LAM: An Open Cluster Environment for MPI

Greg Burns
Raja Daoud
James Vaigl

Ohio Supercomputer Center
Columbus, Ohio

Introduction

LAM is a parallel software environment with a history. The most recent episode is MPI (Message-Passing Interface) [1], the standard library definition for message-passing communication. A standard is long overdue as many new programmers discover that explicit data transfer with clear synchronization can be an effective way to get something done with parallel computers. MPI anticipates the day when libraries (probably refitted as objects) exploiting multiple computing resources will be ubiquitous. Delivering potentially complex, well behaved and portable parallel software requires more than functions to send and receive a message. These goals justify an MPI design that may at first seem burdensome. This paper does not pursue an MPI design review but instead describes the unique aspects of one implementation.

The popularization of message-passing libraries follows the emergence of the multicomputer architecture as a popular solution for large scale parallel computers. If the multiprocessor[1] style had fared better, perhaps PVM [2] would be a semaphore and thread library. If SIMD machines had conquered the world, perhaps data parallel languages would be a few years ahead of their current schedule. Early message-passing libraries followed the model of the first multicomputers. There was a big impressive dedicated box of processors in the machine room front-ended by a familiar UNIX host machine.Applications were hand rolled - no catalogue of parallel libraries. The programmer was usually aware of the topology of processor interconnection. Hardware, library and application software anticipated orders of magnitude scalability. This is how we define MPP parallelism.

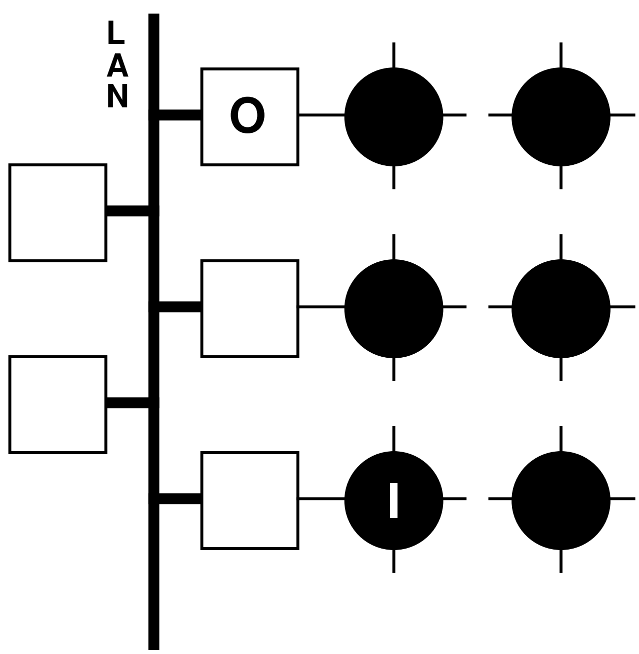
LAM [3] started out in that environment and its native message-passing library derives from the requirements of MPP machines and applications. In the meantime, the multicomputer installed at most businesses and institutions, the local area network of workstations, began to look more attractive in performance and price. Areas where these workstation clusters fell short of MPP capabilities simply became less important. Message-passing libraries reflected this change. PVM does not address topology, an essential element in high end scalability.

We discuss the LAM design by painting the big picture that includes MPP and then paring it down to the cluster architecture, where the relevant modules are described in detail. After a general commentary on the choice of synchronization rules, we show how LAM's answer was extended to host other paradigms, first PVM and then MPI.

Trollius Roots

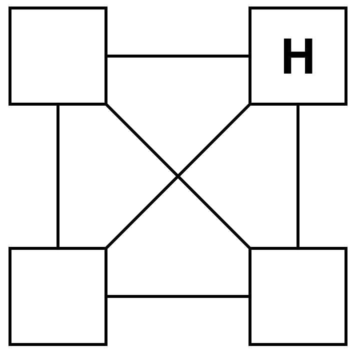
LAM is a programming environment and development system for a message-passing multicomputer constituted with a UNIX network. It is a subset of a greater Trollius [4] system which extends the software to dedicated parallel computer ensembles with no other native operating system.

Trollius delivers power to the user through extensive control and monitoring functionality in a highly layered software structure. You can control details at the lower layers or confine yourself to the comfort and safety of a higher layer. Most importantly, you can decide the balance of performance, functionality and programmer productivity that is best for a particular application. This



Trollius Architecture - Trollius is a uniform foundation for message passing on every multicomputer node. OTB nodes (O) run Trollius on the local operating system while ITB nodes (I) run it natively.

1. I am not considering symmetric multiprocessor products as large scale high performance computers.



LAM Architecture and Topology - LAM is the subset of Trollius that applies only to host nodes. Its fixed topology is fully interconnected.

open architecture design makes Trollius extensible, reconfigurable, and a good foundation for development.

Foundation is the word which best describes the role of Trollius in the total multicomputer software inventory. It is an operating environment that focuses on the basics, the nuts and bolts of parallel processing. Compilers, CASE tools for parallelism, paradigms and abstractions of all sorts need a good foundation. Trollius was not adapted from another computing solution; it was specifically designed for multicomputer applications.

Trollius presents a broad view of the multicomputer architecture. It recognizes that any computer reachable via communication link(s) is a potential participant in a message-passing application. The software presents a uniform programming environment across general purpose computers (e.g.: workstations) and dedicated highly integrated computer ensembles. The latter is usually singled out and tagged as the "parallel machine" and is packaged into an impressive looking box. In Trollius, we refer to these computers as Inside The Box computers or ITB. We recognize these characteristics of ITB computers:

- They are dedicated to single users.
- All incoming and outgoing links are dedicated to single users.
- They are great in number.
- They are going to be used for large parallel computational problems and not the common tasks that general purpose computers do well.
- On average, they have few peripherals.
- They will run Trollius natively.

Since Trollius is not going to provide all the services of a general purpose, multi-user operating system, the ITB computers must be hosted by at least one computer with such an operating system. The host computer(s) will have one or more communication links to one or more ITB computers. As indicated above, other general purpose computers may participate in a multicomputer if the communication links exist such that all computers of both types are reachable (directly or indirectly by hopping through intermediate computers). In most cases, general purpose computers are reachable via a Local Area Network (LAN). In Trollius, we refer to general purpose computers as Outside The Box computers or OTB. We recognize the following characteristics of OTB computers:

- They are shared among multiple users.
- Links to other OTB computers are shared among multiple users.
- They may control peripherals needed by a parallel application.
- They will run Trollius as an application of the native, general purpose operating system.

When running Trollius, we refer to ITB or OTB computers as nodes. Very often we do not distinguish between node types when describing functionality.

Fundamental Identifiers

LAM is a special brand of Trollius. LAM runs only on a network of UNIX machines acting as OTB nodes and does not include the modules that support ITB nodes. These machines can be connected via a local area network (LAN) or be remotely accessible via Internet. LAM presents the topology of the multicomputer as a fully connected graph thus allowing nodes to communicate in 1-hop distances.

The basic objects of a LAM multicomputer are nodes (physical computer), processes (programs running on computers) and messages (data transmitted from one process/node to another process/node). To provide basic synchronization, process management and file access, a few identifying numbers are required. We use these numbers repeatedly throughout this paper.

Every multicomputer node has a unique number called the node identifier. The primary synchronization of all basic LAM message communication is the node ID. All of the network process management functions take a node ID as the first argument. Network filenames can be prepended with a node ID, to specify the node on which they reside. Although higher level paradigms can and do abstract the multicomputer in various ways, basic LAM is a very node oriented environment.

Every LAM node has a multi-tasking local environment. Each process has a locally unique identifier (the UNIX process identifier is used). Many LAM process management functions require a process ID argument.

Due to the multiple process capability on each node, a process and a message must have a way of synchronizing that avoids other processes and messages on the same node. For this purpose, all LAM messages are labelled with a node ID and another number called an event. Events are chosen by the user and are only considered within each node. Thus, a message is delivered to the right node and then to the right event. A process receives a message by specifying an event.

Equal events synchronize. The event has an associate, subordinate object called the type which is found wherever the event is found. It is used for more complex synchronization beyond the event. It is irrelevant if the event does not synchronize. It is irrelevant if it is set to zero. Non-zero types that share a common one bit (bitwise AND is non-zero) synchronize.

Micro-kernel Heart

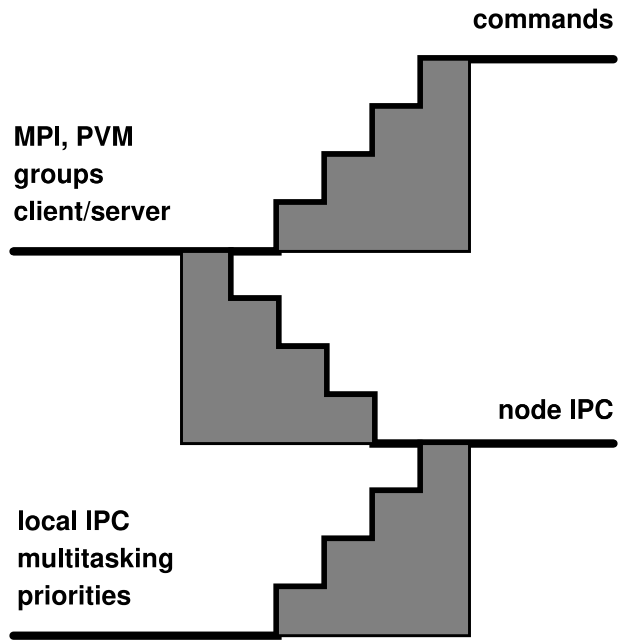
LAM has a highly modular micro-kernel, multi-task structure. Although the environment runs as one UNIX daemon per host, the daemon is really a micro-kernel plus a dozen or so "hand-threaded" system processes. The micro-kernel schedules these internal processes plus any number of external processes (UNIX clients on the same host) in a prioritized preemptive fashion. Internal processes are simply called, while external processes, typically application processes, are replied to through UNIX. The small micro-kernel (more like a nano-kernel) also handles communication between these local processes. Internal system processes, not the micro-kernel, handle network communication. Any internal program can be recompiled and run externally with no change in system behaviour but with an obvious degradation in performance. Indeed, all of the internal processes can be run as separate UNIX processes with the central daemon consisting of only the micro-kernel. This can be a useful configuration for debugging system changes.

LAM enjoys the benefits of modular micro-kernel design without the performance disadvantages. It is trivial to add or subtract services from LAM without seeing or recompiling the rest of the system. This is our justification for calling it an "open" system.

By replacing a couple of modules full of UNIX system calls with a couple of modules full of software interrupts, context switchers and peripheral drivers, LAM can be run natively on your favourite killer micro as an MPP OS with a real-time system on each MPP node.

Network Superhighway

Point-to-point LAM message passing deals with physical nodes explicitly, as opposed to a global task identifier space. The first thing you put in a LAM message is the node ID. Then you worry about which process on the

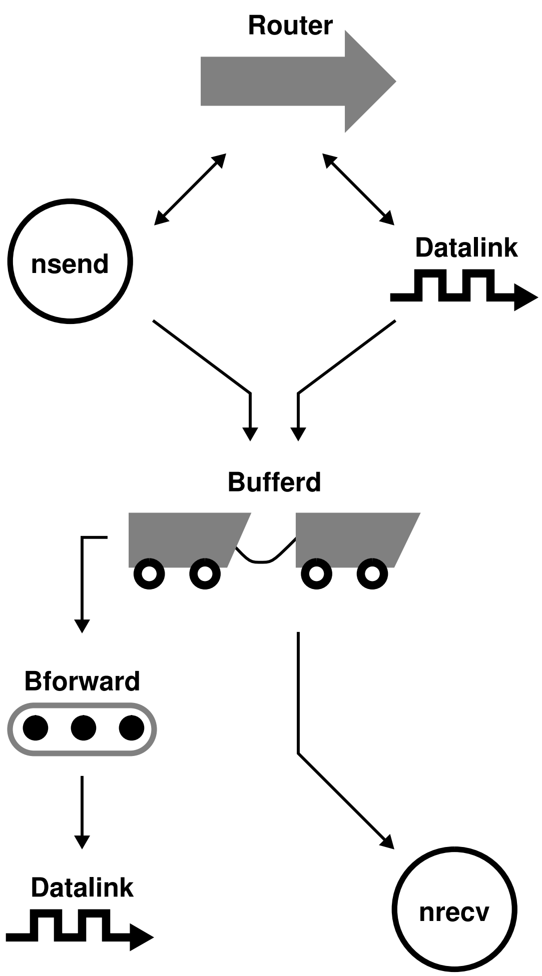


Layered Design - LAM is rigidly layered starting with a micro-kernel with purely local knowledge. Along with network IPC it insulates everything above from heterogeneous platforms.

node will receive it with the message event and type. Many would see this explicit nature as a disadvantage but we do not. Node oriented communication is quite intuitive for the novice user that understands workstations and his application, but not too much about process groups, bulletin boards and other paradigms. For example, a tuple space user will not care which nodes contain which tuples, but the system developer will care. All parallel environments eventually decide explicitly which node gets a message. LAM simply provides a complete API for this layer.

We think LAM's modular structure combined with its node based communication make it the best foundation for building higher level tools and paradigms. We've used it to implement PVM, MPI and a tuple space tool.

A route daemon on each node is consulted to learn the output link on which a message should continue its journey to a specified destination node. In LAM's fully interconnected topology with typically only one network connection, this is a simple question. Although programmers see the multicomputer as having separate links to every other node, LAM (de-)multiplexes these messages through one pair of datalink processes per shared communication link. The datalink processes use the UDP/IP protocol, implementing their own flow control and error correction. They present a flow controlled, reliable communication channel to the rest of the system. For performance reasons, the datalinks have a configurable number of buffers, invisible to the user. The single datalink pair guarding a shared link has a notable weakness. With limited LAM synchronization, a datalink output process cannot selectively accept messages bound for some nodes and not others when the number of nodes is not limited. If one other node stops receiving, the blocking protocol may cause all other nodes to stop sending anywhere. Higher level flow control protocols fix this problem, but LAM does impose these at a low level, preserving performance at the expense of a little danger.



Network Forwarding Processes - After consulting the router, the datalink input, or another process calling nsend(), forwards a message to the buffer daemon. The buffer daemons waits for queries from processes calling nrecv() before delivering the message. Protocol and debugging messages from the buffer daemon move through a worker process before transmitting through the datalink output.

Buffer Mania

The datalinks and route daemon are a working network message-passing system by themselves. Many parallel programmers have come to rely upon extensive message buffering. LAM's fully interconnected topology does not require them for breaking routing deadlock. For stateful programs, non-blocking message send, overlapped communication, debugging and other reasons, LAM has a powerful message buffering system.

The LAM buffer daemon collects messages that arrive on a node and stores them for forwarding to local processes or inspection by a debugger. Local processes treat the buffer daemon like a database, sending queries for messages with the spartan LAM synchronization variables (message event and type), or more complex synchronization fields that we fondly refer to as "SQL". The query may be a probe, a non-blocking receive or a blocking receive. LAM synchronization clients block directly on their synchronization variables in the micro-kernel. SQL synchronization clients block on a client/server protocol with the buffer daemon and therefore experience a bit more overhead.

An asynchronous flow control mechanism is triggered when the total buffered byte count passes a certain point. This causes a system message to be sent to the datalink output processes on all other nodes with instructions to stop accepting user messages (positive message events) for transmission. There is a time lag here that scales with the number of nodes. During this time user messages may continue to arrive and increase buffer levels. Eventually a hard limit in the buffer daemon is reached and the daemon itself stop accepting anything but control messages (debugging queries, flush commands, etc.). The blocking protocol of the datalink processes plugs the link and the process(es) producing all these messages blocks[1]. For this asynchronous flow control mechanism to remain effective while the number of nodes increases, the topology has to be scalable (i.e.: the node degree is fixed).

The buffer daemon never blocks while sending a message. When it needs to respond to a debug client or send the flow control message to its neighbours, it buffers the message for a helper process, called bforward. Bforward sits in a loop receiving a message and sending it to the right datalink process.

1. All of this process blocking and message filtering comes courtesy of the micro-kernel.

LAM has a variety of point-to-point messaging choices.

- packetized, traced, buffered, blocking
- flow controlled at the user level
- virtual circuit, bypassing LAM with TCP/IP
- asynchronous receive, getting an interrupt when the message has arrived
- novice, reducing synchronization to node ID only
- receive probe, testing if a message can be received without blocking and finding its length

Processes can organize into groups and forget about the existence of nodes, much like other process-oriented message-passing libraries. A full range of collective communication functions is available, including group receive with reduction functions, group exchange, a eureka function, and a separate quorum function that blocks the caller until a certain group membership level is reached.

Group membership is dynamic and the programmer is warned to proceed with caution. Some people think that dynamic groups are great for load balancing and other worthwhile pursuits. However, it is impractical to get a distributed implementation that is air-tight. MPI faces this problem at a fundamental level, and is only tolerable with static group membership. PVM's barrier function is vulnerable to race hazard when groups are dynamic.

Client/Server

Upon this network superhighway, we sprinkle whatever servers are necessary for multicomputer operations. Client requests come from anywhere in the network and are packaged on the client side into proxy functions which all accept a node identifier as the first parameter. Occasionally, a service only makes sense within the local node or intends to abstract the concept of nodes, so its proxy function does not have the node identifier parameter. Two remote servers central to LAM operations are the process manager and the file manager, called kenyad and filed respectively.

Kenyad's main responsibilities are starting, monitoring and signalling processes. LAM handles the three-node process creation case of user's node, program's node and target node. If the program and target node are different, the program will be transferred to the target node. The program is located by searching directories in the user shell's path variable on the program node. LAM has its own signal set, much like UNIX, that kenyad delivers to its child processes when requested. The most common signal causes a process to exit. Processes started by a local shell could choose to be independent of kenyad.

The file daemon provides remote UNIX file access with the standard POSIX interfaces with a LAM prefix. For C there is a stdio library with prepended symbols that can switch between local UNIX access and remote LAM access. After much thought, we decided for this total separation between native and LAM file operation. LAM has no bearing on any UNIX operation.

LAM supports a loosely synchronous, parallel I/O access method based on Cubix [5], a research development at the California Institute of Technology. One process on every node must participate in Cubix accesses. A file function is not completed until all client processes have called the same file function. There are two different Cubix access methods that solve two common file read/write problems in data parallel programming.

In Single mode all clients execute the same file function with the same amount of identical data. Read data is broadcast to all and write data is transferred from one arbitrary client. This is useful when all clients want to read a global value from a file, or write a global value to a file. It is especially convenient during output to a terminal. All nodes print an error message and it appears once on the terminal.

In Multiple mode all clients execute the same file function with different amounts of different data. All the data from all the clients is transferred, but the order of transfer is strictly controlled. By default, the lowest value node ID will transfer first and the sequence continues until the highest value node ID transfers last. This is useful in decomposing a data structure during a read so that the right nodes get the right subset of data and in recomposing a data structure during a write so that the data structure is not jumbled.

Without Cubix file access, an application often needs a controlling program to manage the parallel processes and filter I/O. Cubix can eliminate the need for a control program. The non-Cubix default access method is called the Asynchronous method. Without synchronization, a message written by N nodes appears N times on the terminal. A decomposed data structure written to a file appears in a random order.

User Interface

Where there are remote services there are often command programs, executed from any shell. The LAM commands represent its user interface. The important subset of commands manipulate the nuts and bolts of multicomputing: processes and messages.

```
% loadgo n0 -v testsend
[1] 8013 testsend running on node 0
```

Assume that the testsend program sends a message to node 1. Due to the presence of buffers, it runs to completion without blocking.

```
% bfstate n1
NODE  DEST  EVENT  TYPE  LENGTH
n1    n1    6      0     1024
```

The message remains in the buffer and can be inspected. Assume that the testrecv program receives a message. If it blocks because no synchronizing message has arrived on its node, we can see the process.

```
% loadgo n1 -v testrecv
[1] 108 testrecv running on node 1
% state n1
NODE  INDEX  PID  KPRI  KSTATE   PROGRAM
n1    [1]    108  0     BR(7,0)  testrecv
```

```
% lamclean
```

The information under the KSTATE heading indicates that the process is blocked in a receive function on the message event 7 and type 0. This type of isolated debugging is possible because of LAM synchronization on nodes, not processes. The message and the process can be removed, the bug corrected and the programs rerun.

Session Initiation and Termination

A simple text file holds the names, one per line, of the UNIX machines participating in the LAM multicomputer. The file is formally called a boot schema. Three separate tools parse the boot schema and act upon it. The recon tool verifies that each LAM node can be booted. Several conditions must exist before a machine can remotely run the LAM software.

- The machine must be reachable on the network.
- The user must be able to remotely execute on the machine with rsh. Remote host permission must be provided in either /etc/hosts.equiv or the remote user's .rhosts file.
- The remote user's shell must have a search path that will locate LAM executables.

The lamboot tool starts a LAM session for the individual user. IP port numbers are dynamically allocated and part of lamboot's job is to give each node port number information about everybody else, thus enabling the fully interconnected topology. The wipe tool terminates a LAM session.

```
% recon -v <boot schema>
% lamboot -v <boot schema>
% wipe -v <boot schema>
```

Bolstered Synchronization

The message/process synchronization discussed in an earlier section is close to the most primitive we can imagine for multiple processes on multiple nodes. If the LAM message type is removed, leaving only the node ID and the message event, we are willing to submit that this is the lower bound for synchronization in a multitasking multicomputer. Most other message-passing systems at least include the source node ID in the synchronization. LAM does not, for this can be loaded into the message event.

A more general way of looking at message/process synchronization is that you have a certain number of bits and some wildcarding rules. This was the approach used in extending LAM to support other message-passing paradigms. Instead of implementing the new synchronization in a separate daemon with its associated overhead, the extended functionality was placed in LAM's buffer daemon. The new synchronization was not placed in the micro-kernel, thus nothing fundamental has changed in LAM and getting a little extra speed from bypassing the buffers can only be done with LAM messages.

PVM and MPI happen to have the same wildcarding rule a reserved value matches anything, and this rule was adopted for the new synchronization fields (four words of them) recognized by the buffer daemon.

MPI Personality

Message-Passing Interface (MPI) is a formally specified standard produced by an open forum of parallel computer vendors and industry, government and academic users. With so many designers, the resulting library has a distinct sense of a professional API in a way that other message-passing libraries do not. A less desirable outcome of committee design is that MPI also seems to be the union of all possible features. Two features illustrate how MPI is prepared for potentially complex software products in a commercial market. An extra synchronization variable, allocated by the system, is used in all communication. A library from company A communicating between two processes is protected from interfering with another library from company B communicating between the same two processes. Secondly, MPI has extensive data description constructors that can handle arbitrarily complex data structures. Once described, instances of the data structure can be transmitted among heterogeneous platforms with MPI handling the data representation differences. This is more efficient than the buffer packing approach of PVM (although MPI also has that option).

MPI relies completely upon process groups for synchronization, totally ignoring the underlying physical nodes. At the inception of a process, it is a member of the "world" group, which contains all other processes in the job. This implies that all processes in the job are implicitly synchronized, a problem left up to the implementation. LAM has a special command, called mpirun, to start MPI jobs. It requires that all programs be specified in one invocation of the command. To simplify the command line for non-SPMD jobs, the information can be given in a file, which we call the application schema.

Groups can divide and organize into endless subgroups. Source and destination processes are specified by group rank. Before reaching the LAM level, the MPI library resolves the given group/rank into a rank within the world group. In the start-up synchronization with mpirun, each process learns the node identifiers of all the other processes in the job. Thereafter, the world group rank indexes into a table of node IDs for use in the LAM network message send function. Multiple processes within a node are distinguished by using the world group rank as one of the LAM network message SQL synchronization variables. The four synchronization variables in MPI are the source and destination ranks in the world group, the context identifier (the aforementioned system allocated firewall for independent library construction) and the message tag. Equality is the rule everywhere and the source rank and message tag wildcard with a reserved value.

MPI has four different sending modes which map to LAM network services.

- standard mode send - LAM network send
- buffered mode send - non-blocking LAM network send
- synchronous mode send - LAM transport send
- ready mode send - LAM network send

MPI has one receive mode that maps to LAM network receive and a probe function that maps to LAM network probe. All of the send, receive and probe functions have concurrent (immediate, in MPI terminology) counterparts that allows overlap of computation and I/O. The MPI immediate receive does not map to the LAM asynchronous receive. All of the MPI immediate functions are single threaded within the calling process.

Process creation at runtime is almost ruled out by the MPI standard. It can be deduced, but it is not spelled out, that an air-tight implementation must have static group membership (established at job load time) to be in full compliance[1]. Since all processes are members of at least the world group, the only opportunity for creating new processes at runtime is to create new world groups. MPI's intercommunicator technique, a method for communicating between groups, could be implemented for multiple world groups. Without this, there is nothing in MPI to take advantage of LAM's process creation capabilities. We intend to implement this extension as a way of addressing a serious limitation in MPI.

1. We are in favour of this. Static groups are solid and much easier to program.

Much of LAM's value added to the MPI standard comes in the development environment surrounding the programming library. LAM has commands to inspect processes and messages but they do not present their information in terms of group ranks, contexts and message tags. The mpitask and mpimsg commands have the usability of the LAM API counterparts but present information in the MPI sense. Existing LAM commands remove processes or messages.

The extensive suite of collective communication functions, virtual topology mapping, group membership definitions and plenty of other miscellaneous functions are all implemented within the thick library and are completely independent of LAM.

PVM Personality

Putting a PVM face on LAM consisted mostly of trying to map the PVM calls onto LAM calls and also providing services like user buffer management, which really have little to do with the underlying message-passing environment. The buffer pack and unpack routines use the LAM data conversion functions, rather than calling XDR for every data element. Spawning processes, sending signals and performing group operations map onto LAM services very cleanly.

PVM point-to-point message passing maps directly onto LAM network message passing. The three synchronization variables in PVM are the source and destination task identifiers and the message tag. LAM implements the task identifier as a composite of the node identifier and the local process identifier. The message tag is arbitrary, much like LAM's message event. Synchronization is equality for all three variables and the source ID and message tag can be wildcarded (match anything) by a special reserved value. These three variables map to LAM's extended SQL synchronization variables.

A value added aspect of the LAM PVM implementation are the commands for monitoring processes and messages. These commands already exist with LAM but do not present their information in terms of task IDs and message tags. The pvmtask and pvmmsg commands have the usability of the LAM API counterparts but present information in the PVM sense.

Unlike the formal MPI standard, we have taken some liberties with the informal PVM standard. There are some areas where PVM's model doesn't easily fit on LAM, and so were not supported. The first area is user supplied synchronization rules. PVM synchronization beyond the destination process takes place within the application process in the ORNL implementation. This allows PVM to let the user specify a function pointer to redefine the remaining synchronization semantics. In LAM, the PVM synchronization happens in the buffer daemon (where messages are also observable for debugging), a separate process, so a function pointer given by the application is not directly usable. For this reason, and because this feature is seldom used by PVM applications, we decided not to support this PVM capability.

LAM does not currently support the capability to add and delete hosts on-the-fly. A user types a command to start LAM, and the nodes in the machine are specified statically in the boot schema. A casualty of this restriction is the pvm_notify() function, which causes a message to be sent to the caller whenever a host is being deleted by a pvm_delhost() or added by pvm_addhost(). There is also an option to pvm_notify() to send a message to the caller when a task exits. This is not supported because supporting only part of the pvm_notify() semantics is of debatable value and it would require adding more native support in LAM, since only child to parent notification is currently in place.

Conclusion

With its extended synchronization features, LAM is an excellent foundation on which to implement various message-passing paradigms. MPI and PVM are of strategic importance, but any of the presently published libraries could be realised with comparable effort. Whereas MPI has an infrastructure that is prepared for future application development, LAM likewise has an infrastructure that is prepared for future libraries and tools.

Development of LAM was supported by Ohio Supercomputer Center, The Ohio State University, Cray Research and Alex Informatique.

[1] Message-Passing Interface Forum, "MPI: A Message-Passing Interface Standard", (netlib)

[2] Al Geist et al, "PVM 3 User's Guide and Reference Manual", (netlib)

[3] Gregory D. Burns, "The Local Area Multicomputer", Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications, March 1989, ACM Press

[4] Gregory D. Burns et al, "All About Trollius", Occam Users Group Newsletter, August 1990

[5] John Salmon, "CUBIX: Programming Hypercubes Without Programming Hosts", Hypercube Multiprocessors, 1987, SIAM