CONF-9509231--1

LA-UR- 95-2971

TITLE: THE ACL MESSAGE PASSING LIBRARY

AUTHOR(S):

Michael Krogh, CIC/ACL
James Painter, CIC/ACL
Pat McCormick, CIC/ACL
Charles Hansen, CIC/ACL
Guillaume Colin de Verdiere,
Saint-Georges, France

SUBMITTED TO: European T3D Workshop
September 7, 1995
Lausanne, Switzerland



Los Alamos
NATIONAL LABORATORY



Form No. 836 R5
ST 2629 10/91

Rough Draft

# The ACL Message Passing Library

James Painter, Pat McCormick, Michael Krogh, Charles Hansen
{jamie, pat, krogh, hansen}@acl.lanl.gov
Los Alamos National Laboratory
Advanced Computing Lab
Mailstop B-287
Los Alamos, NM 87545

Guillaume Colin de Verdière
Centre d'Etudes de Limeil-Valenton
CEL-V/DMA/AIM
94195 Villeneuve-Saint-Georges, France
coling@limeil.cea.fr

## Abstract

This paper presents the ACL (Advanced Computing Lab) Message Passing Library. It is a high throughput, low latency communications library, based on Thinking Machines Corp.'s CMMD, upon which message passing applications can be built. The library has been implemented on the Cray T3D, Thinking Machines CM-5, SGI workstations, and on top of PVM.

## 1. Introduction

Parallel programs are typically written in one of two styles: SPMD or MIMD. SPMD (Single Program, Multiple Data) programs are typically written in a data parallel language, such as Fortran 90[1]. With such programs interprocessor communications is hidden from the program via the compiler and runtime system.

With MIMD (Multiple Instruction, Multiple Data) programs the programmer must explicitly call message passing primitives for interprocess communications. For such programs to run efficiently and gain the best possible speedup as additional processors are used, the communications cost of the program must be as small as possible. By small, we mean lowest latency and highest throughput. If communications costs are high, then the program will be severely limited in terms of speedup potential as the number of processors increases.

The development of the ACL Message Passing Library was driven by two motivations: performance and portability.

As already mentioned, performance of a communications library is crucial of overall program performance. This fact was made all too clear when we started porting our message passing programs from the Thinking Machines Corp. CM-5 massively parallel computer to the Cray Research Inc. T3D. CRI supplies an implementation of the PVM message passing library for the T3D. The performance of which is poor. We found that our codes not only performed poorly but that they did not scale up (or run in some cases) to large numbers of processors. This was entirely due to the implementation of PVM on the T3D.

Our second motivation, portability, was driven by our investment in CM-5 software. The CM-5 uses a message passing library called CMMD. CMMD is an efficient, simple, but complete, message passing library. In the several years that we had been developing software for the CM-5, we had amass a large collection of libraries and programs based on CMMD and portability was highly desirable.

The remaining sections describe previous message passing systems, describe the implementation of ACLMPL, present timings, describe a few applications that use ACLMPL, and draw conclusions.

## 2. Previous Work

The Parallel Virtual Machine (PVM) library was designed to treat a collection of computers, which may be workstations, servers, vector computers, or even MPPs, as a single distributed parallel computer [PVM]. To accomplish this, PVM supports heterogeneous processors, networks and data types. Besides basic communication primitives (synchronous and asynchronous send and receives), PVM has primitives for process control, synchronization, signaling, process groups, and virtual machine control.

[1] SIMD (Single Instruction, Multiple Data) can be thought of as a more constrained SPMD.

Rough Draft

The Message Passing Interface (MPI) library was designed with efficiency and portability in mind. The MPI feature set was designed by committee which used features and concepts from many various message passing systems [MPI]. What resulted is a "full-featured" message passing library that includes many variations on send and receive (blocking/nonblocking, buffered/unbuffered, receiver-ready, different data types including user specified, and more). Additionally, MPI includes support for global operations (barriers, reductions, gather/scatters, broadcasts, scans, etc.), processor topologies, processor groups, profiling, and error handling. Process management (creation, deletion, migration), active messages, and I/O support are not included in the current standard.

Thinking Machines Corporation created CMMD for the CM-5 massively parallel computer [CMMD]. CMMD supports three styles of communication: synchronous, asynchronous, and active messages (used for event driven applications). The library also includes functions for global operations (reductions, scans, broadcasts, barriers) and I/O. CMMD has no primitives for process control or virtual machine control.

Many other message passing systems exist that provide similar functionality to these three. PVM, MPI, and CMMD are of particular interest to us since they are the "supported" message passing systems for the T3D and the CM-5.

## 3. The Need for Performance

Our software efforts are targeted towards high performance software for MPPs and SMPs (Symmetric Multi-Processors). Our focus is not on harnessing the latent power of desktop workstations. Nor is in running a single program on several supercomputers. Given this, several key differences should be noted between PVM, MPI, and CMMD.

PVM is widely available for most unix workstations and for many common supercomputers and MPPs. It has many basic communications primitives and primitives for process management. PVM's main weakness is that it is not high performance. Past versions utilize a deamon process on each computer node which is involved in communications. Recent versions of PVM allow these deamons to be bypassed; however, performance is still lacking as will be shown.

MPI is a recent message passing system and is not widely available. MPI includes numerous primitives (far more than PVM), except for process management. While efficiency is a main goal for MPI, our benchmarks on the T3D show that it is lacking as well.

Both PVM and MPI also have the goal of supporting heterogeneous data types and computers.

CMMD differs from PVM and MPI in that it is not widely available; however, it is does have a large user base since it is the only supported message passing system available on the CM-5. CMMD has sufficient primitives without trying to include everything. It has the basic communications primitives as well as active messages. It also has the most common global operations.

CMMD was designed for interprocessor communications within the CM-5 and not with processes external to the MPP. This allows for several optimizations. The library does not need to communicate with heterogeneous processors or data types; which avoids unnecessary data conversion and the need for a plethora of different primitives for various data types. CMMD also takes advantage of the underlying hardware. It makes use of both the data network and the control network in the CM-5. In particular, the control network is used in global communications operations such as reductions and broadcasts.

ACLMPL was developed with similar constraints as CMMD: message passing within a single multiprocessor machine (MPPs and SMPs) and sufficient primitives without trying to be all encompassing.

## 4. Implementation

ACLMPL is split into two groups: the synchronous communications primitives and the asynchronous primitives. On top of the synchronous primitives are layered the global communications primitives. Splitting synchronous and asynchronous primitives into two separate groups, with no overlap, makes sense. Layering asynchronous on top of synchronous does not make sense. Layering synchronous on top of asynchronous will work, but it introduces additional overhead (extra function calls, buffering, etc.); and as the timings will show, synchronous communication is faster than asynchronous communications. Additionally, both are faster than the other message passing systems.

The following sections will describe the implementation of ACLMPL on the T3D. Later sections will discuss the differences on the CM-5 and SGI.

### Synchronous Communications

The synchronous message passing API in ACLMPL was implemented first. Synchronous message passing has some potential performance advantages over asynchronous methods since there is no need for intermediate buffering. Data can be sent directly from the sender to the receiver with no need for additional data copying. This can result in much higher band-

Rough Draft

width and lower latency than is possible with an asynchronous protocol. The tradeoff is that computation cannot be overlapped with communications[2].

A simple protocol built on the CRI SHMEM library `shmem_put()` function is used as the lowest level communications primitive on the T3D [CRI]. Figure 1 shows the protocol used to send data between two processes on two separate Processing Elements (PEs). The receiving PE first writes a request block to the sending PE which contains the receive buffer address, its buffer length and a control flag. The request block totals 16 bytes. Each PE has an array of request blocks, indexed by receiving PE. This avoids the need for locks on the request blocks since each block has only one writer.

The sending PE blocks, via a spin-wait loop checking the control flag, until this request block arrives. Once the request block is received by the sender, the sender initiates a `shmem_put()` from the local send buffer address to the receivers buffer address which is taken from the receive request block. Finally, after the data is transferred, a finalization block is transmit-

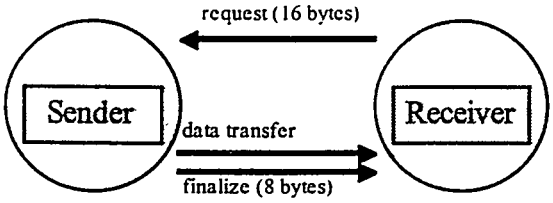


**Figure 1: Synchronous Protocol**

ted back to the receiver, indicating the size of the transfer, in bytes, and a flag value (**DONE**) indicating the transfer has completed. This finalization block consists of 8 bytes.

The receiver, after initiating the request, waits in a spin-wait loop for its flag to change to **DONE**. Once the flag changes to **DONE**, both sender and receiver return. The synchronous protocol requires one round trip between the sending and receiving PEs and a total of 24 bytes of overhead information. This results in very low end-to-end latency (4.5 microseconds for a one word message transmitted between direct neighbor PEs) and high bandwidth (greater than 100MB/sec for *one-to-all* and *all-to-one* communication patterns).

Based upon the synchronous protocol are three user callable functions: send, receive, and send_and_receive (send to one PE and receive from another PE, possibly the same).

[2] Except through the use of a thread package which allows multiple threads of execution on each PE.

## T3D Asynchronous Protocol

Efficient asynchronous message passing exposes a number of implementation challenges on the T3D. Unlike the synchronous case, buffer management issues come into play. Additionally, at least one extra data copy will be necessary between the application memory and a buffer within the message passing library, which is avoided in the synchronous case. Since world aligned `memcpy()` speeds on the T3D are only 170MB/sec[3] (approximately), it is important to minimize the number of data copies in order to achieve high bandwidth.

Our approach to the buffer management problem follows that used in the Illinois Fast Messaging library [FM]. As in FM, we use the fetch-and-increment registers on the T3D to allocate remote buffers from a fixed sized pool of buffers as shown in Figure 2. A sending PE reads the fetch-and-increment register on the receiving PE. The read operation returns the current value of the fetch-and-increment register, while atomically incrementing it as well. If the fetch-and-increment register is out of the bounds of the buffer pool, the sender must block until the receiver removes messages from the buffer pool and resets the fetch-and-increment register. If it is in bounds, the value read gives an index into the receiver's buffer pool, providing a buffer which the sender has exclusive access to. The sender transfers the message data to this buffer, via `shmem_put()`, and transfers a flag value **DONE**, indicating the transfer is complete.

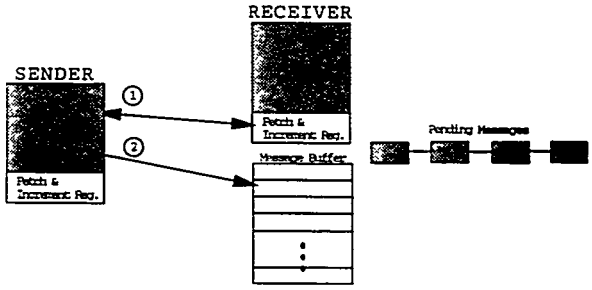


**Figure 2: Asynchronous Protocol**

The receiving PE first checks a linked list of sent-but-not-yet-received messages for a message that matches the receive request. If matching message is found, the data is memcpy'd to the callers buffer and the linked list node is freed. If a matching message is not found in the linked list, the buffer pool itself is scanned for a matching message. If a matching message is found, the data is memcpy'd to the caller's buffer and the buffer pool slot is marked as **RECEIVED**. In most

[3] In earlier releases of the CRI `memcpy`, bandwidth performance was 10-30X worse!

Rough Draft

cases, the linked list is empty and a matching message is found directly from the buffer pool, resulting in a one data copy, in addition to the `shmem_put`.

Each PE periodically checks whether its fetch-and-increment register has overflowed. This check is made each time a send or receive request is processed. The check can be accomplished by examining the last buffer in the buffer pool to see if it is marked as **DONE** or **RECEIVED**. If the fetch-and-increment register is out of bounds, all messages in the buffer pool are copied out into a linked list of sent-but-not-yet-received messages, and the fetch and increment register is reset to zero. This allows blocked senders to resume.

The user callable functions for the asynchronous protocol are asynchronous send, asynchronous receive, and blocking asynchronous receive. ** SHOULD THIS BE WORDED DIFFERENTLY?

## Global Operations

Broadcast and reduce global operations are implemented in ACLMPL using efficient tree based algorithms [Ho]. For simplicity, both broadcast and reduce use PE 0 as the root processor, though the algorithms can be generalized to handle any root PE.

A broadcast from PE 0 is sent in $\log(P)$ phases, where $P$ is the partition size. In the first phase, only PE 0 is active and the broadcast is sent from PE 0 to PE $(P \div 2)$. In the second phase, PE 0 and PE $(P \div 2)$ are active and each sends to PE $self + (P \div 4)$. In the $i$ th phase, PEs which have received the data forward the data onto the PE whose PE number differs only in the $i$ th bit. This is a well known algorithm whose complexity is $O(N \bullet \log(P))$, where $N$ is the size of the broadcast and P is the partition size [4]. The reduction operation uses the same tree structure used in the broadcast but in reverse, again yielding a $O(N \bullet \log(P))$ time bound. Initially all PEs are active. In the $i$ th phase of the algorithm, the PE's which have a $1$ in the $i$ th bit of their PE number send to the PE whose PE number is identical except for a $0$ in the $i$ th bit. The sending node becomes inactive, while the receiving node combines the received data with its own and proceeds to the next phase. At the end of the reduction, PE 0 holds the entire reduced array.

[4] Technically, this time bound and those that follow assume a hypercube interconnection network, though empirical evidence indicate that they match well to measured performance on the T3D 3D torus network as well.

Note that in each phase of the reduction, as we move up to the root of the tree, fewer processors are participating in the reduction. This suggests that a more efficient algorithm could be devised which utilizes all the processors during every phase. We first made this observation in a special case of the reduction algorithm: image compositing in a sort last volume renderer [PRS93]. In our *binary-swap* reduction algorithm we split the array being reduced in half at each phase of the algorithm and keep all PEs active throughout all phases.

In the $i$ th phase of the algorithm, two PE's whose PE numbers differ only in the $i$ th bit split their reduction array into two sub-arrays of equal size. One PE takes the lower sub-array while the other takes the upper sub-array. The two PE's exchange data, combine the received data with their own, and both proceeding to the next phase. At the end of the final phase, the entire array has been reduced, but it is distributed across all the PE's. A final gather stage brings the result together in PE 0. The binary swap reduction algorithm runs in time $O(N)$ when the array size $N$ is much larger than the partition size $P$. On the T3D we have found that $N >= 1024$ is sufficient for binary swap reduction to outperform the simple tree based algorithm.

As previously mentioned, the global operations are built upon the synchronous primitives. Since all PEs must participate in a global operation, asynchrony is not needed. Furthermore, the synchronous primitives are faster since they do not do any buffering of data.

The global operations consist of a broadcast and reduce primitive. The reduce primitive is extensible in that the user can write a reduction operator.

## ACLMPL for the CM-5

Since ACLMPL closely mimics CMMD, The CM-5 version of ACLMPL consists mainly of `#defines` instead of actual functions. This results in no overhead for using ACLMPL on the CM-5. The only real ACLMPL function is the reduction primitive. This is so that the user can write his or her own reduction operator, which is not supported by CMMD. Additionally, we have found the ACLMPL version of broadcast to be faster than the CMMD version for larger message sizes (greater than 1 KB ???).

## ACLMPL for the SGI

The Silicon Graphics version of ACLMPL is based upon IRIX specific interprocess communication (IPC)

Rough Draft

functions.[5] These functions allow for the creation and management of a shared memory pool which is used to facilitate the communication of messages between processors. The current implementation lacks several optimizations, such as using direct memory mapping, which can be used to increase performance. Future development will address this optimization and others.

In addition, ACLMPL has been implemented on top of PVM's `psend()` and `precv()` functions. This not only provides us with a more portable version of the library, but can also help in the early stages of application development and debugging without the use of an MPP.

## 5. Timings

Numerous benchmarks were performed on ACLMPL, MPI[6], PVM, and SHMEM using the T3D. Performance figures are include for SHMEM to give a reference for how the message passing systems compare to using shared memory for communications. Six different test cases were run on various partition sizes and for various message sizes. The six cases are: one PE communicating with all others (one-to-all), all PEs communicating with all others (all-to-all), all PEs communicating with one PE (all-to-one), global reduction, global broadcast, and latency.

The six cases were chosen for the following reasons. One-to-all is typical of initial data distribution, such as when one PE is responsible for reading a file and distributing parts of it to different PEs. Similarly, all-to-one is representative of gathering results back from all PEs for performing serial I/O. All-to-all is indicative of worse case, general communications. Global reduction and broadcast are included since they are very common global operations. The latency benchmark measures the overhead involved in sending very short messages (1 word) and measures the minimum overhead in sending short messages. Because many of the graphs exhibit similar curves, we have chosen just a representative few for this paper.

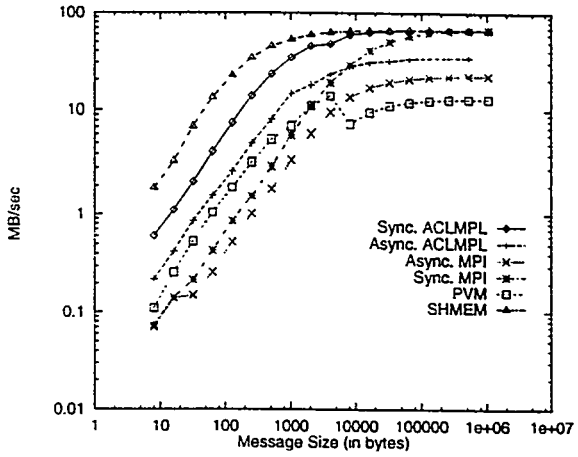


**Figure 3: all-to-all using 2 processors**

Figure 3 and Figure 4 show the performance curves for the all-to-all case on 2 and 128 processors. The Y axis shows throughput and the X axis shows message size in bytes. Several interesting features can be seen.

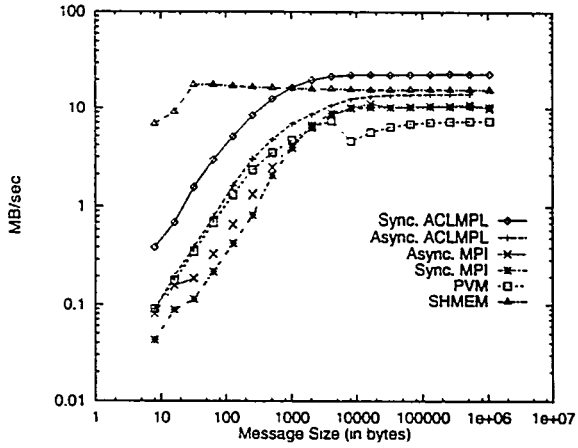


**Figure 4: all-to-all using 128 processors**

Throughput for all of the message passing systems increases greatly until the message size becomes sufficiently large (greater than 1K bytes) and then tapers off. ACLMPL is as fast as all of the other message passing systems for all cases; and for large partitions, it is the fastest including shared memory for certain message sizes. This seems curious at first since ACLMPL is built on top of SHMEM. The explanation is that the SHMEM version floods the T3D network and causes collisions, thus reducing performance. ACLMPL requires serialization (a PE can only receive from one sender at a time) which helps avoid saturating the network switches, thus resulting in greater performance. As the partition size increases, maximum throughput decreases from 67 MB/s to 23 MB/s. The kink in the PVM curve is due to a different, internal algorithm used by PVM for handling

[5] The IRIX routines have better performance than the standard AT&T System V Release 4 IPC routines. See the SGI Insight manual "Topics in IRIX Programming", for details.

[6] The T3D MPI implementation was from EPCC. The MPICH implementation could not properly execute the test programs.

Rough Draft

large messages[7]. Finally, asynchronous ACLMPL functions are also faster than the other message passing systems for large partitions.

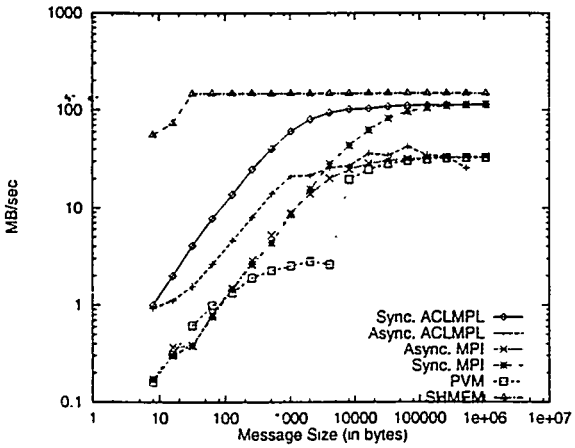


**Figure 5: all-to-one using 128 processors**

Figure 5 shows performance curves for all processors sending to one processor on a 128 processor partition. The synchronous version of ACLMPL is faster than the other message passing systems, as is the asynchronous version in all but a few cases. SHMEM is faster than ACLMPL in all cases since there is not the abundance of collisions on the network as there is with the all-to-all case. Maximum throughput is greater than 110 MB/s for ACLMPL.

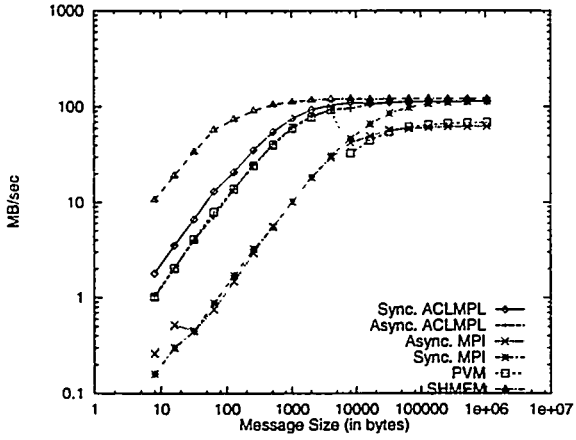


**Figure 6: one-to-all using 128 processors**

The one-to-all case, Figure 6, exhibits similar performance curves with the exception that PVM seems to do better than it did in the all-to-one case. Curiously, the spike in the PVM curve changes direction (HOW TO EXPLAIN??).

[7] See the Cray T3D documentation on the PVM_DATA_MAX environment variable.

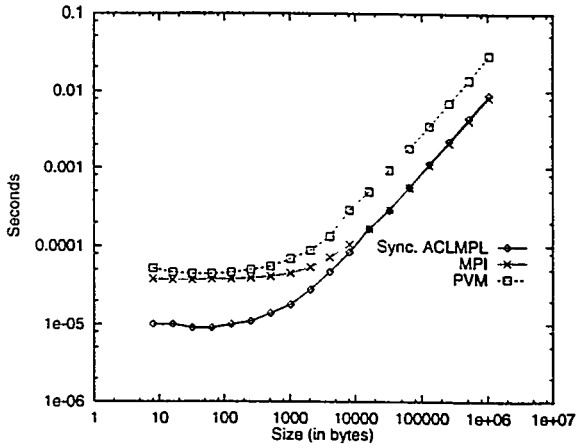


**Figure 7: broadcast using 2 processors**

Figure 7 and Figure 8 show broadcast times for 2 and 128 processors Both graphs exhibit similar curves with the exception of the PVM curve. As the number of processors increases, the upward spike in the PVM curve grows. It should also be noted, that as the number of processors increases, the time for all message passing systems increases regardless of message size.

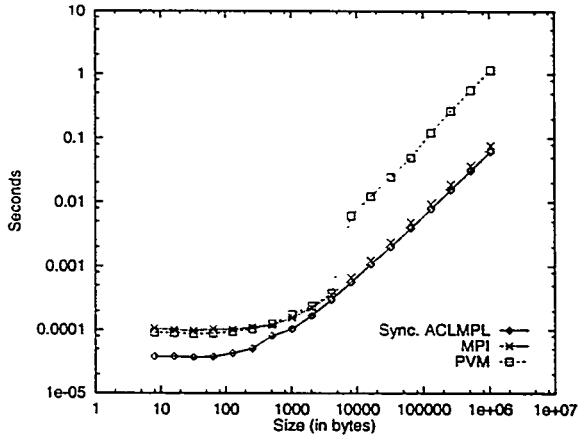


**Figure 8: broadcast using 128 processors**

Figure 9 shows times to perform a global reduce using 128 processors. MPI is significantly slower than ACLMPL; and PVM performs well for small messages but then degrades for larger messages.

Rough Draft

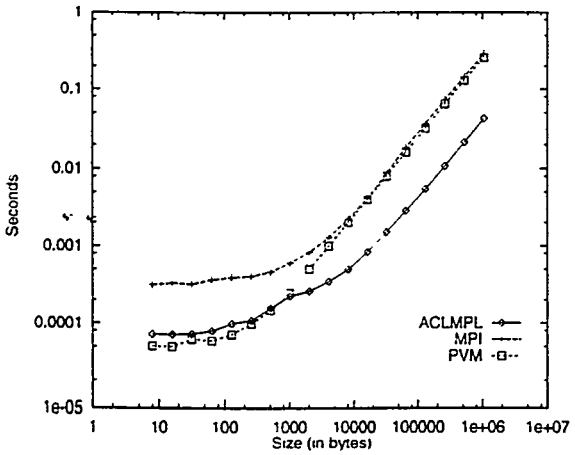


**Figure 9: reduce using 128 processors**

| Protocol | Time (μ seconds) |
|---|---|
| ACLMPL (sync) | 5 |
| ACLMPL (async) | 10 |
| PVM | 25 |
| MPI (sync) | 47 |
| MPI (async) | 40 |

**Table 1: Latency**

Table 1 shows the latency times for sending a one word message. Both the MPI synchronous and asynchronous versions incur significant overhead in sending a short message (greater than 8 times that of ACLMPL synchronous messages).

Table 2 presents performance numbers for 1024 byte messages on a 32 processor partition. The numbers for all-to-all, all-to-one, and one-to-all are in megabytes per second; and the numbers for broadcast and reduce are in seconds. For the first three cases, the synchronous functions in ACLMPL are approximately between 4 and 7 times faster than the other message passing systems, and broadcast and reduce are roughly 10% to 80% faster

## 6. Results

While ACLMPL grew out of the efforts of the visualization group at the Advanced Computing Lab, it is a general purpose communications library. One example of its use is a molecular dynamics application for massively parallel computers. Their application is used to simulate molecules containing several hundreds of millions of atoms. In 1993 they won a Gordon Bell prize for performance (they were able to sustain >53 Gflops on the CM-5). It should be noted that at that time the application was based on CMMD. It is currently being ported to ACLMPL.

ACLMPL has been used in two newly developed visualization applications. One, a sphere renderer, is used by the molecular dynamics projects for displaying their data. The renderer can be used as either a standalone program or as a MIMD callable library. As a standalone program, the renderer can be used either interactively with an X11 graphical user interface (GUI) or in a batch mode. Images are either displayed in an X11 window, a HIPPI frame buffer, or written to disk. Rendering rates on the T3D are approximately 660K spheres per second. For comparison, a SGI Onyx graphics workstation can sustain roughly 19K spheres per second.

The second visualization application is a renderer for volumetric data based upon Binary-Swap Compositing [HAN]. The renderer distributes a 3D data set to the PEs. Each PE is responsible for rendering its own subvolume. After each PE is done, then the subimages are composited together using a technique called Binary-Swap. The user can interact with the renderer either through an X11 interface or through AVS. The renderer can generate approximately 4 frames per second using 128 PEs to render a $128^3$ data set into a 256 x 256 image that is displayed on a HIPPI frame buffer.

## 7. Conclusions

ACLMPL was developed with two goals in mind: to provide high throughput, low latency communications for message passing applications, and to provide portability. As previously shown, ACLMPL is approximately XX times faster than either MPI or PVM on the Cray T3D. This is significant to MPP applications since slow communications can reduce performance and scalability.

Since ACLMPL is based very closely on Thinking Machine Corp.'s CMMD, we can preserve our software investment. We have found ACLMPL to be quite portable and still retain efficiency.

| | ACLMPL (sync) | ACLMPL (async) | MPI (sync) | MPI (async) | PVM |
|---|---|---|---|---|---|
| alltoall | 19.00 | 7.93 | 4.71 | 4.58 | 4.43 |
| alltoone | 61.90 | 36.43 | 10.70 | 10.72 | 8.94 |
| onetoall | 74.00 | 60.61 | 10.70 | 9.96 | 57.02 |
| bcast | 0.000076 | | 0.000135 | | 0.000138 |
| reduce | 0.000162 | | 0.000452 | | 0.000180 |

**Table 2: Performance for 1KB messages on 32 processors**

While we don't expect ACLMPL to become the "new message passing standard", we would hope that it can be seen as a challenge to other message passing systems implementators. ACLMPL should be viewed as proof that it is possible to develop a portable, useable, high performance message passing system for MPPs.

Finally, three major points should be noted. First, synchronous message passing is inherently simpler than asynchronous message passing. This is because buffer management and additional data movement can be avoided. These optimizations should be used. Second, efficient global communications algorithms exist and should be used; otherwise, scalability to large partition sizes is impaired. Last, on the T3D efficient buffer management can be performed using the fetch-and-increment facilities.

---


[PVM] PVM User's Guide and Reference

[MPI] MPI Manual

[CMMD] A Reference to CMMD.

[CRI] Cray Research Inc., SHMEM Manual Page.

[FM] Fast Messages.

[Ho] \cite{Ho,Geijn91}

[PRS93] "Parallel Volume Rendering Using Binary-Swap Compositing", Kwan-Lui Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh, IEEE Computer Graphics and Applications, Vol. 14, No. 4, July 1994.

[HAN] "Binary-Swap Volumetric Rendering on the T3D", Chuck Hansen, Michael Krogh, James Painter, Guillaume Colin de Verdière, and Roy Troutman, Cray Users Group Conference, March 1995, Denver, CO.