MPI and MPICH on Clusters


Rusty Lusk
Mathematics and Computer Science Division
Argonne National Laboratory

Extended by D. D’Agostino and A. Clematis
using papers referenced hereafter
MPI Implementations

- MPI’s design for portability + performance has inspired a wide variety of implementations
- Vendor implementations for their own machines
- Implementations from software vendors
- Freely available public implementations for a number of environments
- Experimental implementations to explore research ideas
MPI Implementations from Vendors

- IBM for SP and RS/6000 workstations, OS/390
- Sun for Solaris systems, clusters of them
- SGI for SGI Origin, Power Challenge, Cray T3E and C90
- HP for Exemplar, HP workstations
- Compaq for (Digital) parallel machines
- MPI Software Technology, for Microsoft Windows NT, Linux, Mac
- Fujitsu for VPP (Pallas)
- NEC
- Hitachi
- Genias (for NT)
From the Public for the Public

- MPICH

- LAM
  - www.lam-mpi.org/

- OPENMPI
  - http://www.open-mpi.org/
Experimental Implementations

- Real-time (Hughes)
- Special networks and protocols (not TCP)
  - MVAPICH (Infiniband, CSE Ohio)
  - MPI-FM (U. of I., UCSD)
  - MPI-BIP (Lyon)
  - MPI-AM (Berkeley)
  - MPI over VIA (Berkeley, Parma)
  - MPI-MBCF (Tokyo)
  - MPI over SCI (Oslo)
- Wide-area networks
  - Globus (MPICH-G, Argonne)
  - Legion (U. of Virginia)
  - MetaMPI (Germany)
Misconceptions About MPICH

- It is pronounced (by its authors, at least) as “em-pee-eye-see-aitch”, not “em-pitch”.
- It runs on networks of heterogeneous machines.
- It can use TCP, shared-memory, or both at the same time (for networks of SMP’s).
- It runs over native communication on machines, not just TCP.
- It is not for Unix only, but supports also Win32 and Win64.
- It runs MIMD parallel programs, not just SPMD.
MIMD and SIMD

- Separate (collaborative) processes are running. Exchanging information is done explicitly by calling communication routines.

- MIMD (multiple instruction multiple data) model allows different processes:
  - $a_0.out$ for process 0, $a_1.out$ for process 1, … $a_i.out$ for process $i$.

- SPMD (single program multiple data) model uses one executable
  - All processes run $a.out$. 

MPICH Architecture

MPI_Send -> MPI Routines

MPID_SendContig -> ADI Routines

MPD daemons ring -> Various Devices

ch_p4, ch_p4mpd, ch_shmem, ch_NT, Globus

Above the Device

Below the Device
(implementations of the channel interface)
The Second-Generation ADI for the MPICH Implementation of MPI
by William Gropp   Ewing Lusk

In this paper we describe an abstract device interface (ADI) that may be used to efficiently implement the Message Passing Interface (MPI). After experience with a first-generation ADI that made certain assumptions about the devices and tradeoffs in the design, it has become clear that, particularly on systems with low-latency communication, the first-generation ADI design imposes too much additional latency. In addition, the first-generation design is awkward for heterogeneous systems, complex for noncontiguous messaging, and inadequate at error handling. The design in this note describes a new ADI that provides lower latency in common cases and is still easy to implement, while retaining many opportunities for customization to any advanced capabilities that the underlying hardware may support.
Our goal is to define an abstract device interface (ADI) on which MPI [1] can be efficiently implemented.

- This discussion is in terms of the MPICH implementation, a freely available, widely used version of MPI.
- The ADI provides basic, point-to-point message-passing services for the MPICH implementation of MPI.
- The MPICH code handles the rest of the MPI standard, including the management of datatypes, and communicators, and the implementation of collective operations with point-to-point operations.
- Because many of the issues involve understanding details of the MPI specification (particularly the obscure corners that users can ignore but that an implementation must handle), familiarity with Version 1.1 of the MPI standard is assumed in this paper [2].
ADLI - Principles

The design tries to adhere to the following principles:

- No extra memory motion, even of single words. A single cache miss can generate enormous delays (on the order of one microsecond in some systems).
- Minimal number of procedure calls.
- Efficient support of MPI_Bsend, particularly for short messages.
- The common contiguous case should be simple and fast.
- The general data structure case should be managed by the ADI, even if it involves simply moving data into and out of contiguous data buffers (putting those buffers under control of the ADI).
- The ADI should be usable by itself (without the entire MPI implementation), simplifying testing and tuning of the ADI.
int MPI_Send( ... )
{
  ...

  #if !defined(MPID_HAS_HETEROGENEOUS)
    if (datatype->is_contig) {
      MPID_SendContig( ... &error_code );
    } else
  #endif

  {
    MPID_SendDatatype( ... &error_code );
  }

  return error_code;
}

The contiguous user-buffer case is an important special case, and one that is handled by all hardware.

This design provides special entry points for operations on contiguous messages.
The basic contiguous send routine has the form

\[
\text{MPI\_SendContig( comm, buf, len, src\_lrank, tag, context\_id, dest\_grank, msgrep, &error\_code )}
\]

\textbf{comm} \\
Communicator for the operation (struct MPIR\_COMMUNICATOR *, not MPI\_Comm).

\textbf{buf} \\
Address of the buffer.

\textbf{len} \\
Length in bytes of the buffer.

\textbf{src\_lrank} \\
Rank in communicator of the sender (source local rank).

\textbf{tag} \\
Message tag.

\textbf{context\_id} \\
Numeric context id.
**dest_grank**
Rank of the destination, as known to the device. Usually this is the rank in MPI_COMM_WORLD but may differ if dynamic process management is supported.

**msgrep**
The message representation form. This is an MPID_Msgrep_t (enum int) data item that will be delivered to the receiving process. Implementations are free to ignore this value.

**error_code**
The result of the action (an MPI error code). This may be set only if the value would not be MPI_SUCCESS. In other words, an implementation is free to not set this unless there is an error. The error code was made an argument so that an implementation as a macro would be easier.

Similar versions exist for the other MPI send modes (i.e., MPID_SsendContig and MPID_RsendContig).

One point that deserves discussion is the use of a global rank for the destination instead of a rank relative to the communicator. The advantage of using a global rank is that many ADI implementations can then ignore the communicator argument (except for error processing). In addition, for communicators congruent to MPI_COMM_WORLD, no translation from relative to global ranks is needed.
Multi-Purpose Daemon (MPD)

- Fast startup of jobs
- Error resistance
- Scalability

<table>
<thead>
<tr>
<th></th>
<th>2 nodes</th>
<th>4 n.</th>
<th>8 n.</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>0.18 s</td>
<td>0.54 s</td>
<td>1.27 s</td>
</tr>
<tr>
<td>P4MPD</td>
<td>0.12 s</td>
<td>0.12 s</td>
<td>0.13 s</td>
</tr>
</tbody>
</table>

Scheduler

mpirun
Parallel I/O for Clusters

- ROMIO is an implementation of (almost all of) the I/O part of the MPI standard.
- It can utilize multiple file systems and MPI implementations.
- Included in MPICH and other implementations

A good combination for Clusters: MPICH + the Parallel Virtual File System (PVFS)
Objective

Model: one server one disk

- Complex management
- Uneven use of the disks space
- Non-transparent access to the data
- Low I/O performance
Parallel vs Distributed File Systems

Parallel file systems allow:
- the file sharing among different computational nodes as the DFSs
- to achieve higher performance w.r.t. DFSs

An example.
A task of 8 processes have to read, globally, 512 MB. The computational part requires 5.3 seconds. If the data are acquired with **NFS** the execution time is of **48.2** seconds, with **PVFS** of **9.36** seconds.
Data Striping

PVFS subdivides files in blocks of 64 KB and stores them using a round robin distribution.

This default strategy may be modified by users.
How to compile

- `mpicc <parameters> –o <executable> <source>`
  
  It is a wrapping of the C/Fortran compiler, therefore it accepts the same list of parameters (e.g. –O3, -g ...)

```bash
user@cluster1:~$ mpicc -v -o ring ring.c
...
mpicc for 1.0.4p1
Thread model: posix
....
gcc version 4.1.0 20060304 (Red Hat 4.1.0-3)
....
/usr/libexec/gcc/i386-redhat-linux/4.1.0/cc1 -quiet
-v -I/home/mpich2/include ring.c -quiet -dumpbase ring.c
....
```
A simple Makefile

### Options
CC=gcc
MPICC=/home/mpich2/bin/mpicc
FLAGS=-O2 -Wall

### Target
all: progseq progpar

progseq: progseq.o seq_functions.o
   $(CC) $(FLAGS) progseq.o seq_functions.o -o $@

progpar: progpar.o seq_functions.o par_functions.o
   $(MPICC) $(FLAGS) progpar.o seq_functions.o par_functions.o -o $@

par%.o : par%.c *.h Makefile
   $(MPICC) $(FLAGS) -c $< -o $@

%.o: %.c *.h Makefile
   $(CC) $(FLAGS) -c $< -o $@

clean:
   rm -f *.o progseq progpar
How to execute with ch_p4mpd

- `mpdboot -n <num> [-f <hostsfile>]`
  It creates the daemons ring on `<num>` machines among those listed in a file (the default one is `~/mpd.hosts`). It requires also a password file (the default one is `~/mpd.conf`).

```
user@cluster1:~$ cat mpd.hosts
cluster1.ge.imati.cnr.it
cluster2.ge.imati.cnr.it
cluster3.ge.imati.cnr.it
....
cluster16.ge.imati.cnr.it
```

- `mpdallexit`: it kill the existing daemons ring.

```
user@cluster1:~$ cat .mpd.conf
secretword=bidibibodibibibu
```

```
user@cluster1:~$ mpdringtest
time for 1 loops = 0.00268411636353 seconds
```
How to execute with ch_p4mpd

We need a daemon for each node, even with a SMP machine. The process ranks follow the daemons order, provided by `mpdtrace`:

```
user@cluster1:~$ cat mpd.hosts
cluster1  → rank 0
cluster3  → rank 1
cluster2  → rank 2
cluster5  → rank 3
....
```

If the number of processes is greater than the number of daemons, the processes are allocated on the machines in a round-robin way following this order.
How to execute

- `mpirun/mpiexec <args> <executable> [<exec-args>]`
  
  args: they are the arguments of mpirun
  
  -np <num>: how many parallel processes to start
  
  ...

```bash
user@cluster1:~$ mpiexec -np 8 ring

Ciao da 0 e da 1 2 3 4 5 6 7
L'ultimo processo e' in esecuzione su: cluster6
```

The `stdio` and `stderr` of all the processes is forwarded to the shell.
It is also possible to kill all the processes with CTRL+C.
Acknowledgements

- Rusty Lusk

- MPI
  http://www.mpi-forum.org/

- MPICH
  http://www.mcs.anl.gov/research/projects/mpich2/

- PVFS
  http://www.pvfs.org/documentation/