_handler;
        int             errormode; /* for error handling                */
} *MPI_Comm;
```

Figure 7: Key data structures of the *Unify* MPI implementation.

References

[1] Mike Barnett, Satya Gupta, David G. Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Interprocessor collective communication library (intercom). In *Proceedings of the Scalable High-Performance Computing Conference (SHPCC)*, pages 357–364. IEEE, May 1994.

[2] Greg Burns, Raja Daoud, and James Vaigl. Lam: An open cluster environment for mpi. Available by anonymous ftp from tbag.osc.edu in pub/lam, 1994.

[3] Ralph Butler and Ewing Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[4] Fei-Chen Cheng. Unifying the MPI and PVM 3 Systems. Technical report, Mississippi State University — Dept. of Computer Science, May 1994. Master's Report. (Available by anonymous ftp: *cs.msstate.edu*, `pub/reports/feicheng.ps`).

[5] Fei-Chen Cheng, Paula L. Vaughan, Donna Reese, and Anthony Skjellum. The Unify System. Technical report, Mississippi State University — NSF Engineering Research Center, June 1994. Version 0.9.1.

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

[7] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[8] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[9] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[10] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of mpi. In *Proceedings of the 1994 International Converence on Parallel Processing*, 1994.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report TM-12187, Oak Ridge National Laboratory, 1993.

[12] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[13] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. (in progress).

[14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[15] William D. Gropp and Barry Smith. Chameleon Parallel programming tools Users Manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[16] R. Hempel, H.-C. Hoppe, and A. Supalov. *PARMACS 6.0 Library Interface Specification*. German National Research Center for Computer Science, December 1992.

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

[18] Anthony Skjellum, Nathan Doss, and Kishore Viswanathan. Inter-communicator extensions to mpi in the mpix (mpi extension) library. in preparation., 1994.

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

[20] Paula L. Vaughan, Nathan Doss, Anthony Skjellum, and Donna S. Reese. Migrating from PVM to MPI, part II.: The Simplify System. Technical Report in preparation., Mississippi State University — NSF Engineering Research Center, October 1994.