The Design and Implementation of an Asynchronous Communication Mechanism for the MPI Communication Model

Motohiko Matsuda, Tomohiro Kudoh, Hiroshi Tazuka
Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology

Yutaka Ishikawa
University of Tokyo, and
Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology

Abstract

Many implementations of an MPI communication library are realized on top of the socket interface which is based on connection-oriented stream communication. This paper addresses a mismatch between the MPI communication model and the socket interface. In order to overcome a mismatch and implement an efficient MPI library for large-scale commodity-based clusters, a new communication mechanism, called O2G, is designed and implemented. O2G integrates receive queue management of MPI into a TCP/IP protocol handler, without modifying the protocol stacks. Received data is extracted from the TCP receive buffer and copied into the user space within the TCP/IP protocol handler invoked by interrupts. It totally avoids polling of sockets and reduces system call overhead, which becomes dominant in large-scale clusters. In addition, its immediate and asynchronous receive operation avoids message flow disruption due to a shortage of capacity in the receive buffer, and keeps the bandwidth high. An evaluation using the NAS Parallel Benchmarks shows that O2G made an MPI implementation up to 30 percent faster than the original one. An evaluation on bandwidth also shows that O2G made an MPI implementation independent of the number of connections, while an implementation with sockets was greatly affected by the number of connections.

1 Introduction

Commodity clusters using the combination of Linux, Ethernet, and TCP/IP are now in wide-spread use, and recently, large-scale clusters with over a thousand nodes are not uncommon. The Ethernet with TCP/IP is a viable communication mechanism with its improvement to over a giga bit-per-second performance. In addition, the Grid computing further promotes large-scale computing using TCP/IP. MPI communication libraries [10] for parallel computing shall use TCP/IP as the transport layer in such an environment [6].

However, there is a mismatch between the MPI implementation model and the communication API of the *socket* model. The socket model abstracts communication endpoints and is almost the only interface used for communication in ordinary OSes such as Linux, Unix, and Windows. It offers messaging through a connection-oriented stream like TCP/IP. Originally, sockets were designed to handle a small number of connections per process. Thus, it is a well-known problem [2] that performance degrades when many sockets are used. Large-scale clusters need many sockets in proportion to the number of nodes, and thus, suffer from this performance degradation. Typically, receiving messages from sockets is performed by repeatedly issuing a sequence of a pair of system calls, `select` and `read`. The `select` system call detects events on sockets, and `read` retrieves received messages at sockets and copies them from the kernel to the user process. This polling-based operation tends to require many systems calls, and the delay in timing of the receive operation affects the flow control of TCP/IP.

The problem with sockets can be overcome by designing a new interface, without changing the underlying messaging layers including the TCP/IP layer and the device drivers for Ethernet NIC, because the problem exists in the API between the user process and the kernel. The OS kernels themselves are designed to handle asynchronous events by interrupts, and thus their work is independent of the number of connections they handle. On the other hand, user processes are sequentially modeled, and they are poor at handling asynchronous events.

There are several research and implementation efforts to overcome this problem: user-level communication [5], fast event notification [15, 8, 4], and asynchronous I/O [12] are examples. User-level communication like U-Net [5] reduces the overhead by accessing NIC devices directly from the user space without the intervention of system calls. It is effective but it is not adequate for the commodity environment, because it rewrites the whole messaging layer and requires a huge maintenance effort for systems consisting of hardware and software from multiple vendors. Fast event notification and asynchronous I/O aim at a general-purpose API and have been proved effective in handling the workload of server applications. Although server applications such as Web servers must be able to handle many connections, the usage of connections in server applications differs from that of MPI applications. A small number of connections become active at once in server applications even when the number of connections that are established are large, while almost all connections become active at once when a collective communication is started in MPI applications. General frameworks fail to efficiently handle situations requiring a large number of active connections.

Thus, we have designed and implemented the *O2G driver* to provide a new API to aid implementation of MPI for large-scale clusters. O2G is a driver for asynchronous receive operation, and it just provides an API between the kernel and the user process, because the problem is in the API. O2G runs message queue operations necessary to implement MPI in the interrupt handler, and skips the socket API without modifying the underlying communication layers of TCP/IP and the NIC drivers. The implementation of O2G is very simple and it is provided as a loadable driver module for the Linux kernel, and is thus very easy to deploy.

In the following, we discuss the design, implementation, and evaluation of the O2G driver. First, Section 2 discusses the basic operations necessary to implement MPI, and then discusses the problem of the socket layer. Section 3 shows the design and implementation of the O2G driver. Section 4 shows some benchmark results and compares the scalability of the sockets and of O2G. Section 5 discusses the tradeoff in using O2G. Section 6 shows and compares strategies for overcoming the socket problem, and then we conclude in Section 7.

2 Implementation Issues on a Socket API

2.1 The MPI Communication Model

MPI [10] defines a standard set of communication APIs for parallel computing. MPI provides a set of message send/receive operations and their variants. `MPI_Send` and `MPI_Recv` are blocking operations, that is, they return to the caller when the operations finish. The variants

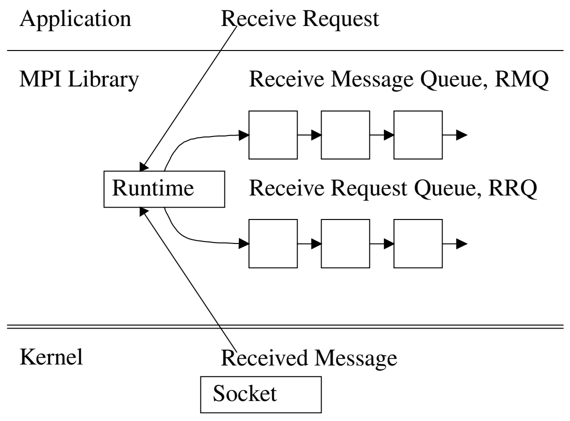

Figure 1. Two Queues used in a Typical MPI implementation

`MPI_Isend` and `MPI_Irecv` are nonblocking operations, that is, they return to the caller immediately and the caller should wait for their completion using other API routines such as `MPI_Wait`. In MPI, a sender sends a message by specifying a data buffer, a destination process, and a tag. Similarly, a receiver receives a message by specifying a data buffer, a source process, and a tag. Tag, source, and destination are all specified by integers. On the receiver side, wild-cards can be used for tags and sources. Matching of tags and sources to destinations is performed on each message.

Typical MPI implementations based on the stream communication layer need to read out every message from the stream continuously for two reasons. First, messages are matched against requests searching for a corresponding tag and source/destination pair, where a matching message may possibly be in the middle of the stream and some uninteresting messages may intervene before the target one. For this purpose, MPI implementations typically manage queues of messages in the library in addition to the communication queues in the kernel. Second, efficient use of the TCP protocol requires messages be read out continuously. TCP is based on a flow control mechanism in which the receiver tells the sender the amount of receive buffer remaining. The size of the receive buffer is relatively small compared to the size of messages sent in typical MPI applications. When messages are not read out, the receive buffer is occupied and the sender stops due to the flow control. The flow control is done end-to-end, and the communication latency between the ends affects the communication performance.

2.2 Receive Queues

In typical implementations of MPI such as MPICH [7] or LAM/MPI [3], basically two queues are used to implement the receive operation. Figure 1 shows the queues: a Receive Message Queue and a Receive Request Queue.

A **Receive Message Queue, RMQ** (sometimes called an *Unexpected Queue*), holds received messages which are still in the pending state, where a corresponding `MPI_Recv`/`MPI_Irecv` is not yet issued. An entry in the RMQ records the information on tag, source, and message content. The MPI library creates a new entry and inserts it in the RMQ when a message is received.

A **Receive Request Queue, RRQ** (sometimes called an *Expected Queue*), holds requests which are ready to receive a message. The requests are in the pending state, where a corresponding `MPI_Recv`/`MPI_Irecv` has been issued but no corresponding messages are yet received. An entry in the RRQ records the information on tag, source, and the address of a receive buffer. The MPI library creates a new entry and inserts it in the RRQ when `MPI_Recv`/`MPI_Irecv` routines are called.

When a receive request is issued by calling `MPI_Recv`/`MPI_Irecv`, the MPI library examines the RMQ searching for a matching entry of a message. If a matching entry exists, the receive request is completed with the found message by copying the content from the entry. On the other hand, if a matching entry does not exist, the request is marked as pending and it is inserted in the RRQ. The inserted entry will be completed with a matching message later, and then the entry will be removed.

Very similarly, when a message is received, the MPI library examines the RRQ searching for a matching entry of an already issued request. If a matching entry exists, the request in the entry is completed with the message by copying the content from the message to the buffer specified in the request. On the other hand, if a matching entry does not exit, the received message is marked as pending and it is inserted in the RMQ. The inserted entry will be completed with a matching request later, and then the entry will be removed.

2.3 Problems of the Socket API

In typical implementations of MPI, a TCP/IP-based one-to-one connection-oriented stream and the socket API are used. The sockets are accessed by the `read`/`write` system calls. Event notification and polling are done by the `select` system call.

Since sockets in OSes like Linux and Unix do not provide asynchronous communication primitives, polling is used instead. Polling is performed by a pair of two system calls, a `select` followed by a nonblocking `read`. A nonblocking `read` operation is enabled by setting the file descriptor in nonblocking mode using the `ioctl` system call.

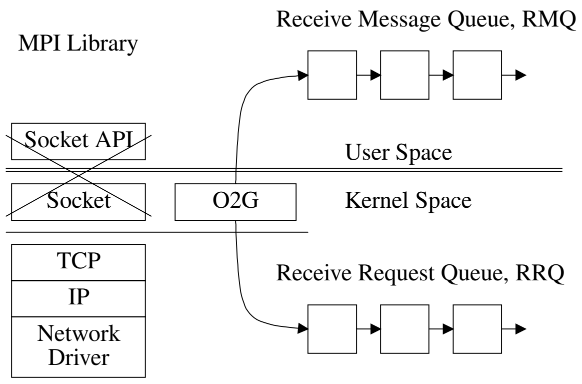

Figure 2. Overview of O2G Operation

MPI implementations based on TCP/IP and Ethernet should consider scalability up to some thousands of nodes, because large-scale commodity clusters have become common, recently. In addition, the Grid environment further requires more scalability where the variance in communication latency and the variance in communication bandwidth are both large [9]. In such an environment, the socket API has the following major problems.

First, implementations based on polling require frequent issues of system calls. The `select` system call should be called even when no message is ready. And also, nonblocking `read` returns from the system call without reading the message to the end by definition, and thus needs many more issues of system calls. Generally, system calls have large overhead to switch contexts between the kernel and the user process, and this affects the performance.

Second, implementations based on polling using the `select` and `read` system calls cannot read out the stream constantly and cannot satisfy the requirement of TCP mentioned in the previous discussion. This is because the read out is limited at the time of polling, and the delay causes the TCP buffer to become full and triggers the flow control which affects the performance.

Third, it may be just an implementation issue, but the `select` system call is typically implemented as an operation whose computation cost grows linearly with the number of sockets. The cost becomes unacceptable in thousand node clusters.

3 O2G Driver

3.1 Design of the O2G Driver

O2G optimizes the receive operation by immediately retrieving received messages from the communication layer to the user space. To attain this, O2G implements the operations on both the RMQ and RRQ queues, which are described in Section 2, in the driver of the Linux kernel. The operations are performed in the protocol handler invoked by interrupts, so that messages are processed immediately. O2G is provided as a loadable Linux driver module, and it does not require patching or reconfiguring the kernel code, nor rebooting the OS at all.

Figure 2 shows an overview of the operation of O2G. Since the entries in the RMQ include the data space of message contents, buffers for them are taken in the user space. On the other hand, since the entries in the RRQ are used to scan a match when a message is received, they are held in the kernel space. The entries do not include the message data and they are small in size and consume little memory in the kernel.

A received message is handled ordinarily in the kernel's protocol stack, where the message is copied into a buffer called *SKB*, which is a buffer structure used to manage communication protocols in Linux. Normally, a message copied in an SKB is put in the receive buffer and kept there until a user process issues the `read` system call. On the other hand, an SKB put in the receive buffer is handled immediately with O2G. O2G matches a message against the entires in RRQ, and then copies the content out to the user space if a match is found. Or, a new entry is created and inserted to the RMQ if a match is not found. The operation is done in the interrupt handler on receiving a message and has no delay. The work of O2G consists of decoding an MPI header, searching for a matching entry in the RRQ, and creating an entry for the RMQ. Copying the message content itself is performed similarly as an ordinary `read` system call on sockets. The cost of these operations is low and they are adequate to be run in the interrupt handler.

3.2 The O2G Driver Interface

O2G is controlled from a user process by the `ioctl` system call, because the interface of O2G is provided as a device driver. Figure 3 shows the library routines which wrap the driver interface.

```
/*Initialization*/
o2g_init(int n_socks);
o2g_register_socket(int sock, int rank);
o2g_set_dump_area(void *area, int size);
o2g_start_dumper_thread(int n_thrds);
/*Queue Control*/
o2g_put_entry(struct queue_entry *e);
o2g_cancel_entry(struct queue_entry *e);
o2g_free_entry(struct queue_entry *e);
o2g_poll(void);
```

Figure 3. User API of the O2G Driver

```
{
  /*Register a hook function*/
  struct sock *sk = ...;
  sk->data_ready = data_ready;
}
void data_ready(struct sock *sk,
    int len) {
  tcp_read_sock(sk, ..., data_recv);
}
int data_recv(..., struct sk_buff *skb,
    unsigned int off, size_t len) {
  char *buf = ...;
  skb_copy_bits(skb, off, buf, len);
}
```

Figure 4. Hook Function in the Socket and its Usage

The initialization routines open the O2G device and register the sockets to use with O2G. `o2g_init` opens and initializes the device. `o2g_register_socket` associates a given socket and its peer *rank* (process number) and marks the socket for use with O2G. `o2g_set_dump_area` tells the device the location of the buffer area in the user space for creating RMQ entries. `o2g_start_dumper_thread` starts some number of threads, which are used to dump message content to the user space in case the interrupt handler cannot handle the received message by itself. The usage of the threads is explained later.

The queue control routines manage the RRQ and RMQ queues. `o2g_put_entry` inserts an entry in the RRQ. `o2g_cancel_entry` cancels an entry once inserted by `o2g_put_entry`. `o2g_free_entry` frees an entry in the RMQ which resides in the user space when it is completed. `o2g_poll` is used to wait for an event, and it blocks the process until a message is received. It is implemented with an ordinary `select` system call on the O2G device, but is scalable because it has to wait for the single O2G device.

3.3 Receive Event Hook

O2G is invoked inside the communication protocol stack via a hook function registered in the socket structure. Linux has an implementation of NFS in the kernel, and it has a mechanism to invoke routines of the SUN RPC [14] in the kernel. An arbitrary hook function can be registered in the socket structure, which is called when the protocol stack handler passes a message to the socket layer. The hook is a function pointer `data_ready` in the socket structure `sock`.

Figure 4 shows an example of the use of a hook in the socket structure. This example copies the message content in an SKB to a buffer `buf` allocated in the kernel space. A `data_ready` procedure is called every time an SKB is passed to the socket layer, and `data_recv` actually copies the content, where `tcp_read_sock` is a utility procedure used to ease handling of an SKB. As shown, Linux provides a very easy way to handle messages in the protocol stack, and O2G utilizes this mechanism.

3.4 Handling Context Mismatch

O2G performs receive processing in the protocol stack, and this is executed by the interrupt handler. Thus, if the context of the interrupted process does not match the user process, the protocol handler cannot access the user space directly. To deal with this case, O2G delegates the work to the pre-started dumper threads. Dumper threads are started by `o2g_start_dumper_thread` at the initialization. Similarly, it delegates the work when page faults occur in writing to the user space. These context switches due to context mismatch are equally necessary in the `read` system call on sockets. The difference between `read` and O2G is that `read` wakes up a process blocked in the `read` system call. The needed work is equivalent and the cost is not unique to O2G.

3.5 Avoiding Race Conditions

Since the RRQ and RMQ are accessed from both the protocol hander in the kernel and the user process through the O2G's API, they have a race condition. The interface to O2G takes account of a race condition and may fail due to a potential race. For example, the API routine `o2g_put_entry` puts an entry in the RRQ. Although it should be called after checking that a matching message is not yet received, it is possible a message may come at the time the `o2g_put_entry` is called. If a potential race has been detected, the interface routine returns EAGAIN to indicate an error. The conventions of Unix-like OSes are such that EAGAIN forces to retry a system call again. When the MPI library gets an EAGAIN from the O2G driver, it checks the RMQ and then calls the O2G driver again after assuring that no true race condition exists. `o2g_cancel_entry` has a similar race condition and may return EAGAIN, too.

Table 1. PC Cluster Specification

Node PC	
CPU	Dual Xeon 3.06GHz, 512KB Cache
Chipset	Intel E7501
Mother Board	SuperMicro X5DPR-iG2+
Memory	4GB DDR266
NIC	Intel 82546EB
OS	RedHat 8 (Linux-2.4.24)
NIC Driver	Intel e1000 5.2.2
Network	
Switch	Force10 Networks E1200

4 Evaluation

4.1 The Experimental Setting

To evaluate the effectiveness of O2G, we compared an MPI implementation using O2G and one using sockets. In the experiment, we used YAMPII [16], which is a portable and full MPI-1.2 implementation developed by us that uses sockets. In the following, we distinguish between the two implementations by referring the socket version as YAMPII/Sock, and the O2G version as YAMPII/O2G. These two implementations share almost all the same code, but they differ only in queue management. Inserting to and scanning in the queues are performed in the user library in YAMPII/Sock but in the driver in YAMPII/O2G.

In the experiment, we used a medium-scale cluster with 256 Xeon PCs. Table 1 shows the specification of the PC and the network switch used. While each PC is a dual processor machine, but is used as a uni-processor machine. The network consists of a single big nonblocking switch and all PCs are attached to it directly. The Linux kernel is version 2.4.24. The compiler is GCC version 3.3.3 for both Fortran and C, and all programs are compiled with the -O3 optimization option. The buffer size of TCP was set to 128KB in the benchmark to equate the condition. In the evaluation, we set the TCP option, `TCP_NODELAY`, which disables Nagle's algorithm that delays a send for small messages. Note that all implementations of MPI used in the experiment set this option.

4.2 The Bandwidth Benchmark

Scalability is the major aim of the development of O2G. In order to verify it, bandwidth measurement was performed in a simulated environment of up to a thousand nodes. We measured the bandwidth varying the number of connections and sockets. An MPI implementation needs to receive messages from all nodes, and it is required to poll all the

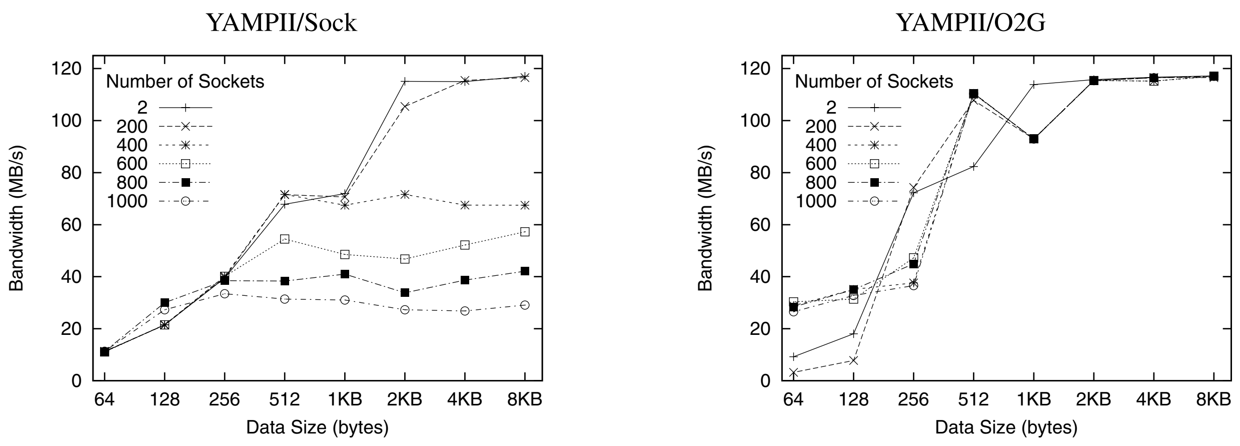

Figure 5. Bandwidth Performance to Number of Sockets

connections. Thus, it is estimated that the number of connections affects the performance.

In this experiment, bandwidth is measured between the two nodes, and the third node is used to simulate a thousand nodes which makes many connections up to a thousand to the first two nodes. The third node only makes connections but does not carry on any communication.

Figure 5 shows the bandwidth when the number of connections are varied. YAMPII/Sock and YAMPII/O2G are shown in different graphs. The left graph shows degradation in performance in YAMPII/Sock as the number of connections increases, while the right one shows stable performance in YAMPII/O2G. Here, both use the same TCP/IP protocol stack but only differ in the usage of the socket layer. This result shows that while the Linux kernel itself is scalable and its performance is not affected by the number of connections, the socket API is greatly affected.

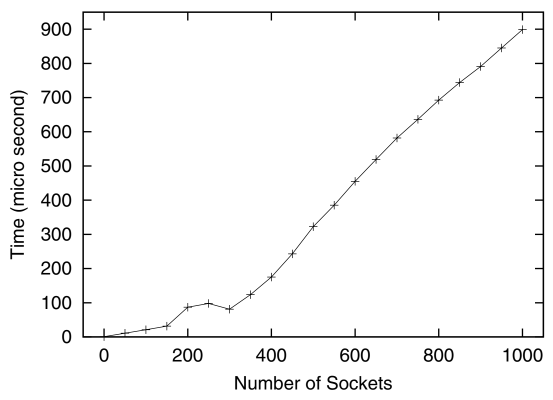

Figure 6. Select System Call Time by Number of Sockets

4.3 Select System Call Overhead

The previous experiment has shown the number of connections affects the socket API, and thus, we measured the performance of the `select` system call directly. With the same settings as those of the experiment, we measured the time taken for the `select` system call by varying the number of connections. The CPU clock counter was used to measure the time, and the minimum from ten runs was used. In the experiment, all sockets have no data and are inactive.

Figure 6 shows the result. The time is about 1 sec for each socket, but it grows almost linearly with the number of connections. The breakdown of processing of the `select` system call consists of taking a file structure from a file descriptor (an integer), taking a socket structure from it, taking a structure for the TCP layer, and then accessing it. And this is carried out on the sockets one by one. The socket structure is mutually excluded during its access. The polling action itself is very simple, and it only checks the flag and the sequence number in the structure to see whether the data exists to process. There is another `poll` system call which is an extension of the `select` system call. The time taken by the `poll` system call should be larger than the `select`, because it works in the same way as the `select`, but it passes more arguments than the `select`.

This experiment implies that each polling takes about 1 milli-second in a thousand node cluster. A receive operation in the MPI library consists of at least two pollings, one for the header and one for the body of the message, and 2 milli-second is needed. This 2 milli-second is pure overhead and is relatively large considering that over 200KB can be transferred in 2 milli-second on giga-bit class networks.

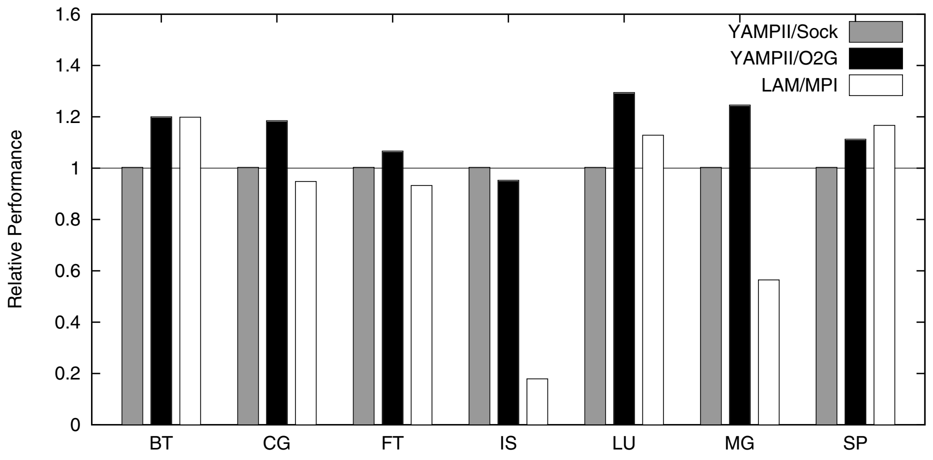

Figure 7. NPB (CLASS=B, NPROCS=256) Benchmark Result

Table 2. NPB (CLASS=B, NPROCS=256) Benchmark Result

('Mop/s total' Value)

	YAMPII /O2G	YAMPII /Sock	LAM/MPI
BT	49598.25	41430.83	49532.39
CG	3334.20	2820.92	2664.98
FT	12279.43	11545.06	10731.99
IS	153.15	161.25	28.37
LU	32359.09	25056.89	28192.26
MG	22450.11	18065.15	10146.55
SP	15450.30	13921.89	16202.92

4.4 NPB Benchmarks

We used the NPB (NAS Parallel Benchmarks) [1] (version 2.3) in the evaluation, which is a suite of benchmarks from NASA and derived from their aero-space simulation code. The data set size used is CLASS B, and the number of processes is 256. The benchmark suite consists of eight programs from three pseudo-applications, BT (Block Tridiagonal ADI), LU (LU Factorization), and SP (Scalar Pentadiagonal ADI), and from five kernels, CG (Conjugate Gradient), FT (Fourier Transform), EP (Embarrassingly Parallel), IS (Integer Sort), and MG (Multi-Grid Method). It is commonly used in benchmarking MPI implementations. The EP benchmark is omitted from our experiments, because EP rarely communicates and its result is apparent.

LAM/MPI [3] is a widely used free implementation of MPI, and we included it in the comparison to see if YAMPII is comparable in the quality of implementation. In the experiment, we used the most recent version available at the time, LAM/MPI 7.0.4. MPICH [7] is another widely used free implementation of MPI, but we omitted it from the comparison because it is consistently slower for all benchmarks than other implementations.

Figure 7 shows the performance relative to YAMPII/Sock. The performance is measured by the *Mop/s total* value. Table 2 lists the absolute values for reference. It shows YAMPII is comparable to a much more mature MPI implementation.

YAMPII/O2G is 5 to 30 percent faster than YAMPII/Sock except for the IS benchmark. Since the IS benchmark uses all-to-all collective communication, it is very sensitive to the pattern of communication in this scale of a cluster (256 nodes). Congestion of TCP traffic seems to be greatly influenced by the pattern of communications and the algorithms for collective communications used. We regard this is the reason for both the low performance of LAM and the lower performance of YAMPII/O2G in the IS benchmark.

The code of O2G is naive and immature compared to the optimized and frequently-improved socket layer of Linux. For example, the socket layer of Linux incorporates a short-cut for SKB queuing when the sequence number is known to be consecutive to that for the messages received so far. Also, it fakes the interrupt context by setting the flag temporarily. Although the improvement using O2G may seem not very large for 256 nodes, we can well expect it should be much more for a thousand nodes from the experiment on the overhead of the `select` system call.

5 Discussion

5.1 Performance Trade-offs of O2G

Received messages may be buffered in the RMQ, when they reach the destination before a matching receive operation is issued. In this case, two copies are needed, once to buffer the message and once to consume the message. Two copies are an overhead. A message is immediately processed in O2G, while processing is delayed until poll time using sockets. This early processing in O2G has a preferable effect to the flow control of TCP, but may increase the possibility of unmatched messages and extra copying.

O2G chooses the early processing, preferring asynchronous processing to extra copying, because recent machines have high memory bandwidth and keeping network performance high is more of a concern.

5.2 Polling Messages

The MPI specification [10] discusses the progress of processing. This means that an application program may need to insert some MPI library functions to perform polling at appropriate points in the program. In polling-based MPI implementations, no message processing is performed until the application program calls MPI library functions. Thus, in the worst case, deadlock may occur when the application calls MPI functions rarely, because of consumption of buffer space occupied by unprocessed messages. The MPI specification requires that MPI library functions be called appropriately. On the other hand, messages are processed in the interrupt handler in O2G, and this assures progress of processing implicitly.

5.3 Message Send Optimization

O2G only modifies the receive operation and does not change the send operation. We considered that the ordinary asynchronous I/O operations [12] can be used for the sender side. On the receiver side, the handing of RRQ and RMQ are necessary and the asynchronous receive provided in the ordinary asynchronous I/O is not applied directly. Thus we designed a new driver. The sender is required only to `write` a message, and the ordinary asynchronous I/O operation suffices.

Although it is beyond API issues, the sender side should be involved in the implementation specific behavior of TCP more deeply to adapt to the high performance Ethernet environment. We observed an almost ten-fold difference using the all-to-all collective communication algorithms in the IS and FT benchmarks. Performance is very much affected by the algorithms and it is due to the interaction between the congestion control of TCP and the pattern/timing of communication. In practice, we observed that an apparently poor algorithm that has a barrier-like synchronizing behavior sometimes performed better.

6 Related Work

6.1 Optimization of Select

devpoll [15] is a mechanism provided in Solaris. It uses `/dev/poll` device, and limits the number of sockets to search for events. `poll`, a system call introduced to remove the limitation of the `select`, has enlarged the overhead of copying parameters between the user process and the kernel in order to pass more information. `devpoll` eliminates the overhead.

epoll, recently introduced to the Linux kernel, is similar in functionality but is different in the interface.

kqueue [8] is implemented in some BSD-line Unix systems. It filters events to be notified to users by an *eventlist*. Event search is limited to the sockets with the events, which make the search fast. It makes the search fast, but underlying processing runs a filter and spreads the overhead around into the protocol processing.

Also, using a realtime signal for asynchronous event notification is discussed and its effectiveness is shown by Chandra, et al [4].

6.2 Asynchronous I/O

Asynchronous I/O is an operation which returns control immediately, before completion of the system call. This enables the issuance of background I/O operations behind the processing of a user program. This assumes high volume I/O operations.

aio_read/aio_write are defined in the real time extension of POSIX [12], and provided in Linux and some Unix systems. **aioread/aiowrite** are one provided in Solaris. Both provide nonblocking I/O system calls, and are very similar except in the details.

6.3 Remote DMA Write

RDMAP [13] is a standard protocol designed to achieve remote write DMA over TCP/IP. It is designed to reduce the copy overhead to improve communication performance.

6.4 Driver Level MPI Implementation

Myricom designed MX [11], which provides an MPI-like model at the Myrinet driver level. While this is not on the ordinary protocol stacks, but an implementation of MPI using this driver may be similar to one using O2G.

6.5 Advantages of O2G

Optimization of `select` is introduced mainly to lighten parameter passing of the system call. O2G does not use polling at all and does not have a problem with parameter passing. Also, the optimization of `select` achieves its effect by limiting the number of sockets handled at one time. However, in MPI implementations, many sockets become active in normal operations. The effectiveness of the optimizations of `select` is not clear in this situation. Since O2G is a driver and directly accesses the messages, it does not need to reduce the overhead of sockets to attain performance. The use of optimization of `select` or asynchronous I/O essentially does not reduce the number of system calls needed to read messages. In MPI, it is necessary to issue at least two `read` operations for a message to receive a header followed by a body.

In addition, asynchronous I/O cannot attain immediate processing of received messages of O2G, because normally `read` for the body can only be issued after processing the header. Otherwise, buffering of the message body is necessary because its size is not known in advance, and this introduces extra cost to manage the buffer for handling messages. Thus, the basic behavior is the same as that of polling.

7 Conclusion

This paper has shown the design and implementation of the O2G driver, which optimizes the receive operation targeted for large-scale commodity clusters. O2G skips the system call API which requires large overhead in proportion to the number of connections, without any modification to the underlying TCP/IP protocol stacks and NIC drivers. O2G implements queue processing necessary to implement MPI in the protocol handler invoked by interrupts.

The bandwidth measurement in a simulated environment with a large number of connections shows the effectiveness of O2G. In using O2G, the performance is almost independent of the number of connections, while the sockets are effected by the number of connections even though almost all of the sockets are inactive. Evaluation with the NPB benchmark suite shows that the asynchronous communication and overhead reduction is effective.

Advances in the technology of CPUs and NICs have changed the balance of performance, and the combination of Linux, Ethernet, and TCP/IP is now a viable platform for a large-scale high performance cluster. In contrast to the base technology of OSes, the programing model and the API incur a large overhead to scale to a thousand nodes. It is an old problem, but no general solution is given yet. Though O2G is specialized towards the implementation of MPI and it is of limited applicability, it is a solution to the problem and we have shown its effectiveness.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. International Journal of Supercomputer Applications, 1995. http://www.nas.nasa.gov/Software/NPB

[2] G. Banga and J. C. Mogul. Scalable Kernel Performance for Internet Servers under Realistic Loads. Proc. of the 1998 USENIX Annual Technical Conference, 1998

[3] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. Proc. of Supercomputing Symposium, pp.379–386, 1994. http://www.lam-mpi.org

[4] A. Chandra and D. Mosberger. Scalability of Linux Event-Dispatch Mechanisms. Hewlett Packard Laboratory, HPL-2000-174, 2000.

[5] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. Proc. of the 15th ACM Symposium on Operating Systems Principles, 1995.

[6] The GridMPI Home Page. http://www.gridmpi.org

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing, Vol.22, No.6, pp.789–828, 1996.

[8] J. Lemon. Kqueue: A Generic and Scalable Event Notification Facility. BSDCon 2000, pp.141–154, 2000.

[9] M. Matsuda, T. Kudoh, and Y. Ishikawa. Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment. Proc. of The 3rd International Symposium on Cluster Computing and the Grid (CCGrid2003), pp.10–17, 2003.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, May 5, 1994*. University of Tennessee, Knoxville, Report CS-94-230, 1994.

[11] Myricom, Inc. Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet. MX API Reference, 2003. http://www.myri.com/scs/MX/doc/mx.pdf

[12] POSIX. POSIX 1003.1 (Realtime Extensions). IEEE POSIX Std. 1003.1-2001, 2001.

[13] RDMA Consortium.
http://www.rdmaconsortium.org

[14] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Internet Engineering Task Force, 1995.

[15] Sun Microsystems. Solaris Manual Pages. poll(7d).

[16] YAMPII, Yet Another MPI Implementation.
http://www.ilab.is.s.u-tokyo.ac.jp/yampii