MPI 2 features

George Bosilca
Graham Fagg

Overview

» MPI-1 what it missed out
» And why...
» MPI-2 what it included
» Language issues
» Process Management
» Establishing Communication
» Single sided Communication
» Intercommunicator Collective Operations
» I/O including Parallel I/O (PIO)

What MPI-1 Missed out

» A lot of things that were hard to agree on for a standard
» Anything that might not run well on a early 90s late 80s MPP
» Remember who made the standard
» NEC, IBM, SGI, Intel, HP....
» And for which parallel machines ....

Basics of MPI-1

» All characteristics of message passing are contained within communicators
» Communicators contain:
  » Process lists or groups
  » Connection/communication structures topologies
  » System derived message tags/envelopes to separate messages from each other
» All processes in an MPI-1 application belong to a global communicator called MPI_COMM_WORLD
» All other communicators are derived from this global communicator.
» Communication can only occur within a communicator.
» Safe communication

MPI-1 Internals

Processes

» All process groups are derived from the membership of the MPI_COMM_WORLD communicator.
» I.e.: no external processes.
» MPI-1 process membership is static not dynamic like PVM or LAM 6.x
» simplified consistency reasoning
» fast communication (fixed addressing) even across complex topologies.
» interfaces well to simple run-time systems as found on many MPPs.

MPI-1 Application
Disadvantages of MPI-1

» Static process model
  » If a process fails, all communicators it belongs to become invalid. I.e. No fault tolerance.
  » Dynamic resources either cause applications to fail due to loss of nodes or make applications inefficient as they cannot take advantage of new nodes by starting/spawning additional processes.
  » When using a dedicated MPP MPI implementation you cannot usually use off-machine or even off-partition nodes.

» Multi-language operation is not supported
  » MPI specifies both ANSI C and F77 binding
  » No agreed standard for data type conversions between languages even upon the same architecture
  » Communicators cannot be passed between different language modules due to their representation
    » In F77 a communicator is an INTEGER
    » In C it is usually a pointer to a structure
  » non standard methods exist in different implementations

MPI-2

» Problem areas and needed additional features identified in MPI-1 are being addressed by the MPI-2 forum.
» These include:
  » inter-language operation
  » dynamic process control / management
  » parallel IO
  » extended collective operations
  » Support for inter-implementation communication/control is not being considered.
  » See the MetaComputing lectures for more...

MPI-2: Language Issues

» Think back to MPI-1
  » What is a communicator in ‘C’
  » What is a communicator in ‘Fortran’
  » Does it depend on the implementation ?
  » How could I write a program in both ‘C’ and ‘Fortran’ and have them pass communicators?

MPI-2: Language Issues

» MPI-2 Handle conversion
  » Fortran to C/C++
    » ‘C’ Wrappers convert Fortran handles
    » C/C++ to Fortran
      » Not supported
        » Why do you think not ??
    » C to C++
      » Overloading C++ operators called on the C++ side
    » C++ to C
      » Not supported
        » Want to write a new C++ compiler ??

» All wrappers of the form:
  » MPI_<C_CLASS>_t MPI_<CLASS>_f2c (MPI_int handle)
  » All handles in fortran are integers (INT*4)
  » Handles in C are implementation dependant (pointers or integers)
**MPI-2: Language Issues**

- C++ binding
  - C++ has types C does not such as
    - MPI::COMPLEX, MPI::BOOL etc
  - Uses a namespace called MPI that contains all object types such as:
    - Class comm; class Cartcomm, class Datatype,..
**MPI-2 Spawn Functions**

- MPI_COMM_SPAWN
  - Starts a set of new processes with the same command line
  - SPMD

- MPI_COMM_SPAWN_MULTIPLE
  - Starts a set of new processes with potentially different command lines
  - Different executables and / or different arguments
  - MPMD

**Spawn Semantics**

- Group of parents collectively call spawn
  - Launches a new set of children processes
  - Children processes become an MPI job
  - An inter-communicator is created between parents and children
  - Parents and children can then use MPI functions to pass messages

**Types of Communicators**

- *Intra*communicator
  - "Normal" communicator
  - MPI_COMM_WORLD is an intracommunicator
  - One group of processes

- *Inter*communicator
  - Two groups of processes: local and remote
  - Always communicate relative to remote group
  - Both types can be used with MPI_SEND / MPI_RECV

**Continue Previous Example**

- MPI_COMM_WORLD and one derived communicator
  - Both are intracomms
  - Create another derived communicator
  - Now have 2 groups
Dynamic Processes: Spawn

Parents call MPI_COMM_SPAWN

Two processes are launched

Children processes call MPI_INIT

Children create their own MPI_COMM_WORLD
An intercommunicator is formed between parents and children.

Children call MPI_COMM_GET_PARENT to get intercommunicator.

MPI-2: Process Management

- The creating processes get an intercommunicator to the child group as soon as the spawn call returns.
- They can send to it before the children have called MPI_Init().

- Is the Intercommunicator functionality enough to allow for a full dynamic process model with efficient communications?
  - Hints
    - What are the limits of an intercommunicator
    - How are multiple child groups
MPI-2: Process Management

- Can we do this??

MPI-2: Process Management

» Can the two children groups create a direct connection?
» using an MPI communicator create / merge call of some kind?

MPI-2: Process Management

- We do not have any overlapping communicators between the three groups...

MPI-2: Process Management

- Can not do collectives directly

Dynamic Processes:
Connect / Accept
Establishing Communications

» MPI-2 has a TCP socket style abstraction
  » Process can accept and connect connections from other processes
  » Client-server interface
  » MPI_COMM_CONNECT
  » MPI_COMM_ACCEPT

Establishing Communications

» How does the client find the server?
  » With TCP sockets, use IP address and port
  » What to use with MPI?
  » Use the MPI name service
  » Server opens an MPI "port"
  » Server assigns a public "name" to that port
  » Client looks up the public name
  » Client gets port from the public name
  » Client connects to the port

Server Side

» Open and close a port
  » MPI_OPEN_PORT(info, port_name)
  » MPI_CLOSE_PORT(port_name)
  » Publish the port name
  » MPI_PUBLISH_NAME(service_name, info, port_name)
  » MPI_UNPUBLISH_NAME(service_name, info, port_name)

Client Side

» Lookup port name
  » MPI_LOOKUP_NAME(service_name, info, port_name)
  » Connect to the port
  » MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
    » comm is a intracommunicator; local group
    » newcomm is an intercommunicator; both groups

Server Side

» Accept an incoming connection
  » MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
    » comm is a intracommunicator; local group
    » newcomm is an intercommunicator; both groups

Connect / Accept Example
Server calls MPI_OPEN_PORT

Server calls MPI_PUBLISH_NAME("ocean", info, port_name)

Server blocks in MPI_COMM_ACCEPT("Port A", ...)

Client calls MPI_LOOKUP_NAME("ocean", ...), gets "Port A"

Client calls MPI_COMM_CONNECT("Port A", ...)

Intercommunicator formed; returned to both sides
**Server calls MPI_UNPUBLISH_NAME(“ocean”, ...)**

**Connect / Accept Example**

**Server calls MPI_CLOSE_PORT**

**Connect / Accept Example**

**Both sides call MPI_COMM_DISCONNECT**

**Connect / Accept Example**

**Peer-to-Peer Demonstration**

- Start two independent MPI jobs via two separate mpirun commands
- One application publishes its port name
- The other looks up the port name
- They connect via MPI_COMM_CONNECT and MPI_COMM_ACCEPT
- The processes exchange data, disconnect, and shut down

**Summary**

- Server opens a port
- Server publishes public “name”
- Client looks up public name
- Client connects to port
- Server unpublishes name
- Server closes port
- Both sides disconnect
- Similar to TCP sockets / DNS lookups

**MPI_COMM_JOIN**

- A third way to connect MPI processes
  - User provides a socket between two MPI processes
  - MPI creates an intercommunicator between the two processes

→ Will not be covered in detail here
Collective Operations

- Collective operations are defined on both intra- and intercommunicators
- Hence, can use collectives on the communicators returned by SPAWN, ACCEPT, CONNECT
- However -- beware!
  - Intracommunicator collectives are “familiar”
  - Intercommunicator collectives are different
  - Read the MPI-2 chapter on “Extended Collectives”

Disconnecting

- Once communication is no longer required
  - MPI_COMM_DISCONNECT
  - Waits for all pending communication to complete
  - Then formally disconnects groups of processes -- no longer “connected”
  - Cannot disconnect MPI_COMM_WORLD

Single Sided communications

Put / Get / Accumulate

MPI-2: Single sided communications

- Normal message passing operation needs at least two parties
  - A sender who performs a send call
  - A receiver who performs a receive call

Why is this? And what does it have to do with... memory management / protection?

MPI-2: Single sided communications

- Earlier Cray MPP systems allowed processes to remotely access other processes memory via shmget and shmput system function calls.
  - This is known as Remote Memory Access (RMA)
MPI-2: Single sided communications

- Data (memory) in a fixed range (a window) is made available with a MPI_Win_create ( ) call.
- Freed with MPI_Win_free ( )
- Data can then be accessed via
  - MPI_Put ( )
  - MPI_Get ( )
  - MPI_Accumulate ( )

MPI-2: Single sided communications

- The communication calls (put/get/accumulate) are non-blocking
- The operation occurs sometime after the call BUT before a synchronization point

MPI-2: Single sided communications

- RMA communication is in two classes
  - Active
    - Memory is moved from one process to another
    - One process calls the move
    - Both must call the synchronization (including the owner of the target memory)
  - Like message passing

MPI-2: Single sided communications

- RMA communication is in two classes
  - Passive
    - Memory is copied from a target to two other processes
    - Both processes call the copy
    - Both must synchronize (complete) their move, expect the target does not need to synchronize
  - Like shared memory

MPI-2: Single sided communications

- Semantics of an active target operation cycle are:
  - Sync on a win (RMA target window)
  - Start of an epoch
    - Perform zero or more transfer operations
    - Such as put/get/accumulate
  - End of an epoch
    - Sync on a win
  - Passive target operations require no sync operations

MPI-2: Single sided communications

- MPI_Win_fence ( )
  - A collective that can be used across the whole application to control operations so that they occur at the correct time such as exchanging data at the end of a time step
MPI-2: Single sided communications

» MPI_Win_fence ( ) is a collective that allows groups to sync their memory operations.

» Think of a set of MPI_Isends/MPI_Irecvs followed by a MPI_Waitall and a MPI_Barrier

MPI-2: Intercommunicator collectives

» Collective operations require an intracommunicator
» Sometimes it would be useful to perform a collective (such as a broadcast) across an intracommunicator
» As in the case of when we cannot create an intracommunicator
MPI-2: Intercommunicator collectives

- Semantics of these (MPIX*) calls
  - We need to know which is the root group and which is the leaf (target) group
  - How do we know the root group?
    - All processes in that group other than the real root call the operation with MPI_PROC_NULL as their root
    - The root calls the operation with the root set to a special constant MPI_ROOT
  - MPIX was a project at MSU that first proposed these extensions to MPI (hence MPIX)

MPI Profiling

- Tool writers like to tell you what is happening when running an MPI application so you can improve things
  - Three ways of doing this
    - Instrument the MPI library
      - Common way of doing it
    - Instrument your application before it links to the MPI library
      - Useful for very complex debuggers
    - Catch the runtime calls
      - Pardym (do it without the source available)

MPI-1: Profiling

- How it works
  - All MPI calls have a non-profiled version called:
    - PMPI_<CALL>
  - The standard MPI_<CALL> links to this PMPI_<CALL> unless a replacement MPI_<CALL> is provided.
**MPI-1: Profiling**

» Mylib has MPI_Send() implemented

![Diagram](image1)

**MPI Application**

* MPI_Comm_rank()
* MPI_Send()

* PMPI_Comm_rank

* Real MPI lib

**MPI Application**

* MPI_Send()

* PMPI_Comm_rank

* PMPI_Send

* Real MPI lib

**MPI-1: Profiling**

» The profile library does have full access to the real MPI library, by calling the PMPI_<CALL> calls.

» The tool writer only needs to implement the calls they require
  * Rather than 248+

**MPI-1: Profiling**

» An application can pass information to the profile library
  * MPI_Pcontrol (level)
    * Indicated to the profile library what level of debugging you might want
    * Can also use the MPI key attributes as well.

**MPI-1: Profiling**

» MPICH comes with multiple profile libraries
  * We are interested in the mpilog library
  * Mpcc myfile.c -o myfile -mpilog -Impe ...

**MPI-1: Profiling**

» When executed will produce a
  * <exename>.clog file
  * This needs to be converted to an slog (scalable log file) before viewing etc
  * /usr/local/mpich-1.2.1/clog2slog myfile.clog
MPI-1: Profiling

- Then run the logview on it
  - /usr/local/mpich-1.2.1/bin/logviewer myfile.slog
  - Choose the <MPI Process> button and hit <display>

MPI-1: Profiling

- That was a barrier followed by a set of broadcasts so that they could be benchmarked.
- You will need to show that you can use the logfile viewer for your homework exercises