Fast Message Passing via the ALLCACHE Memory on KSR Computers*

Kaushik Ghosh (kaushik@cc.gatech.edu)

GIT–CC–94-37

September 24, 1993

Abstract

A large body of applications have been built which use a message-passing style of interprocess communication. Thus, it is important to be able to support efficient message-passing even on shared-memory computers. Unfortunately, ‘direct’ porting of message-passing packages to shared-memory computers invariably produces unacceptably poor performance. In this paper, we discuss schemes for efficiently implementing the primitives of two commonly-used message-passing packages – PVM and TCGMSG – through the ALLCACHE memory of KSR computers. First, we describe a generic interface for message-passing and buffering, and map the specific calls of these two packages on to this generic interface. We then derive analytical results about the achievable bandwidth for message-passing via the shared ALLCACHE memory on KSR machines. Further, we describe a simple but widely-used benchmark called ping-pong, and report the performance of this benchmark on our implementation of each of the two message-passing packages. Finally, we suggest some new features to the system software on KSR machines which might support such packages more efficiently, and point out some drawbacks in the interfaces of the packages which hinder their efficient implementation on multiprocessors.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

*This work was performed while Kaushik Ghosh was on an internship at Kendall Square Research (July - September '93), and was supervised by Dr. Stephen R. Breit. Dr. Breit also provided useful comments on the paper. Dennis Marsa implemented an earlier version of TCGMSG on KSR.
1 Introduction

In recent years, distributed computing has been widely used for high throughput at low cost. Message-passing is a low level of communication software on a network of computers. While powerful high-level abstractions like Distributed Shared Memory can be built up over a layer of message passing software, raw, low-level message-passing produces the best performance.

Several message-passing libraries are available today: CPS [2], Linda [1], TCGMSG [4] and PVM [5], to name a few. Most of these libraries are complete parallel-programming environments, rather than mere low-level communication software. There is now an increasing number of application programs that have been parallelized using these “standard” message-passing interfaces. Therefore, it is highly desirable to support common message-passing interfaces on shared-memory multiprocessors like the KSR.

While it is possible to do a naive and direct ‘port’ of such communication software for intra-machine communication on a shared-memory multiprocessor, the performance of such ports is almost always unacceptable. This is because the libraries use socket-based communication: the time required to perform socket operations is excessive, and sockets are not really required for intra-machine communication when shared-memory is available. Thus, the goal of this work was to develop efficient implementations of message passing primitives by exploiting the shared memory on KSR computers. Further, we had another important aim: that of demanding as little modification as possible to the application programs using the message-passing packages.

With these objectives, we designed and implemented buffering schemes that made optimal use of the memory system on KSR computers. We heavily optimized intra-machine communication (i.e., communication between processes local to a KSR machine), but left inter-machine communication as it is done in the ‘standard’ way in the respective message-passing libraries. We achieved 10 MBytes/sec average message transfer rate on PVM, and 10.6 MBytes/sec on TCGMSG, for communicating processes on a single KSR1 computer\(^1\).

The rest of this paper is organized as follows: in section 2 we describe a generic interface for message passing libraries, pointing out how the relevant calls in PVM and TCGMSG map onto the interface. We also state why most common message-passing protocols require buffering. Further, we give a brief account of the architecture of the KSR1, describe our buffering scheme, and derive analytical results about the achievable rates for message transfer via shared memory on the KSR1. Later, we discuss some specifics of TCGMSG in section 3.1, and PVM in section 3.2 respectively. We point out features that might prove useful enhancements to the system software on KSR computers for supporting message passing, and indeed any kind of client-server computing. We also mention drawbacks in the interfaces of the message-passing packages that hinder their efficient implementation on multiprocessors. We describe a simple but widely-used benchmark called ping-pong and report the performance of TCGMSG and PVM on this benchmark in section 4. In section 5 we conclude with an overall assessment, and anticipated future work with respect to this project.

\(^1\)We implemented the interfaces specified in version 4.03 of TCGMSG and 3.1 of PVM
2 General Approach

In this section, we describe the approach we took to tune two libraries – PVM and TCGMSG – on KSR parallel processors. We introduce a generic interface for message passing, which we use to explain the buffering scheme, and argue about maximum achievable message passing rates on KSR1 computers.

2.1 A Generic Message-Passing Interface

We abstract the message transfer operations in the two libraries as send and receive operations. The send operation dispatches specific data for a particular recipient; a variation of send dispatches data for all, or a group of processes in the system (the broadcast and multicast mechanisms). The receive operation gets data from a specified process, which can be a wild-card. Receives can be blocking (wait until data becomes available) or non-blocking.

Thus, in TCGMSG, a send will consist of a SND_() call and a receive will consist of a RCV_() call. In PVM, a send will consist of one or more pvm_pk*() followed by a pvm_send(), while a receive will consist of a pvm_recv() followed by one or more pvm_ipk*().

It is possible to improve on the performance of such communication software by means other than modifying the `send’ and ‘receive’ routines. Thus, e.g., one could use some kind of kernel-level threads instead of ‘processes’ in the application. This would allow the application to have shared data between ‘processes’ by default – precluding the need for any kind of extra buffering in the communication software. A ‘send’ would directly write into appropriate memory locations, while a ‘receive’ would read from them. However, our goal of maintaining portability between platforms did not allow us to take this approach. For, Unix processes\textsuperscript{2} have distinct address spaces associated with them. Thus, processes A and B in an application can both refer to a variable c in their private address space, and this would produce two copies of c. However, having threads as the control entities for A and B would produce only one copy of c. Almost all applications using message-passing assume the ‘distinct-address-space’ model. Using a kernel-threads (thus shared global data) approach would require rewriting such applications. Our goal being to run existing applications with minimum modification, we could not take this approach.

2.2 Architecture of the KSR1

The KSR supercomputer is a NUMA (non-uniform memory access) shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes or 34 rings. Each node consists of a 64-bit processor, 32 MBytes of main memory used as a local cache (with each cache line being 128 bytes long), a higher performance 0.5 Mbytes sub-cache, and a ring interface. CPU clock speed is 20 MHz, with peak performance of 40 MIPS per node (due to two functional units that can operate in parallel). Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheap\textsuperscript{3}. A subcache miss produces a

\textsuperscript{2}Most of the communication software libraries mentioned run on Unix.
\textsuperscript{3}Most of the communication software libraries mentioned run on Unix.
delay of 23 cycles, a cache miss produces a delay of 150 cycles, and a ring miss gives rise to a delay of 600 cycles (one level). At the lowest level, the parallel programming model offered by the KSR’s OSF Unix operating system is one of kernel-level threads which offer constructs for thread fork, thread synchronization, shared memory between threads, etc.

2.3 Buffering Scheme

In this section, we discuss a few salient points about the KSR-specific buffering scheme in TCGMSG and PVM, and point out the basic differences between the approaches in these packages.

New low level send and receive routines were written for the KSR with the ALLCACHE memory architecture in mind. The two libraries implement two kinds of sends and receives; local and remote versions. The local versions are used to pass messages between processes running on the same machine via shared memory, while the remote versions are used to pass messages between processes running on different machines via TCP sockets. The remote versions were not changed significantly in our implementation. However, the local versions were completely rewritten for the KSR.

While TCGMSG had a scheme in place to use shared memory, and semaphores to implement message passing via shared memory, PVM used socket-based communication.

The performance of this original scheme is poor in part due to the high overhead associated with semaphore and socket operations. Each semaphore or socket operation results in a system call. Faster synchronization between sender and receiver can be achieved by using the KSR atomic state operations gsp.vt and rep which is done in the new send and receive routines. Further, the new send and receive primitives written for the KSR implement a more sophisticated buffering scheme. The idea behind these new routines is that each process maintains a pool of buffer space that is likely to be held exclusively by its cache.

The basic approach in the new routines is to share buffers between processes, writing into these buffers on a ‘send’ and reading from them on a ‘receive’. The approach we follow is much the same for both of the libraries: we use shared headers, as well as message-body buffers. However, unlike TCGMSG, where the maximum number of processes that will be used is known at startup, PVM allows dynamic creation of new processes (pvm_spawn()). We allocate 227 bytes of shared memory5, equally among the processes. Each process gets just over 4MBytes; thus, roughly 31 PVM processes can be supported at a time6.

A sending process will use buffer space from its own pool to send a message which will minimize the send time since it does not have to invalidate sub-pages in other caches as it copies the message into the buffer space. The receiving process will move the sub-pages containing the message from the sender’s cache to the receiver’s cache during the process of copying the message out of the buffer space. Once the message is received, the buffer space used to send the message is held exclusively by the receiving process and is added to its pool of buffer space for reuse later. To make sure that sub-pages actually move from the sender’s cache to the receiver’s cache, the assembly code that performs the copying in the receive routine must be modified. The assembly code reads the message data out of the buffer space using ld8 instructions which will leave a copy of the referenced

---

3 All the performance measurements in this paper were carried out on a single-ring, 32-processor KSR1 machine.
4 KSR computers are equipped with Pthreads; a kernel-level threads library.
5 Theoretically, we can use as much as 215 bytes of shared memory. However, at the time of this implementation, if we used 215 or more, shmget() failed, and if we used between 227 and 229, shmget() returned properly, but the runtime system produced segmentation violations at odd places. Thus, the current implementation uses 227 bytes of shared memory.
6 The current implementation uses 128 each of shared and private headers, 4 fixed headers, and 128 each of shared and private slots and 4 shared slots per PVM process. These numbers can be reduced to accommodate a larger number of PVM...
data in the sender’s cache. Since we want the sender to relinquish its copy so that the receiver has an exclusively copy, the ld8 instructions must be changed to ld8.ex instructions.

The ld8.ex instructions are not generated by the compiler, since the receive primitive only reads the locations. They are manually inserted in the appropriate assembler file.

However, if prefetches can be used, blocking prefetches in exclusive mode during a receive offer much the same performance benefits as modifying the ld8 instructions would, without using any special assembler files. Further, using blocking prefetches in exclusive mode gives the best performance in both of the libraries.

Each process maintains a pool of message headers and message slots which together make up the buffer space. Message headers contain the vital information about a message including its type, size, destination, and for very small messages, the data itself. Message slots are used to hold message data for messages larger than those which can be transferred in a message header. When sending a message, the sending process will choose a message header and as many message slots as are needed to hold the message to send from its list of free headers and slots. The sending process searches for free headers and slots in its private pool first. If its private pool is empty, it searches the private pools of other processes to find free buffers. The header and slots are filled in, linked together into a linked list and then added to a list of received messages for the receiving process. The receiving process takes the message off of its list of received messages, copies the data out of the message and adds the now free header and slots to its pool of free headers and slots for use later when it performs a send operation.

There is a slight difference in the way message transfers work in PVM and TCGMSG. In PVM, all data to be sent has to be packed into the active buffers before the send can start, since the pvm_pk*s must complete before the pvm_send. However, in TCGMSG, the send can start (and the receiving process can begin to receive the message) before all the data has been packed in: essentially allowing a sort of ‘pipelining’ of the message transfer. For this reason, the ping-pong benchmark (see section 4) performs somewhat better when run under TCGMSG than under PVM.

2.4 Estimates of Message-Transfer Rates

Here, we present some simple results about the expected message-passing bandwidth on the KSR1 assuming large messages (set up times negligible). We perform this analysis first for the non-prefetch buffering scheme (with ld8 changed to ld8.ex and then for the buffering scheme that uses prefetches.

Sending a byte involves

- loading the byte into the processor (this load is usually from subcache, or local cache), and
- storing the byte into shared memory (the shared memory in a send is always a part of the local cache).

Receiving a byte involves loading the byte from remote cache (the sender’s cache). This operation loads not only one byte but the whole subcache line (128 bytes) in which the byte in
question resides. Thus, there is one cache-miss penalty for every 128 bytes in receiving large messages. Thereafter, the received byte is stored in the private memory (usually local sub-cache) of the recipient process.

The delay figures for the stores in both cases above is 0. Hence, we will consider only the loads in the following discussion.

Thus, sending 128 bytes requires 256 cycles (2 cycles for the delay of loading each byte from local cache). Receiving 128 bytes in the no-prefetch case requires 150 cycles to get the first byte from remote cache, and thereafter 2 cycles for each of the remaining 127 bytes: thus a total of $150 + 2 \times 127 = 404$ cycles. If prefetches were used, the receive operation would require $2 \times 128 = 256$ cycles, since the relevant line would already be prefetched into local cache. The clock cycle is 50 nanoseconds on the KSR1.

Therefore, the projected average transfer rate (assuming that all processors could be kept busy transferring data all the time) is about

$$
2 \times \frac{128}{(404 + 256) \times 50 \times 10^{-9}}
$$

when prefetches are not used. This reduces to 7.76 MBytes/sec. If prefetches are used in receives, this figure would go up to about

$$
2 \times \frac{128}{(256 + 256) \times 50 \times 10^{-9}}
$$

i.e., about 10 MBytes/sec.

3 Two Message-Passing Libraries: TCGMSG and PVM

The approach described above is general enough to allow most message-passing libraries to be mapped onto the shared memory on KSR computers. In this section, we discuss the salient features in the implementation of the buffering scheme for two popular message-passing libraries: TCGMSG and PVM.

3.1 TCGMSG on KSR

The placement of processes on the available machines is specified in a configuration file in TCGMSG, the topology being known, and there being no call to dynamically spawn new processes after the `BEGIN()` call completes, the total number of processes that are going to be set up on the KSR machine is known at initialization. A chunk of shared memory is allocated and initialized for each of these processes. This memory is divided into a number of buffers: for message headers, and slots for the message bodies.

The initial 'master' process uses `rsh` to initiate processes on each specified host. These 'initial processes', in turn, use `fork` to set up the appropriate number of children on each host. If the host is a KSR machine, the initial process sets up some shared memory using `mmap()`. Since TCGMSG
produces the children using \texttt{fork}, thus without overlaying memory, children see the shared memory region set up by the parent. The children are bound to the processors in the cluster on the KSR machine (this is achieved using the \texttt{psm\_bind()} call).

A \texttt{SEND()} operation checks to see if the recipient process is local. If not, the ‘standard’ TCGMSG approach is used to send the message. If it is local, the message contents are written into one or more shared buffers, which are then linked into the appropriate recipient process’s list of unreceived messages. However, if the message is short enough, the ‘data’ is put in the unused part of the header (which is about 72 bytes in TCGMSG 4.03 on the KSR – headers and slots being aligned on 128-byte subpage boundaries for optimal performance).

A \texttt{RECV()} operation checks to see if the process to be received from is local (or, if the ‘source’ is a wild-card, it checks to see whether there is a pending message from any local process). If there is a pending message, and if it is of the proper type (as specified in the \texttt{RECV()} operation), the header is removed from the linked list of buffers. The message size is determined from the header, and the body of the message is then received, slot by slot. If the message is arriving from a non-local process, ‘standard’ TCGMSG routines are used.

3.2 PVM on KSR

PVM directs that a \textit{pvm\_daemon} be set up on a per-user basis: multiple login sessions of a user will use the same pvm\_daemon. We allocate a chunk of shared memory and perform local communication using the shared memory instead of sockets. However, the local PVM processes will be “registered” in the daemon tables, so that messages received from PVM processes running on other machines can be routed to these local processes.

Shared memory is set up when the PVM daemon is started. We use system V \texttt{shmat} to allocate the shared memory. This is because the ID of the shared memory region has to be made known to processes that will be started up later, and there does not seem to be a clean way to do this using \texttt{mmap()}\footnote{While we could have used \texttt{mmap()} to map in a named file, we did not want to leave such a large file (several hundred MBytes) lying around in the filesystem; so we went with the \texttt{shmat} approach.}. The ID of this shared region is written in a file in /tmp. Every PVM process reads this file at startup, finds out the ID of the shared region, and \texttt{shmat()} that region. After that, communication takes place in much the same way as in TCGMSG on the KSR. On exiting, the PVM daemon, removes this area of shared memory\footnote{The KSR implementation creates a file called /tmp/pvm\_shmid.<userid>. This file contains the shmid of the shared memory region, which is used by different PVM processes to attach the shared region to their address space.}.

We need to perform an \textit{explicit} attach of the shared region in the application program: this is because \texttt{pvm\_spawn()} may be used to create a process. PVM allows separately compiled programs to spawn each other; hence, it uses \texttt{execv()} which overlays shared space. The spawned program need not perform an explicit \texttt{pvm\_mytid()}. PVM dictates that this routine \textit{must} be called before any other pvm routine by a process not created using \texttt{pvm\_spawn}. Therefore, ALL processes running PVM on the KSR must call an initialization routine \texttt{KSR\_init()}, which attaches the shared memory region to the process’s address space, and binds the process to a KSR processor.

Unlike TCGMSG, PVM allows dynamic spawning of new PVM processes. Therefore, we use a load table to bind processes to processors, performing simple load balancing. If a spawn can take place on the local machine\footnote{The \texttt{flag} parameter in \texttt{pvm\_spawn()} is zero, or \texttt{flag} is two and \texttt{work} is ‘KSR1’, or \texttt{flag} is one and \texttt{work} is the local host (as determined by \texttt{gethostname()}).}, we bind the spawned process to the node in the processor set with the lowest load, and increment the load on that processor by one. On the other hand, when a PVM
process exits, we detach the shared memory region that was attached to it, and decrement the load on the corresponding processor by one. All of this initialization is performed in KSR\textunderscore init().

This brings us to a problem with processor sets on the KSR. Processes created by a \texttt{pvm\_spawn()} call are created by the daemon. Typically, the daemon would be started up by a command \texttt{allocate\_cells 1 pvm} from the Unix prompt on a KSR machine. This produces a dedicated set of 1 processor for that user, and runs the daemon on that set. An application would later be brought up with a command \texttt{allocate\_cells n application} to run on \texttt{n} processors. However, if the application created processes through \texttt{pvm\_spawn()}, the spawned processes can run only on the processor set of their parent: viz., the 1-processor processor set of the daemon. We have to find a way to pass send rights to the processor set port. For now, both the daemon and the application run on the default processor set\textsuperscript{10}.

We are mostly concerned with the send and receive routines in PVM. PVM takes the approach of packing multiple entities into an active message buffer before sending it out to the recipient process. The recipient, likewise, receives the active buffer, and unpacks individual entities from it. The relevant PVM routines are \texttt{pvm\_pk*()}, \texttt{pvm\_upk*()}, \texttt{pvm\_send()}, \texttt{pvm\_recv()} and \texttt{pvm\_mcast()}. The * represents a family of routines for different data types \{byte, short, integer, float, long, double, complex, string \ldots\}.

At the first \texttt{pvm\_pk*()} following a \texttt{pvm\_send()}, we set up a new message header, getting the header from the process's list of private, shared, or fixed headers (exactly as in TCGMSG on the KSR). If the current data item being packed is small enough, we put it into the space in the header (similar to the approach in TCGMSG). If it is too large to fit in the header, we mark the amount of data contained in the header as 0, and search for a slot from the list of private, shared and fixed slots of the process (exactly as in TCGMSG). On finding an available slot, we pack the rest of the data - continuing to copy into the current slot and obtaining more slots if there is too much data to fit into one slot. If a part of a slot is unused, the next \texttt{pvm\_pk*()} (before the next \texttt{pvm\_send()}) will start to pack where the previous one had ended, instead of obtaining a new slot. A field in the header denoting the amount of data in bytes that has been cumulatively packed in the current buffer, is increased. We do not follow XDR or other encodings that the user might have specified in initializing PVM, since they are not required if heterogeneous machines are not used. Heterogeneous environments are discussed later.

On a \texttt{pvm\_send()}, the header (with the slots packed by previous 'unsent' \texttt{pvm\_pk*}()s) is placed into the list of received headers of the recipient as in TCGMSG. Further, we increase the count of 'pending receive' in the header by 1 (these are used in \texttt{pvm\_recv()}). If there have been no \texttt{pvm\_pk*}()s since the last \texttt{pvm\_send()}, an error status is returned.

On a \texttt{pvm\_mcast()}, the header from previous unsent \texttt{pvm\_pk*}()s is put in the list of received headers of \textit{each} of the recipients, and the 'pending receive' count in the header is incremented by the number of recipients of the multicast.

A \texttt{pvm\_recv()} removes the first available header and slots in the recipient's received buffers which had originated from the named sender. If the named sender is a wildcard, it searches the local buffers for received messages. If an unreceived message has arrived at this PVM process, the message is removed (i.e., 'received'). If all the local buffers are empty, it looks for and removes

\textsuperscript{10}As a proposed solution, \texttt{allocate\_cells}, or its variants, could return permissions to the Mach port corresponding to the allocated processor set. The Mach call \texttt{thread\_assign()} could then be used to bind a thread to the processor set. This would also be useful in a lot of client-server type of applications, where the load on the client processor sets is low when the load on the server processor sets might be high.
messages from remote PVM processes.

Before the actual ‘receive’ is executed, \texttt{pvm\_recv} on the KSR does some buffer management. If a message had been received prior to the current receive, the old received buffer is no longer required by this PVM process, and can potentially be returned to the pool of free buffers. A flag (which is set in the \texttt{pvm\_upk} code, as described below) is checked: if this flag is \texttt{FALSE}, it means that the upk operations on the old message did not remove all the bytes that were packed in it. In this case, the ‘pending receive’ field is decremented in the old header. If the decrement operation causes ‘pending receive’ to become zero, the header and slots in the old received message are returned to the free pool.

\texttt{pvm\_upk*}’s remove items from slots: the items that were packed by corresponding \texttt{pvm\_pk*()} calls in the sender. In addition, when the cumulative number of bytes extracted from a received buffer becomes equal to the length of that buffer (as determined from the header), the ‘pending received’ field is decremented. If the ‘pending received’ field reduces to zero as a result, the buffer (header and all the slots) is freed; otherwise, the flag mentioned in \texttt{pvm\_recv()} above is set to \texttt{TRUE}.

The motivation behind the ‘pending receive’ field is that a user program may, perhaps due to bugs, not unpack all the data from a received buffer before performing the next \texttt{pvm\_recv()}. Since a buffer might have been received from a multicast rather than a send, it might not be possible to free the ‘old received buffers’ since they may not have been fully unpacked at all local PVM processes.

The approach outlined thus far works only for local sends and receives. In PVM, there is no way to know which process is the destination for data being \texttt{pvm\_pk*}-ed. The destination is known only at the next \texttt{pvm\_send}.

We follow the optimistic approach: we assume that the data is being packed for a local process, and therefore directly copy the data into the headers and slots, without going through the encoding (XDR, etc.) that has been prescribed. If later, during the actual \texttt{pvm\_send()}, we discover that the recipient is non-local, we copy back the data items into the PVM buffers (as opposed to the KSR-specific buffers), this time following the encoding scheme, and send out the message following PVM’s standard approach.

\textbf{Note:} if the user knows that the next send is going to be to a remote process, then \texttt{internal\_pvm\_primitive-name} may be used; thus, \texttt{internal\_pvm\_pkbyte} instead of \texttt{pvm\_pkbyte}, \texttt{internal\_pvm\_send} instead of \texttt{pvm\_send} etc. While this makes remote send/recv faster (since packing two times is not necessary any more), these ‘internal’ routines were created especially for the KSR version of PVM; they are not ‘officially’ supported.) These routines can be either called directly, or the user may call the KSR-specific routines \texttt{KSR\_remote\_only\_start()} and \texttt{KSR\_remote\_only\_end()}. All pack, send, multicast, receive and unpack operations ‘between’ these calls will operate on PVM’s standard active buffers and use PVM’s standard socket operations for communication.

An open issue is the following: even if there were support for cross-processor-set binding of threads, we would need a way to specify the number of processors that should be used by \texttt{pvm}\textsuperscript{\textsuperscript{11}}. Thus, while there is a way to run the \texttt{pvm} application on a ‘dedicated’ processor set – the way in

\textsuperscript{11}There is no support for doing this in \texttt{pvm}, perhaps because \texttt{pvm} was basically meant to be a package connecting workstations on a network.
<table>
<thead>
<tr>
<th>Package</th>
<th>Time for 0 bytes data transfer (μsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVM</td>
<td>110</td>
</tr>
<tr>
<td>TCGMSG</td>
<td>94</td>
</tr>
</tbody>
</table>

Table 1: Measures of latencies of PVM and TCGMSG (microseconds)

which we took the ‘ping-pong’ measurements discussed in section 4, it would be desirable to run in the default processor set until one is able to specify the number of processors that should be used on a remote machine; this would provide uniform behavior regardless of whether PVM is started from a KSR computer or from some other type of computer. Recall that it is possible to specify the number of processors to use when the PVM application is started from a KSR machine.

Performance would also be improved if there were a block-copy instruction available at the assembler-level on the KSR1. At the time of implementation, the compiler used eight-byte load/store as the unit of message-copying because the 64-byte version did not work under certain circumstances.

4 Performance Results from the Ping-Pong Benchmark

We measured the performance of the libraries with a ‘ping-pong’ benchmark. We first mention the salient points about the application, which has been described in detail elsewhere [6, 7].

The ping-pong benchmark repeatedly moves data between two processes. In each cycle of communication, the sender process sends to the recipient process. The roles of the processes are interchanged in alternate cycles.

Here, we report the performance of the ping-pong application running over PVM version 3.1 and TCGMSG version 4.03 on a KSR1. While we implemented separate versions of both of these libraries with and without prefetches (using `ld8.exe` in place of `ld8`), the measurements mentioned below were taken from the versions that use blocking prefetches of headers and slots in exclusive mode in the receives and unpacks. The latency and the bandwidth were as shown in tables 1 and 2, and graphically in figure 1.

Note that these are in response to message passing within a 32-node KSR1. For communication between machines (across the ethernet), the bandwidths are substantially lower, and the latency higher.

5 Conclusions and Anticipated Future Work

We investigated the effectiveness of the shared memory on KSR computers in carrying out a message-passing style of interprocess communication. We designed protocols to effectively map message passing, taking into account the ALLCACHE memory system on the KSR computers. We
<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>PVM bandwidth</th>
<th>TCGMSG bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>16K</td>
<td>6.52 Mbytes/sec</td>
<td>9.49 Mbytes/sec</td>
</tr>
<tr>
<td>32K</td>
<td>7.99 Mbytes/sec</td>
<td>9.81 Mbytes/sec</td>
</tr>
<tr>
<td>64K</td>
<td>8.96 Mbytes/sec</td>
<td>10.21 Mbytes/sec</td>
</tr>
<tr>
<td>128K</td>
<td>9.22 Mbytes/sec</td>
<td>10.43 Mbytes/sec</td>
</tr>
<tr>
<td>256K</td>
<td>9.80 Mbytes/sec</td>
<td>10.52 Mbytes/sec</td>
</tr>
<tr>
<td>512K</td>
<td>9.88 Mbytes/sec</td>
<td>10.6 Mbytes/sec</td>
</tr>
<tr>
<td>1M</td>
<td>9.93 Mbytes/sec</td>
<td>10.6 Mbytes/sec</td>
</tr>
</tbody>
</table>

Table 2: Bandwidths (Mbytes/sec) for different message sizes on PVM and TCGMSG.

showed the feasibility of our ideas by modifying the send and receive primitives in two popular message passing libraries to work via the shared memory on KSR computers. The implementation of our protocols on these libraries running on a 32-node KSR1 was described in detail, and the performance of a well-known benchmark reported. The message transfer rates on each of the libraries, after two different types of optimization – which were also described in detail – are extremely promising. Further, applications need minimal amount of modification to take advantage of this fast message passing: TCGMSG applications need no modification at all, while a single function, \texttt{KSR\_init()} has to be called initially for PVM applications.

Much of our future work investigates similar approaches to tuning other message-passing libraries (e.g., P4) for very fast intra-machine communication between processes on a KSR computer. We pointed out several features of PVM which either were not designed for use with multiprocessors, or need more support from the system software than is currently available on KSR computers. We are also investigating these issues.

References


Figure 1: Bandwidths for various message sizes in PVM and TCGMSG on a KSR1 (MBytes/sec)
