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
Background (1)

- Large scale commodity clusters (1,000 nodes)
  - Linux + TCP/IP + Ethernet is variable
    - Ethernet: Large scale non-blocking switches
    - TCP/IP: Processing overhead is modest now
  - Good MPI implementation is needed

- Demand for asynchronous handling of messages
  - MPI has new applications in wide-area communication
    - Read operation only takes a small amount of data each time from a slow link

- Old Socket API
  - Polling-based API (behavior is synchronous/serialized)
  - Overhead proportional to #connections
Background (2)

Problems of Sockets in implementing MPI library
- Frequent system calls
  - Loop with a pair of `select` and `read`
  - Serialize receive processing
- Large overhead on large scale clusters

Design a simple kernel module for MPI
- MPI message handling in the interrupt handler
  - Bypass Socket API
  - No changes to TCP/IP layers and NIC drivers
- Loadable driver module of Linux
  - No kernel reconfiguration, no rebooting needed
Outline

- Basics: MPI Communication Model and Typical MPI Implementation
- Issues on Socket API
- Design and Implementation of O2G Driver
- Performance Evaluation
- Related Work (brief)
- Summary
**MPI Communication Model**

- **Basic operations of MPI**
  - MPI_Send(buf, size, datatype, destination, tag, comm)
  - MPI_Recv(buf, size, datatype, source, tag, comm, status)

- **Matching of messages**
  - **Sender**: specify destination process and tag
  - **Receiver**: specify source process and tag

**Message Matching**: Messages are exchanged between MPI_Send and MPI_Recv when a source and destination and a tag match
Typical MPI Implementation

- Two queues
  - **Message Queue** (Unexpected Queue)
    - Hold received messages, whose matching `MPI_Recv` is not yet issued
  - **Request Queue** (Expected Queue)
    - Hold `MPI_Recv` requests, which is pending for an unreceived message

- `MPI_Recv` only receives matching messages
- Unmatched messages/requests are queued
Issues on Sockets (1)

- Socket: Communication end-point
  - Communication API of OSes: Linux, Unix, (Windows)
  - Connection oriented stream (over TCP/IP)

- Receiver should repeatedly read out communication stream for matching messages
  - Search for a message whose tag matches
  - Need asynchronous receive of messages
    - Sometimes independent from program specification
  - Delay of receive operation affects TCP/IP flow control
    - Sensitive to latency, because of end-to-end control

- Socket API is not designed for MPI
Issues on Sockets (2)

- Loop with `select` and `read` for asynchronous receive
  - Polling results in frequent system calls
    - Serialize processing on each Socket
    - Serialize header decoding followed by body receive
  - Sockets are set to *non-blocking mode*
    - Read finishes prematurely, still increases system calls
- Implementations do not support large number of Sockets
  - Overhead is proportional to #connections
  - Overhead is for connected connections (not active ones)
MPI Implementation with Sockets

Key Points:
- Loops with `select` and `read` system calls
  - `select` detects a socket with receive data
  - `read` extracts data from kernel space to user space
- Source/destination and tag matching is done in user space

Diagram:
- User Space
  - Message Queue
  - User Process
  - Select & read
- Kernel Space
  - Protocol Stack
    - Socket
    - Queue
  - TCP
  - IP
  - NIC
  - Protocol Handler
Design of O2G Driver (1)

- O2G driver bypasses Socket layer
- Queue operations are in protocol handler
  - Match tag and source/destination
  - Write message data directly to user space
  - Internal work is similar to `read`, matching is extra

```
Design of O2G Driver (1)

O2G driver bypasses Socket layer
Queue operations are in protocol handler
  - Match tag and source/destination
  - Write message data directly to user space
  - Internal work is similar to `read`, matching is extra
```

![Diagram of O2G Driver](image_url)
Design of O2G Driver (2)

Linux driver module
- Data is immediately handled by the interrupt handler
- Data is written to user space directly

Diagram:
- User Space
- Kernel Space
- Message Queue
- Request Queue
- Socket API
- TCP
- IP
- Network Drivers
- O2G

Message
Source
Tag
Content data

Request
Source
Tag
Buffer address

Invoked thru a Hook
Implementation of O2G (hook function)

- O2G driver is invoked thru a hook function
  - Hook is for Kernel NFS (Network File System) in Linux
  - Hook is set per socket, called for each packet

Hook usage example:

```c
{
    /* Register a hook */
    struct sock *sk = ...;
    sk->data_ready = data_ready_fn;
}
void data_ready_fn(struct sock *sk, int len) {
    tcp_read_sock(sk, ..., data_recv_fn);
}
int data_recv_fn(..., struct sk_buff *skb, int off, int len) {
    char *buf=...;
    skb_copy_bits(skb, off, buf, len);
}
```
Implementation of O2G (API)

API of O2G

- Library API to wrap IOCTL of the driver
- Initialization
- Receive request queue operation

/* 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);

/* Request entry API */
o2g_put_entry(struct queue_entry *e);
o2g_cancel_entry(struct queue_entry *e);
o2g_free_entry(struct queue_entry *e);
o2g_poll(void);
Subtle Issues on O2G Driver

- **Process context mismatch handling**
  - Interrupt handlers sometimes cannot write user space
    - Delegate write processing to a pre-started user thread
    - Handle page fault in user space in the same way

- **Race condition avoidance**
  - Request queue is processed from both user process and interrupt handler
    - Race condition is detected by recording last few messages
    - O2G driver returns EAGAIN at detecting race condition
    - `o2g_put_entry, o2g_cancel_entry`
Evaluation: Base MPI Library

- **Base MPI System: YAMPII**
  - Developed at Univ. of Tokyo
  - Full MPI-1.2 (MPI-2.0 under development)
  - Full scratch, LGPL license
  - Supported communication layers
    - Socket (TCP/IP)
    - Myrinet (SCore-PM cluster system)

- Queue management code is modified for O2G
  - “YAMPII/Sock”: Socket version (original code)
  - “YAMPII/O2G”: O2G version
Evaluation: Setting

- Overhead reduction
  - Point-to-point bandwidth
  - Time spent in `select` system call
  - NPB (NAS Parallel Benchmarks)

**Benchmark Setting**

<table>
<thead>
<tr>
<th>AIST Super Cluster (F32 Cluster, 256 Node)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node</strong> (use 1CPU each node)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>OS</strong></td>
</tr>
<tr>
<td><strong>Network</strong> (Non-blocking switch)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
</tr>
</tbody>
</table>
Evaluation: Bandwidth (1)

- Bandwidth varying #connections
  - Artificially add connections
  - Vary #connection from 2 to 1,000
  - Third node (Node#2) makes dummy connections, but does not perform any communication

![Diagram]

- Measure bandwidth between Node0 and Node1
- Node0 sends messages to Node1 using MPI_Send
- Node1 receives messages from Node0 using MPI_Recv
- Node2 makes dummy connections (0 to 988)
- Node2 does not perform any communication
Evaluation: Bandwidth (2)

- Point-to-point, uni-directional
  - Message size: 64B to 8KB (X-axis log-scale)
  - O2G is not affected by #connections
  - Socket also performs well up to 200 connections

YAMPII/Sock

YAMPII/O2G
Evaluation: Select Time

- Time spent in **select** system calls
- Vary #connections, no message data
- Measurement by CPU clock counter (min from 10 trials)
Evaluation: NPB Benchmarks (1)

NPB (NAS Parallel Benchmarks, Ver 2.3)

- 256 Nodes NPROCS=256, Data set size CLASS=B
- Relative Performance to YAMPII/Sock (Mops/total)

![Bar chart showing relative performance comparison between different configurations.](chart.png)
Evaluation: NPB Benchmarks (2)

- NPB (NAS Parallel Benchmarks, Ver 2.3)
  - 256 Nodes NPROCS=256, Data set size CLASS=B
  - Add LAM/MPI performance for comparison
Related Work

- Overhead reduction of **select**
  - **kqueue** (FreeBSD)
    - Filter events (eventlist filter)
  - **devpoll** (Solaris) (**epoll** in Linux kernel 2.6)
    - Confines sockets to check, with a device /dev/poll

- Asynchronous I/O (**aio_read/aio_write**)
  - Large data I/O in background processing

- **O2G advantage**
  - devpoll (kqueue) needs as many as system calls
    - Behavior is still polling based
  - Asynchronous I/O needs as many as system calls
    - Body read can be issued after header decode
Summary

- MPI optimizing driver for large scale commodity clusters
  - Simple & straightforward O2G driver module
  - Overhead reduction of system calls
  - Asynchronous behavior
  - No changes to TCP/IP layers and NIC drivers
  - Most queue operations of MPI done in interrupt handler

- Evaluation in middle scale cluster with Ethernet
  - Performance independent to #connections
  - Observe 10% to 20% speed up in 256 nodes

- Future work
  - Evaluation of merit of asynchronous behavior
  - Design of an abstract interface, currently most part is YAMPII specific
END