

BROWN UNIVERSITY  
Department of Computer Science  
Master's Project  
CS-94-M3

“Earl: A Tool for Portable Distributed Debugging”

by

Ira L. Lough

**Earl:  
A Tool for Portable Distributed Debugging**

**Ira L. Lough**

**Department of Computer Science  
Brown University**

**Submitted in partial fulfillment of the requirements for  
the degree of Master of Science in the Department of  
Computer Science at Brown University**

**December 1993**



---

**Professor Robert H. B. Netzer  
Advisor**

December 22, 1993

## Earl: A Tool for Portable Distributed Debugging

*Ira L. Lough*

*irl@cs.brown.edu*

Dept. of Computer Science  
Brown University  
Box 1910  
Providence, RI 02912

### Abstract

Developing and testing a new debugging tool on a collection of  $n$  parallel machines would require developing  $n$  implementations of that tool, one for each machine. Re-implementation is also required to run existing distributed programs (programs developed for a specific target machine) on some other system. Portability tools do exist which allow users to develop programs using communication primitives that have been implemented on a collection of distributed machines. However, they do not apply to existing programs which use the primitive operations of a particular machine. This paper describes the design of Earl, a system for program portability and the portable implementing and testing of debugging techniques for message-passing machines. A message-passing program (written for some parallel machine) is linked with a special library that maps the machine's native message-passing primitives to our generic set. Another group of libraries maps the generic set back to the native primitives of some - but not necessarily the same, parallel machine. To test a debugging tool using Earl, only a single instance of the tool need be implemented, and only for our generic set of message-passing routines (instead of the peculiarities of any particular machine). Since the primitive operations of all machines are mapped to and from this same set, the debugging algorithms incorporated into these routines can be used with any program on any supported machine. The ability to transport a program to other machines and run with a number of debug algorithms allows the user to quickly analyze the results of debug/performance tools across many architectures; possibly unveiling sensitivities of a debug algorithm's implementation to those architectures (e.g. *buffering* of messages may be more feasible on one type of architecture than on another). Our results show little overhead for the mappings; both in the common-case (porting programs to and from the same machine) and cross-machine mappings. In fact, significant speedup can be achieved when porting to some target machines (those better suited for a programs' inherent characteristics). Finally, with the completion of the Message-Passing Interface (MPI) Standard on the horizon, combined with the strong need of the community to utilize such a standard, the Earl Library may prove valuable for porting existing programs to this standard.

December 22, 1993

## 1. Introduction

Developing and testing a new debugging tool on a collection of machines is complicated by the need to implement the tool on each machine. For example, a tool for providing program re-execution on message-passing parallel machines would trace the order in which messages are delivered during execution (so the same order can be reproduced during re-execution). To implement and test such a tool on a collection of  $n$  parallel machines would require developing  $n$  message-tracing libraries, one for each machine. Additionally, porting existing distributed programs to a number of different target machines requires program re-implementation. Portability tools and environments exist that allow users to develop programs using communication primitives that have been implemented on a collection of distributed machines. However, such tools do not apply to existing programs which have been developed using the communication primitives of a particular machine. For this reason developing and assessing new debugging tools and programs across a collection of parallel machines can be time consuming. This paper describes the design of Earl, a system for program portability and the *portable* implementing and testing of debugging techniques for message-passing parallel machines. A message-passing program (written for any parallel machine) is linked with a special library that maps the machine's native message-passing primitives to our generic set. Another group of libraries maps the generic set back to the native primitives of some - but not necessarily the same, parallel machine. To test a debugging tool using Earl, only a single instance of the tool need be implemented, and only for this generic set of message-passing routines (instead of for the peculiarities of the message-passing primitives of a particular machine). This system allows the debugging tool to be implemented only once, but used on programs written for any of the machines. New debugging strategies can then be quickly implemented and evaluated across a variety of programs and machines.

Our main result is the design of a *single* set of generic message-passing routines. The set contains functionality common to that of all supported machines and the structural definition of each routine within the set is very general. These characteristics allow the primitives of all machines to be mapped to and from our one generic set. Subsequently, only two mappings are required to map the primitives of each machine to every other machine; allowing low maintenance (such as the addition of a new supported machine). Earl's infrastructure is a collection of message-passing libraries that implement the mappings to and from this set. Once this infrastructure is in place, debugging tools can be incorporated into the generic routines, and thus written only once. Programs developed with the communication primitives of a particular machine now become portable; their primitives can be mapped to this same generic set then back to the primitives of any supported machine. The debugging tools incorporated into these routines can now be used with any program on any supported machine. Combining program portability with portable debugging gives developers of debug/performance tools access to a wider range of quality test programs for which to analyze their algorithms (while only having to implement these algorithms once). Portability can be extended further when the communication primitives of a program are mapped (via the Earl Library) onto the primitives of an existing portability environment; mapping a program to these environments now allows the original program to be run on all target machines for which their primitives have been implemented. With the

*December 22, 1993*

portability environment of a message-passing interface standard almost complete (the MPI Standard)[4]6, Earl could be very useful in porting existing programs to this standard.

Our results show little overhead both in the common case mappings (to and from the same machine) and cross-machine mappings. Below we first cover related work and then outline Earl's design and test implementation results for the mapping of PVM, iPSC/2 and MPI Standard programs.

## **2. Related Work and Motivation**

Virtually no work has been published on portable debugging for parallel machines. However, systems for portable parallel programming are similar in spirit. Most provide a set of communication and tracing primitives that have been implemented on a collection of distributed machines. Each provides a means of writing portable programs: the user develops programs using the tools' primitives, and these programs can be run on a variety of machines without change. Examples of such systems are PICL (Portable Instrumented Communication Library)<sup>5</sup>, PVM (Parallel Virtual Machine)<sup>3</sup>, and Express<sup>1</sup>. Of particular importance among portability environments is the soon to be Message Passing Interface Standard (MPI). The MPI effort is an attempt to establish a practical, portable, efficient, and flexible standard for message passing. It uses the most attractive features of a number of existing message passing systems. As a result, the design of the MPI interface is not much different from current practice (such as Express and PVM); the interfaces are written at an application level and are easy to use, open, and extendible. Major goals of the standard are to allow efficient and reliable communication while maintaining a design that can be quickly implemented on many vendor's platforms. Designers of the standard hope that the definition of a message passing standard will provide vendors with a clearly defined set of routines that they can be implemented efficiently, or in some cases, be provided with direct hardware support. Furthermore, once established as a community standard, it is highly probable that libraries of debug and performance algorithms will be applied to the interface<sup>7</sup> (similar to PICL's tracing facilities which allow information useful for debugging to be collected).

Although most existing portable systems are valuable tools, they were not designed to aid in porting and debugging programs already written for other systems. In contrast, Earl is a system that allows debugging tools to be written once and applied to programs that have been developed on any collection of machines. Existing systems can be viewed as a mapping of its' communication primitives onto any distributed machine. Earl can be viewed as two mappings: one from the communication primitives of a given machine to our generic set, and another from our generic set to any distributed machine. This pair of mappings allows a single debugging tool to be run on programs written for any supported machine. In addition, it allows programs written for one machine to be automatically run on another; giving developers access to a wider range of test programs with which to analyze their debug and performance tools (while only having to implement these tools once). In this sense, Earl can be thought of as an extension of other portable systems; instead of programming with a given set of primitives, the user can program in the primitives of any machine, and automatically port (and debug) the program to any other

December 22, 1993

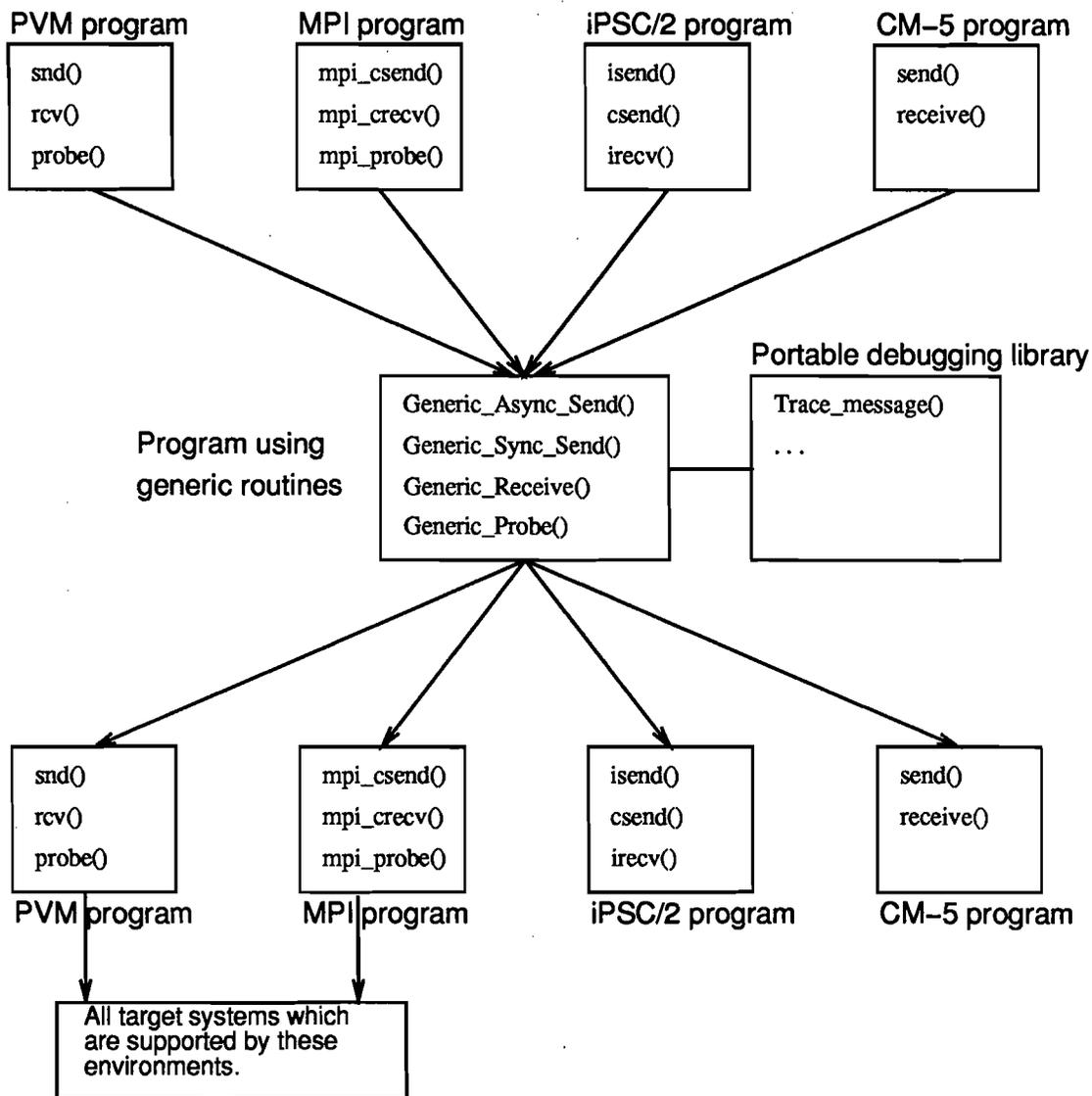
machine (Earl only addresses portability issues for a program's communication primitives, issues which emerge from porting between different operating systems and underlying hardware are not considered). Used in conjunction with existing portable systems, Earl can be used to map the communication primitives of a particular machine to the primitives of some portable set. In this way, the original program is able to run on all machines that portable system supports. This test implementation achieved this by mapping iPSC/2 programs[2] to the PVM System. These programs are now able to execute on such systems as the Connection Machine and workstation clusters such as a SUN or SGI. It is hoped that mappings similar to our test implementation will be utilized to map programs developed for these portable systems (or that have been developed with with the communication primitives of any particular machine) to the MPI standard.

### 3. System Design

The following sections describe the Earl system design. The design allows both machine dependent programs and debug algorithms to become portable. Machine specific programs become portable by being mapped through the generic set and onto the message-passing primitives of other supported machines. Debug algorithms become portable since they are developed with the same generic set of routines; and thus implemented only once. The following sections illustrate how this portability can be achieved through Earl's unique system design (a *two-level mapping* scheme) and the advantages of providing this portability within a single set of generic message-passing routines.

#### 3.1. Two-Level Mapping Scheme

The process of a two-level mapping is illustrated in Figure 1. The *first level* takes the message-passing primitives of one machine and maps them to our generic set. The *second level* then takes each generic routine and maps it back to the message-passing primitives of any of the target machines. Most existing portability systems begin with a set of primitives similar to those mapped to by our first level processing. It is this step which distinguishes Earl from such systems; by doing so it allows existing programs (those developed with the primitives of a particular machine) to become portable. The following illustrates a simplification of a two-level mapping for the common case (mapping to and from the same machine):



**Figure 1. Earl system design: A program is first mapped to one that uses only a generic set of message-passing routines, and then mapped to any target machine.**

Details of 1st Level	Details of 2nd level
<pre> firstlevel_send("same param list as original instr") {     /* Call Earl Generic Operation */     Generic_send("most general parameter list"); } </pre>	<pre> Generic_send("most general parameter list") {     /* Call original primitive operation */     original_send("original parameter list"); } </pre>

The two-level mapping scheme is achieved by linking the program with a special library (consisting of *firstlevel\_()* operations) to map any (supported) machine's message-passing primitives onto our generic set. The program is then linked with a second library (consisting of our *Generic\_()* operations) to map each generic operation back to a machine's primitive operations. Mapping the primitives of all machines to and from a *single* generic set has required that the definition of that set's operations be very general (making it easy to map from any primitive) and that they combine to provide a superset of each machine's functionality (making it easy to map back to any primitive). This superset of functionality allows programs from various machines (and thus varying levels of functionality) to be ported between on another. Providing these characteristics for our single set (and especially maintaining them in the wake of new operations and functionality from a new supported machine), has led to a structural design which is *open* and *extendable*. The set is considered open since the need to modify an existing operation (due possibly to the addition of an equivalent operation for a new machine) may involve only modifying the already general parameter list for the respective generic routine. Similarly, the set is extendable since the addition of functionality from a new supported machine simply involves creating a generic operation of equivalent functionality (or modifying an existing operation to have a wider range of functionality). The major result of our work is the design of a single generic set of message-passing routines which achieve this generality and functionality; by doing so they allow a common place in which to implement debug algorithms, efficient program portability, and easy maintenance such as the addition of new supported machines. The following sections describe how this generality is achieved within our single generic set while allowing efficient mappings for both the common and cross-machine cases.

### 3.1.1. Common-Case Mapping

A *common-case* mapping involves mapping the communication primitives of one machine to our generic set; the generic set then maps back to the communication primitives of the *same* machine. The philosophy of the two-level mapping scheme for the common case was illustrated in the above example. The original primitive operation is first substituted with a first-level equivalent routine; this routine then maps to our generic set. The general structure of each generic routine (e.g. a very inclusive parameter list) and the existence of an operation to handle the functionality of each primitive for all machines makes mapping to this set easy. The second level mapping then ensures that the functionality of the original operation is translated to the target machine. Of course, in the

common case, this simply involves having the generic operation re-issue the original primitive; allowing the common case to remain extremely efficient.

### 3.1.2. Cross-Machine Mapping

A *cross-machine* mapping is the mapping of primitives from one machine to our generic set; the generic set then maps the functionality of each primitive to *some other* machine. A principle concern for Earl's common generic set is that the functionality of the original primitive operation may not be directly supported on the target machine. The ability to map all machine's primitives to a single generic set depended on the generic set's general definition structure (making it easy to map from any primitive; as demonstrated in the common case example). Mapping *from* the generic set back to the primitive of any machine, however, requires that the set consists of all the functionality of any target machine; this allows the functionality for any primitive of the source machine to be mapped onto the target machine (even if this functionality is not directly supported). The following example illustrates how this functional superset can be achieved:

*Example.* A program developed for a particular system may use a routine which can receive a message of multiple message types. When porting the same program to a machine which does not have a functionally equivalent operation (e.g. iPSC/2), an Earl generic routine can be created by using a combination of the primitives available on the target machine. Earl first maps the primitive to the generic set via the first-level mapping. It is then the responsibility of the generic operation to create the functionality needed to map the primitive to the target machine:

1st Level Mapping	Needed member of functional superset
<pre> /* Original operation is mapped to */ /* a Generic equivalent.          */ firstlevel_recvm(buf, len, mtype, num) {     /* This functionality is not directly */     /* supported on target machine.      */     Generic_Recv_Multi(buf,len,mtype,num); } </pre>	<pre> Generic_Recv_Multi(buf, len, msgtype, num) {     /* Loop through all msgtypes */     while (true) {         /* Use non-blocking probe and recv() of */         /* target machine to achieve functionality */         probe(-1);         if message of "msgtype" has arrived             Recv() the message in;             return;         else             get next "msgtype";     } } </pre>

Thus, the Earl library contains a *Generic\_Recv\_Multi()* operation (and a library of others) which can be applied to

machines for which this functionality is not directly supported. There are many examples of such cases, the common philosophy of the Earl Library is to utilize the fundamental operations of each machine to create and support the widest range of functionality within the Generic Library. In this respect, the Earl Generic Library contains a superset of functionality for all supported machines. This functional superset allows programs from various systems (and thus varying levels of functionality) to be ported between one another. It is this characteristic (combined with each generic routines' general definition) that allows efficient mappings to and from the primitives of most distributed memory machines (and particularly to and from the primitives of existing portability environments).

In addition to mapping the functionality of primitive operations, a source and target machine's *environment processing* (process allocation, startup/initialization processing, etc) and *communication methodologies* may vary greatly. Thus, the benefits of the Earl design (program portability, portable debug algorithms, easy maintenance, etc) are dependent on the design's ability to port both the functionality and environment from one machine to any other. The following sections describe some of these important portability issues.

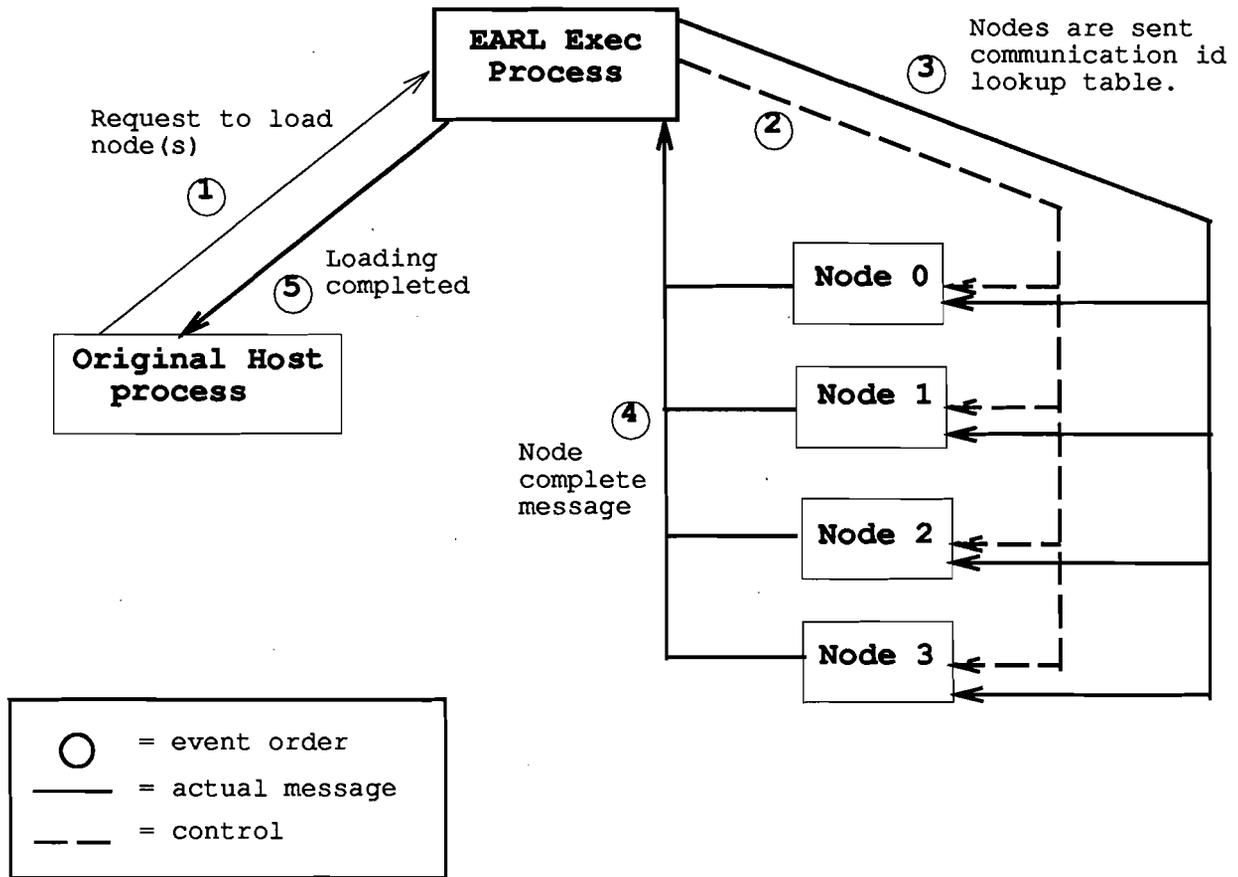
### 3.1.2.1. Mapping Startup/Initialization Processing

As noted throughout the on-going MPI effort, program startup/initialization processing is an important and difficult issue in establishing program portability. In fact, most portability issues (process initialization, establishment of process communication names/ids, etc) occur during program startup. Just as most parallel applications utilize a *host* process to help initiate the parallel application (startup processing and the loading of subsequent processes), the Earl system utilizes an Executive Process (EXEC) to help establish these portability issues. In the Earl system, the EXEC is loaded by the original program's host process. The EXEC's main responsibility is to perform all startup processing; that processing critical in establishing program portability. The EXEC is loaded on the first available processor; but not that of the host process (i.e. not the host processor). This is due to the requirement of many distributed systems that the host process be solely resident on a special host-like processor; that capable of performing special tasks such as file I/O, or having special restrictions such as a (reduced) maximum length for host destined/originated messages. The following two sections describe how the EXEC process helps to establish portability for target systems with various startup/initialization methodologies.

#### 3.1.2.1.1. Traditional Startup Schemes

In most distributed systems, a program's host process is responsible for loading subsequent processes via one or more system load requests. Since this processing is critical to program portability, the Earl systems requires all such processing be done by the EXEC process. Figure 2 illustrates the role of the EXEC process.

The host process first issues a *load request* message to the EXEC (instead of issuing the request directly via a system call) and remains suspended. We can assume for the scope of this paper that the host is loading all nodes with a *single* load request; individual requests (one for each node) requires similar but some additional



**Figure 2. The Executive process is responsible for initiating processes and mapping a source machines communication ids to that of the target machine.**

processing. Following the load request message, the EXEC then performs the system loading for all processes. During this loading phase, a system's process ids are usually created; establishment of these ids is an important portability issue critical to both communication and non-communication processing. For example, all communication primitives utilize some sort of id mechanism for specifying a source or desination process; mapping such primitives between machines also requires mapping these id values (examples of some id mechanisms are combining a group name with integer ids, postive integer values, specific integer range, etc). Additionally, programmers

*December 22, 1993*

often depend on these id values for *non-communication* processing; *requiring* the ids to be within the source system's specifications. The following example illustrates the importance of mapping ids from the source system for non-communication processing:

---

```

main() {

    Array[NUM_PROCS];
    .
    .
    recv(type, buf, len);

    /* Keep track of total length of msgs from each node */

    x = source_of_msg();
    Array[x] = Array[x] + len;

    /* The return value from the system call was expected */
    /* to be within a certain range.                */

}

```

The example illustrates the dependency between the original program code and the communication ids of the source system. The program is developed with knowledge of the system's id range (i.e. 0 ->  $n - 1$ ); porting to a machine which uses arbitrary integer values (e.g. PVM) can result in quick program termination. To avoid such errors and to establish communication portability, the EXEC is responsible for creating and maintaining a table which maps the target machine's ids to that of the source machine. A message containing this lookup information is then sent to each process. Having this complete id set available to each node during initialization eliminates the need for a process to query (i.e. send additional messages) to the EXEC process during program execution.

Following receipt of the communication id lookup table, each node then sends a *node complete* message back to the EXEC. Upon receiving all node complete messages, the EXEC issues the final *load complete* message to the original host process; allowing it to continue execution. Note that the ordering of events by the EXEC is very important. Recall that the original host process (after sending the initial request to load message) is blocked until all nodes have been initiated and received communication ids. Since the original host process is no longer actually initiating each node (i.e. it issues the EXEC to do it), continuing its execution after the EXEC load request could result in the host sending a message to a node not yet loaded. Some environments (e.g. PVM) are sensitive to such issues and may yield system errors (rather than simply ignoring the sent message).

Coordinating the Earl EXEC process with node programs (to establish the necessary process synchronization and communication portability) can be achieved with little impact to execution of the original program. Most systems require each process to perform special (system) registration/ de-registration processing; processing which is usually performed at the very beginning and end of each process. Porting a program from a system which does *not* require it to one that does, combined with Earl's to coordinate the EXEC process, had lead to the following design convention:

Original program	After mapping
<pre>main() {     /* no system calls made */     x = y;     printf("%d0,x); }</pre>	<pre>main() {     /* Generic "Initialize" substituted in */     Generic_First_Call();     x = y;     printf("%d0,x);     /* Generic "Finalize" substituted in */     Generic_Last_Call(). }</pre>

The *Generic\_First()* and *Generic\_Last()* routines are placed at the very beginning and end of a process's program (or are substituted for a program's original system calls); they help to accomplish two important tasks: perform all system registration/de-registration which may be required on the target system (but not on the source machine) and to receive and send appropriate initialization messages to and from the EXEC (those message necessary in establishing the process synchronization and communication ids mentioned above).

The design and role of the EXEC allows the addition of any new messages within the original program to occur only at startup. This is necessary since messages which are introduced throughout the execution of the program could add difficulty to such debugging issues as race detection and message tracing. Combining low additional messages with the fact that the EXEC process is confined to the *non*-common case mapping, helps both the common and non-common case implementation to remain efficient.

#### 3.1.2.1.2. Implicit and Interactive Startup

In some distributed systems, the loading of subsequent processes is implicit to the developer. For example, in the MPI system, developers need not make any explicit system calls to load or initialize subsequent processes; this is done within the *MPI\_main()* function. Additionally, some systems (e.g. iPSC/2) provide the user with the option of initiating processes interactively within the distributed environment (instead of placing the loading or startup commands within the program code). When porting from such systems to one which requires explicit loading calls, the Earl system must *detect* (within the host process) the need for these node processes to be initiated, and subsequently perform the loading. Having Earl simply place the load command(s) within the source program's host process is fine for systems with implicit loading (since this is guaranteed to be the only loading command), but this technique may yield multiple (unnecessary) load commands in environments which provide the *option* of interactive loading.

Earl accomplishes portability between these and traditional startup schemes by monitoring all system calls of the host process and determining whether subsequent processes *should have* already been started. For example, the first instance that the host is to issue a send or receive operation (or any operation relating to communication,

December 22, 1993

such as probe or any routine which queries information about a pending or received message), Earl will invoke the EXEC process with the *load request message*. All subsequent loading and initiating of these processes (and the establishment of communication portability) is then accomplished by the EXEC in the same manner as in traditional loading schemes (Figure 2).

### 3.1.2.2. Mapping Send and Receive Operations

Perhaps the most important portability issues (for both semantic and efficiency reasons) lie within the time critical send/receive operations. Although all systems provide some sort of communication operations, many differ in the methodology in which they are implemented. For example, a developer may utilize many different primitives on the source machine which could be achieved with a single operation on the target machine. Additionally, message-passing systems differ in the types of operations they provide (e.g. synchronous, buffered, non-buffered). Thus, the challenge for Earl is to allow the communication routines for various machines to become portable; the generic library must ensure that this portability is not achieved at the expense of efficiency, a program's semantics, or the structure of the Earl library itself. The following sections discuss how Earl achieves these goals.

#### 3.1.2.2.1. Implementation Methodology

One aspect of communication which may be implemented differently on various machines is *buffer packing* (currently, Earl address only contiguous packing mechanisms). Buffer packing characterizes the method a buffer can be created (as for a send operation) and how it can be extracted (as for a receive operation). One traditional mechanism allows the user to simply provide a single starting address and length (of the contiguous message data) within the *send* operation. Another method for buffer packing is the "scatter/gather" technique. In such a system, data is gathered from the message buffer on the sending process, and subsequently scattered into a buffer on the receiving process. The data is contiguous but may be of mixed data types. The scatter/gather technique is implemented with multiple primitive operations; allowing the user (in the case of a send) to clear, pack, and ultimately send the buffer. Porting programs from a traditional packing scheme to a scatter/gather technique requires that Earl perform any necessary packing/unpacking. For example:

December 22, 1993

Source program	Send operation after 1st level mapping
<pre>main() {     .     /* Traditional contiguous buffer */     /* packing program.          */     csend(msgtype, buf, len, source, pid);     .     . }</pre>	<pre>firstlevel_send("original parameter list") {     .     .     /* All three generic operations are required */     /* when porting to a scatter/gather buffer */     /* packing system.          */     .     Generic_clearbuffer();     Generic_Buildbuffer(buf, len);     Generic_BBSend(buf, len, dest, mtype, pid);     .     . }</pre>

The original *send* operation is mapped to the three generic operations at the first level. These routines provide all the necessary functionality when porting to *any* scatter/gather implementation. The functionality of all three routines may not always be needed, but must be provided to handle various implementations. For example, some scatter/gather systems may not require a *clearbuffer()* operation prior to sending a message, but its absence from the library when porting to a particular system (e.g. PVM) could result in the new message data being appended to the end of the previously sent message. Generality is a necessary characteristic of the Earl generic library; this allows various implementations to be mapped between one another via our single set (avoiding the need to map the primitive operations of each machine to every other machine). One of the challenges of this, however, is keeping the common case mappings efficient. In the above example, only the original send operation would be necessary when mapping to and from the same machine. Earl achieves efficiency while maintaining generality (efficiency for the common case and generality for cross machine mappings) by having the generic *clearbuffer()* and *packbuffer()* be null routines in the libraries of all target machines which do not require this functionality (i.e. all systems utilizing the traditional buffer packing mechanism which are mapped in the common case).

Mapping buffer packing mechanisms in the reverse direction (*from* a scatter/gather technique to one which only requires a single send operation) involves similar processing; the buffer packing primitive(s) of the source program need to be simulated by Earl to provide the necessary starting address and length of the contiguous data. To achieve this, dynamic memory allocation is needed to create a single contiguous buffer large enough to hold the entire message. Allocation of memory is generally not a problem, particularly for blocking or synchronous send operations; memory can be released immediately after issuing the send (since the buffer has been safely sent or copied). Asynchronous send operations, however, may require Earl to monitor (within the same process via non-blocking communication routines) whether the message has been sent. Only then can the allocated space be freed.

Systems which utilize the scatter/gather buffer mechanism generally do not specify (within the *recv* communication primitive) the length of the expected message; the data is usually extracted using one or more unpacking

December 22, 1993

operations. Mapping such communication routines to a system which requires a specific length within this operation requires that Earl (in addition to packing and unpacking messages described above) determine the length of the packed message. This is accomplished with the following generic operation:

```
Generic_Getpnd_Length(msgtype) {  
  
    int length;  
  
    /* Wait for specified message */  
    blk_probe(msgtype);  
  
    /* Now determine length of pending message */  
    recvinfo(&length, &source, etc);  
  
    return(length);  
  
}
```

The original receive operation can now be mapped to the Generic library in the following manner:

original program	Same program after mapping
<pre>main() {     .     .     .     /* No length specified */     recv(msgtype);     .     . }</pre>	<pre>main() {     int len;     .     .     /* Determine length of the pending message */     len = Generic_Getpnd_Length(msgtype);      Generic_Block_Recv(buf, len, msgtype);     .     . }</pre>

The *Generic\_Getpnd\_Length()* routine is another example of the need for the Earl Library to be a functional super-set for all supported machines. In this particular example, the generic operation allows programs of various buffer packing methodologies to be mapped to and from our single generic set.

#### 3.1.2.2.2. Communication Modes

When porting communication routines (send/receive) between machines, it is important to understand the characteristics (modes) of those operations for both the source and target machine. Communication modes and characteristics refer to such things as execution flow (such as suspending execution until an operation has completed) and message buffering (such as preserving the message buffer from corruption). Not all machines support all types of send/receive operations; porting an operation of one type to one which has different characteristics on the target machine may cause program ambiguities to occur (e.g. program deadlock, buffer corruption, etc). Thus, it is the responsibility of the Earl Library to map a specific operation to the generic library, and ensure (in the cases

where the semantics of the original operation are not directly supported on the target machine) that those characteristics are supplied to prevent any program degradation. The following two examples illustrate this point.

*Example 1.* A *synchronous* send operation is one which suspends execution until the message being sent is received by the designated process (i.e. the sending process receives an acknowledge from the destination process). An *asynchronous* send operation is one which *may* suspend until the message buffer has been cleared (i.e. until the message has actually been sent) but does not wait until the message is received by the designated process. Mapping a program containing asynchronous operations (via Earl) to a system utilizing only synchronous routines requires that Earl provide the needed functionality to avoid the potential of program degradation, such as deadlock:

Program using asynchronous sends

Program mapped to synchronous sends

```
/* The sends do not wait for */
/* the message to be received */
```

```
Async_send(type1);
Async_send(type2);
```

```
/* The recvs can be posted */
/* in any order.          */
```

```
recv(type2);
recv(type1);
```

```
/* The send now waits until */
/* the msg is received      */
```

```
Generic_Sync_Send( type1, );
Generic_Sync_Send( type2, );
```

```
/* The recv() for type2 will */
/* wait indefinitely.        */
```

```
Generic_Receive( type2, );
Generic_Receive( type1, );
```

One can see that if the asynchronous sends of the source machine are simply mapped to a target system's synchronous operations, there is the potential that the second message (type2) may never be received (resulting in program deadlock). This can be avoided by modifying the *Generic\_Receive()* operation to ignore unintended messages (i.e. put them back into the system message buffer) and return only when the intended message has been received:

```

Generic_Receive(buf, len, msgtype) {
    while (true) {
        if "msgtype" arrived
            return that message;
        else
            recv(any message type);
            put this message back in message buffer;
    }
}

```

*Example 2.* Similar risks exist when *buffered* and *non-buffered* send operations are mapped to one another. A system which uses a buffered send operation is one which suspends execution until the data has been safely placed in the system buffer (but not necessarily received by the designated process). Users often rely on this technique and freely modify (immediately after the send operation) the message buffer. A non-buffered send operation is one which returns immediately after initiating the transfer to the destination process (the iPSC2 *isend()* is one such example of a non-buffered send operation). There is no guarantee, in this case, that the message has cleared the buffer; users often depend on other communication routines such as *probe()* or *msgdone()* to determine whether it is safe to modify the message data. Mapping the buffered send operations of a program to non-buffered sends on some target system presents the possibility that the message buffer will be corrupted before it is actually sent. For example,

Program using buffered sends	Program mapped to non-buffered sends
<pre> main (); . . x = mynode(); send(&amp;x, len, msgtype) {  /* user assumes x is already sent and */ /* available for re-use.          */  x++;  } </pre>	<pre> main (); . . x = mynode(); Generic_NonBuf_Send(&amp;x, len, msgtype) {  x++;  /* The message buffer may not be free */ /* and the "x++" value may be sent. */  } </pre>

To avoid such a situation, the Earl system maps the source machine's buffered send to the *Generic\_Buf\_Send*; this

December 22, 1993

operation may use the *non*-buffered send operation of the target machine, but Earl manually buffers the message within the generic routine to avoid the possibility of data corruption (i.e. the data is copied to a temp buffer to allow the user to freely modify data at the original address). Data corruption is not a concern for Earl when porting synchronous send operations; this is ensured by the inherent characteristics of the operation (i.e. waiting for the message to be received by the designated process).

It is the responsibility then for Earl to ensure that the semantics of the original program are preserved. Currently, the Earl library supports blocking and non-blocking send operations but only blocking receive operations. Support for non-blocking receives is complicated by limitations on some of the target machines. This processing could be provided within the generic library to allow portability to all target machines (even ones which do not directly support it). However, supplying this functionality would involve heavy message buffering; doing so in a portable environment (where target machines have different, and usually unknown buffering capacities<sup>4</sup>) could hinder reliability.

### 3.2. Advantages of the Earl Design

The key component of the Earl design is that the communication primitives of all machines are mapped (via the library) to *one* set of generic routines. This design philosophy has some immediate advantages:

(1). Mapping to a single generic set allows a common place in which to implement debug/performance algorithms; making these algorithms portable. The algorithms can be developed using our generic routines (those which all primitives are mapped to) instead of using the peculiarities of any specific machine. The addition of a new supported machine does not require any modification to these algorithms; since the primitives of the new machine are mapped to the same generic set. Thus, the time consuming task of implementing and analyzing debug algorithms across a collection of architectures is reduced. Additionally, one can quickly determine any sensitivities a debug algorithm's implementation may have on a particular architecture. For example, a common mechanism in *playback* for message-passing programs is to buffer a message that does not arrive in the same order as in the original execution. The debug algorithm may have originally been developed on a system with large buffering capacities. Running the algorithm in a portable environment involves the possibility that the buffering capacities of some of the target machines may not be sufficient. Because the tools are now developed with the Earl generic routines, however, this inadequacy can be discovered quickly (without having to re-implement the algorithm).

(2). Existing programs which have their communication primitives mapped to our generic set not only become portable (by being mapped back to the primitives of any machine), but the debug algorithms incorporated into the same generic set can now be run with any program on any supported machine. Developers of such algorithms now have access to a wider range of quality test programs for which to analyze their debug and performance tools.

December 22, 1993

(3). Mapping to one generic set allows for easy maintenance. Only two library implementations are needed to map a machine's primitives to every other machine's primitives (by mapping first to our generic set). The addition of a new supported machine only requires implementing these two mappings. A mapping scheme which implements mappings directly from each machine's primitives to every other machine's primitives would require  $n$  implementations (a mapping from the new machine's primitives to every other machine's primitives).

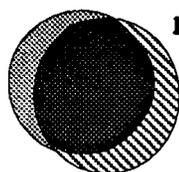
#### 4. Relationship of the Earl Library to MPI

The general design of the Earl Generic Library allows easy mapping of existing programs to the MPI standard (mapping programs *from* the MPI standard onto Earl is equivalently easy, however, mapping *to* a community standard seems much more useful). Once this standard is established, and there exists systems to directly support it, a user can take existing programs (those developed with the communication primitives of a particular machine) and map them (via Earl) onto this standard. Because the MPI standard is not a functional superset for all existing systems (i.e. it has intentionally left some out), mapping primitives directly from an existing program to MPI (instead of mapping first to the Earl Library) is not possible. However, mapping the primitives of Earl to and from the MPI Standard primitives is quite easy; a result of the similar design goals of the two systems.

Recall that the role of Earl (mapping the primitives of all machines to and from one generic set) requires it's generic set to provide a superset of functionality for all supported machines; allowing mapping back to any machine's primitives easy. Also, the generic routines within this set have a very general structural definition, allowing easy mapping from any primitives. The design of the MPI standard is not much different from current practice, and provides the most attractive features of a number of message-passing systems. It is this common functionality of MPI to existing systems, combined with the functional superset and general structure of Earl routines, which yields efficient mappings between the primitives of the two systems. The Venn diagram in Figure 3 helps to illustrate this relationship. The diagram can be applied to the commonalities seen in the *message-passing operations* used by both systems. For example, all (Earl) supported machines combine to use many different modes of communication routines (synchronous, blocking, nonblocking); all of which are popular. Likewise, the MPI standard recognizes the need and desire of programmers for these operations; thus providing the user the option of using all three types. As a result, mapping communication routines (those routines which are the core of any message-passing system) between Earl and MPI is straightforward and efficient. This was demonstrated in a simple mapping between an MPI *ring.c* program and the PVM system; the program was quickly mapped to the Earl Generic library with no structural modifications necessary to any of Earl's generic operations.

The Earl system may prove useful once MPI is finalized and hardware exists to directly support the standard interface. Furthermore, once widely accepted, it is anticipated that libraries of debug and performance tools will be available within the MPI system (i.e. tools will be developed with the MPI primitives instead of using the peculiarities of any particular machine). Users can now apply the advantages of the MPI system (direct hardware

## Earl Generic Library



MPI Standard Library

-  Common functionality between MPI Standard and existing distributed systems (those supported by Earl).
-  Operations of existing systems not desired in MPI Standard.
-  Operations not provided within Earl supported machines (or possibly from any existing system).

Figure 3. Relationship of the Earl Library to the MPI Standard.

---

support, developmental and debugging tools, etc) to *existing* distributed programs by mapping them (via Earl) to the standard interface.

## 5. Results

Tables 1,2 and 3 illustrate the results for our test implementation. The programs were executed with variable factors such as number of processors, input data size, etc. This was done to acquire any correlation between the overhead invoked by Earl and the duration of program execution (due to the lack of extensively long executing programs). Programs were run both in the common case and between different machines. Table 1 describes the common case mappings for PVM programs (running a PVM developed program through the Earl Generic Library then back to PVM). One can see that the overhead invoked by Earl is quite small (< 5%). In particular, programs with low message traffic show a decrease in overhead as program execution time is naturally extended (by increasing matrix size for "chol" and increasing the number of processors for "fft"). One could speculate then that the overhead invoked by Earl is quite negligible for programs of low message traffic; particular for longer executing programs. For programs with heavy message traffic ("puzzle"), increasing their normal execution time (in this

Program	Original Time	Mapped through Earl	% increase	Comments
"puzzle"	1.81	1.84	1.7 %	30 levels, moderate message traffic
"puzzle"	39.32	40.38	2.7 %	41 levels, Heavy message traffic
"fft"	9.56	9.95	4.1 %	4 Processors
"fft"	12.65	12.85	1.6 %	8 processors
"chol"	1.29	1.3	1.1 %	matrix size = 32
"chol"	2.89	2.91	.7 %	matrix size = 100

**Table 1. PVM Programs - Common-case mappings.**

case by increasing the number of levels of the puzzle), shows a slight increase in overhead (although still quite small; from 1.7% -> 2.7%). This may be due to the slight modifications made within the time critical send/rcv operations. However, despite a *significant* increase in message traffic for the "puzzle" program (by increasing the number of levels from 30 to 41), the total execution time only increased by 2.7%.

Table 2 illustrates the results for common case mappings for iPSC/2 programs. As with the PVM common case results, the overhead invoked is very small (< 7%). However, unlike PVM, both programs ("HMult" and "Tester") show a slight increase in overhead as normal execution time is extended (in this case by increasing the number of processors). Since the implementations of the generic library will vary from machine to machine (depending on how easy it is to map to and from that machine), overhead will also vary.

Table 3 illustrates Earls efficiency for cross-machine mappings (mapping a program developed for one machine onto the primitives (via Earl) of some other machine). This implementation mapped iPSC/2 programs to the PVM system; PVM was implemented on a network of SUN SPARC workstations. Although our test results

Program	Original Time	Mapped through Earl	% increase	Comments
"HMult"	2.02	2.08	2.97 %	4 Processors
"HMult"	2.24	2.35	4.9 %	8 processors
"Tester"	2.25	2.33	3.6 %	4 Processors
"Tester"	2.37	2.53	6.8 %	8 Processors

**Table 2. iPSC/2 Programs - Common-case mappings.**

are restricted to running the iPSC/2 programs on a collection of SUN workstations, porting these programs to the PVM system allows them to be executed on any PVM supported machine (e.g. Cray, Connection Machine, etc). The "HMult" and "Tester" programs were developed for the iPSC/2 and primarily provide low-moderate message traffic. Mapping the primitives of these programs to our generic set and then back to the primitives of PVM actually yielded faster execution times. The significant *decrease* in execution times (up to 82%) seems to be due to the low overhead invoked by the Earl mappings combined with the fact that the target machine is better suited for the characteristics of the program. The CPU intensive "HMult" program is able to run on the faster processors of the SPARC5; thus speeding up execution time. A similar correlation can be seen with the intense message-passing characteristic of the "cell" program. Message-passing is certainly faster on the program's source machine (iPSC/2) and thus some overhead is seen when ported to a machine not designed for such processing. This initial 62% overhead could also be contributed to the additional processing invoked by the Earl EXEC process (additional messages in the loading sequence). However, one can see that as the normal execution time of the "cell" program is extended (in this case by increasing the number of processors), the overhead decreased from 62% to

December 22, 1993

less than 6%. The waning of this overhead (as the cell execution time is naturally extended) seems consistent with the fact that the EXEC overhead is confined to only startup processing.

Overall, the programs in this test implementation show a negligible overhead invoked by Earl (< 10%). Although we did not have any programs of significant execution time (hours) to test, increasing their normal execution times with variable factors such as input data size and number of processors, showed a positive correlation between execution time and Earl overhead. Additionally, significant speedup can be achieved when running a program on a machine better suited for that program's inherent characteristics. The lack of significant overhead also helps developers of debugging tools to acquire quality programs for which to test their algorithms; developers are no longer restricted to using only those programs developed for machines currently available to them. The positive results allow those developers to use programs that have been developed for any machine, and run them without gross program degradation (at least no significant degradation stemming from portability, the debug algorithms themselves may invoke additional overhead).

---

Program	Original cube time	Execution on PVM	% increase	Comments
"cell"	11.5	18.67	62.3%	4 Processors
"cell"	38.5	40.8	5.97%	8 Processors
"HMult"	2.02	.38	-81.2%	4 Processors
"HMult"	2.24	.67	-70.1%	8 processors
"Tester"	2.25	.58	-74.2%	4 Processors
"Tester"	2.37	.96	-59.5%	8 Processors

Table 3. iPSC/2 programs - mapped to the PVM system (SUN SPARCs).

---

## 6. System Limitations/Restrictions

When porting programs via the Earl system, users should consider some of the following issues that have not been addressed/implemented (these issues apply only to cross machine mappings, *full* functionality is supplied in all common case mappings):

- Non-blocking receive operations are not supported when porting to PVM.
- Earl has been tested with programs which have consisted mainly of one process per node, and programs in which no more than one process initiates subsequent node programs (when porting to PVM).
- No Real-Time Clock routine has been implemented when porting to PVM.
- The PVM to iPSC/2 mapping has not been implemented.

## 7. Conclusion

Mapping the primitives of all machines to and from our generic set allows existing distributed programs to become portable. Debug and performance algorithms can be instrumented within this set, but applied to programs for many different machines. Additionally, a program can be run on a machine other than the one for which it was originally developed. This is particularly valuable for developers of debugging tools; developers are now provided with access to a much wider range of quality test programs with which to analyze these tools. Program portability is also useful when a program can be run on a machine which is better suited for that program's inherent characteristics. For example, a program designed for heavy algorithmic calculations (and low message traffic) which has been developed on a machine with little CPU power, can now be ported (with low overhead) to a machine with much faster CPU capabilities.

Finally, with the completion of the MPI Standard almost complete, users may depend solely on the standard for the development of subsequent distributed programs and tools; possibly making existing portable systems obsolete. The Earl Library, however, ensures that programs already developed for these tools (and for programs which use communication primitives of any particular machine) do not follow the same fate.

## Appendix: Generic Message-Passing Routines

The following tables list the Earl generic operations currently implemented. These routines form a set of communication primitives which contain functionality common to all Earl supported machines; allowing iPSC/2, PVM, and simple MPI programs to be mapped between one another. The general structural definition of each routine yields a very general parameter list; however, given the small number of machines currently supported, not all parameters are utilized (and are thus not described). Also, each routine may be implemented differently for a given target machine, therefore, the descriptions are kept at a high level and return values are described only where applicable. For ease of reference, the routines are grouped into the following six categories:

December 22, 1993

## Category I

### Process startup, initialization, and termination

---

int **Generic\_LEXEC** (char \*file, char \*arch, long node, long pid)

- Loads the Earl EXEC program on the next available processor (called inherently by original program's host process).

---

int **Generic\_allocproc** (long \*numproc, char \*name, char \*type, char \*srname, long keep, long \*me, long \*host, long pid)

- Allocates a cube or number of processors specified by *numproc*. Returns the id of the calling process.

---

void **Generic\_attachprocs** (char \*cname)

- Attaches to a group of processors of *cname* and makes it the current group/cube.

---

void **Generic\_LastCall** (int release)

- Performs all termination/cleanup processing required by each process. This is the last executable statement in each process.

---

int **Generic\_load** (char \*file, char \*arch, long node, long pid)

- Loads the executable *file* on the given architecture (*arch*). For systems which allow users to specify destination, the file is loaded on node number *node* with a process id of *pid*. Returns the process id for the process to be loaded.

int **Generic\_loadm** (char \*file, char \*machine, long node, long pid)

- Same as *Generic\_load* but initiates the file on the specified *machine*.

---

int **Generic\_FirstCall** (char \*component, long \*numproc, long \*me, long \*host)

- Registers a process within the target system. Performs all necessary initialization processing which includes synchronization with Earl EXEC and receipt of the communication id lookup table during process startup. Returns the process id of the calling process.

---

void **Generic\_relcube** (char \*cname)

Releases the group/cube specified by *cname*.

---

void **Generic\_setpid** (long pid)

Sets the process id for the host process.

---

int **Generic\_termgroup** (char \*component, long instance, long node, long pid)

- Terminates the process(es) specified by *component*, *instance*, *node*, or *pid*. This routine also terminates all pending messages. Returns < 0 if error.

---

int **Generic\_termproc** (char \*component, long instance, long node, long pid)

*December 22, 1993*

- Same as *Generic\_termgroup* but has no effect on pending messages.

---

December 22, 1993

## Category II

### Inter-process communication

---

long **Generic\_Breceive** (char \*buf, long bytes, long msgtype)

- Receives a message of the specified *msgtype* (Blocking). Returns the actual message type.

---

long **Generic\_Breceivem** (char \*buf, long bytes, long \*msgtype, int num)

- Receives a message specified by any of the *num* message type values specified in *\*msgtype*. (Blocking)

---

long **Generic\_NBreceive** (char \*buf, long bytes, long msgtype)

- Non-blocking receive operation. (Not supported when porting to PVM).

---

long **Generic\_BBsend** (char \*buf, char \*component, long bytes, long mtype, long inst, long pid, long node)

- Send a message to process(es) specified by ( *component* and *inst*) or by (*node* and *pid*). (Buffered)

---

long **Generic\_NBNBsend** (char \*buf, char \*component, long bytes, long mtype, long inst, long pid, long node)

- Same as *Generic\_BBsend* but is *Non-buffered*.

---

December 22, 1993

**long Generic\_Sync\_send** (char \*buf, char \*component, long bytes, long mtype, long inst, long pid, long node)

- Same as *Generic\_BBsend* but is synchronous. Not supported on iPSC/2 or PVM.

---

**void Generic\_clearbuf** ()

- Initializes/clears the send buffer.

---

**int Generic\_build[type]** ([type] \*ptr, int num)

- Inserts *num* values beginning at *ptr* into the send buffer. [type] can represent an integer, float, complex, string, bytes, double float, or double complex values.

---

**int Generic\_xtract[type]** ([type] \*ptr, int num)

- Extracts *num* values of datatype [type] from received message and assigns it to *ptr*. [type] is the same as for *Generic\_build*.

---

December 22, 1993

## Category III

### Pending or received message inquiries

---

long **Generic\_Bprobe** (long msgtype)

- Check if message of type *msgtype* has arrived (Blocking). Return value is dependent on target system (actual message type or status).

---

long **Generic\_getmcount** ()

- Returns the length of a pending or received message.

---

long **Generic\_getmpid** ()

- Returns the process id of the process which sent the message.

---

long **Generic\_getmsource** ()

- Returns the node id of the process which sent the message.

---

long **Generic\_getmtype** ()

- Returns the message type of the pending or received message.

---

long **Generic\_getpnd\_Length** (long msgtype)

December 22, 1993

- Returns the actual length (in bytes) of the pending or receive message specified by *msgtype*. This routine is used by Earl when mapping programs from systems using scatter/gather buffer packing mechanisms (where no length is specified within the receive operation) to systems utilizing traditional buffer packing schemes.

---

void **Generic\_msgcancel** (long id)

- Cancels an asynchronous send or receive message specified by *id*.

---

long **Generic\_msgdone** (long id)

- Determines whether the specified asynchronous send/receive operation has completed. (Non-blocking)

---

void **Generic\_msgwait** (long id)

- Waits for the asynchronous message specified by *id* to complete. (Blocking)

---

long **Generic\_probe** (long msgtype)

- Same as *Generic\_Bprobe* but is asynchronous. Return value dependent on target system.

---

long **Generic\_probem** (int num, long \*msgtype)

- Same as *Generic\_Bprobe* but checks for messages specified by any of the *num* message types (*\*msgtype*).

---

*December 22, 1993*

**int Generic\_recvinfo (int \*bytes, int \*msgtype, char \*component, int \*pid, int \*instance)**

- Returns the characteristics of a pending or received message including it's length, type, and sending process. Returns < 0 if error.

---

*December 22, 1993*

**Category IV**  
**Process/environment inquiries**

---

long **Generic\_cubeinfo** (struct cubetable \*ct, long numslots, long global)

- Obtain information about allocated cubes.

---

long **Generic\_cubysize** ()

- Returns the number of allocated processors in the group or attached cube.

---

int **Generic\_groupstatus** (char \*component, int instance)

- Determines whether the indicated process is active. Return value dependent on target system.

---

long **Generic\_myhost** ()

- Returns the node id of the host process.

---

long **Generic\_mynode** ()

- Returns the node id of the calling process.

---

long **Generic\_mypid** ()

- Returns the process id of the calling process.

*December 22, 1993*

---

**long Generic\_nodedim ()**

- Returns the dimension of the allocated cube.

---

**long Generic\_numnodes (int \*nnodes, int \*nformats)**

- Obtains the number of nodes and data formats. Returns the number of nodes.

---

**long Generic\_who (char \*component, int \*pid, int \*numproc, int \*instance, int \*host)**

- Obtains information about the calling process and environment such as component name and instance, node and pid, number of processors allocated, and the node name/id of the host process. Return value dependent on target system.

---

December 22, 1993

## Category V

### Process synchronization

---

int **Generic\_barrsync** (char \*barrname, int num)

- Blocks caller until *num* calls with the same *barrname* have been made. Returns < 0 if error.

---

void **Generic\_readysig** (char \*event)

- Sends signal with the specified name.

---

void **Generic\_wait\_one\_cmplt** (long node, long pid, long \*cnode, long \*cpid, long \*ccode)

- Suspends caller until the specified process has completed.

---

void **Generic\_wait\_all\_cmplt** (long node, long pid)

- Suspends caller until *all* specified processes have completed.

---

void **Generic\_waitsignal** (char \*event)

- Suspends caller until the specified signal name occurs. Used in conjunction with *Generic\_readysig*.

---

*December 22, 1993*

## **Category VI**

### **Miscellaneous**

---

**double Generic\_clock ()**

- Returns the actual elapsed time in milliseconds (since node initialization or elapsed time used by UNIX for the host process). Not supported in PVM.

---

**void Generic\_CTOH[t] (unsigned long \*sv, short n)**

- Swap byte order from cube to host. [t] represents float, long, short, or double values. Not supported in PVM.

---

**void Generic\_HTOC[t] (unsigned long \*sv, short n)**

- Swap byte order from host to cube. [t] represents float, long, short, or double values. Not supported in PVM.

---

**void Generic\_flick ()**

- Relinquish CPU to another process. Not supported in PVM.

---

**void Generic\_flushmsg (long typesel, long node, long pid)**

- Flush specified messages from the system.

---

December 22, 1993

long **Generic\_ginv** (long num)

- Returns the inverse of *Generic\_gray*. When porting to PVM, *num* is returned.

---

long **Generic\_gray** (long num)

- Return the binary-reflected Gray code for an integer. When porting to PVM, *num* is returned.

---

void **Generic\_killsyslog** ()

- Terminate a *syslog* process. Not supported in PVM.

---

void **Generic\_led** (long lstate)

- Turn the node board's green LED on or off (supported on iPSC/2 only).

---

long **Generic\_masktrap** (long mask)

- Enable or disable a receive trap. Not supported in PVM.

---

void **Generic\_newserver** (char \*cubename)

- Start a new file server for the specified cube. Not supported in PVM.

---

void **Generic\_setsyslog** (long stdfd)

December 22, 1993

- Start a syslog program. Not supported in PVM.

---

### References

- [1] *The Express Parallel Toolkit*, ParaSoft Corporation ().
- [2] *iPSC/2 User's Guide (Preliminary)*, Intel Scientific Computers (March 1988).
- [3] A. Beguelin, J. Dongarra, G. A. Geist, R. Mancheck, and V. S. Sunderam, "A Users' Guide to PVM: Parallel Virtual Machine," *Technical Report ORNL/TM-11826*, Oak Ridge National Laboratory, (July 1991).
- [4] J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker, "A Proposal for a User-Level, Message Passing Interface in a Distributed Memory Environment," *Technical Report ORNL/TM-12231*, Oak Ridge National Laboratory, (June 1993).
- [5] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "A Users' Guide to PICL: A Portable Instrumented Communication Library," *Technical Report ORNL/TM-11616*, Oak Ridge National Laboratory, (September 1992).
- [6] W. Gropp and E. Lusk, "A Test Implementation of the MPI Draft Message-Passing Standard," *Technical Report ANL-92/47*, Argonne National Laboratory, (December 1992).
- [7] A. Skjellum, N. E. Doss, and P. V. Bangalore, "Writing Libraries in MPI," *Mississippi State University Computer Science Department & NSF Center for Computational Field Simulation*, ().