

MPI 2 features

George Bosilca <bosilca@cs.utk.edu>
Graham Fagg <fagg@cs.utk.edu>

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 IO (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

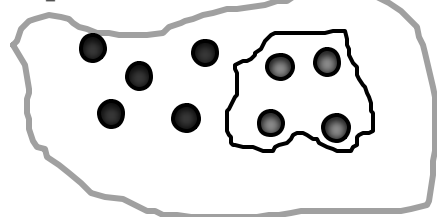
- » **All** characteristics of message passing are contained within communicators
- » Communicators contain:
 - » Process lists or groups
 - » Connection/communication structures **topologies**
 - » System derived message tags/envelopes to separate messages from each other
- » All processes in an MPI-1 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

MPI_COMM_WORLD Derived_Communicator



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.

Disadvantages of MPI-1

- » 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 I/O
 - » 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 ??

MPI-2: Language Issues

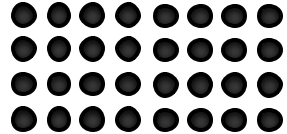
- » All wrappers of the form:
 - » `MPI_<C_CLASS> MPI_<CLASS>_f2c` (MPI_Fint handle)
 - » All handles in fortran are integers (INT*4)
 - » Handles in C are implementation dependant (pointers or integers)

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..

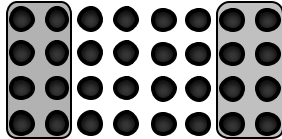
MPI-2: Process Management

- » In MPI-1
 - » You get a set of process after you call MPI_Init ()
 - » You keep them until you call MPI_Finalize
 - » Or you halt..



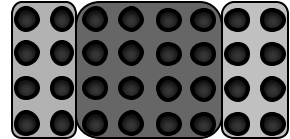
MPI-2: Process Management

- » In MPI-1
 - » You get a set of process after you call MPI_Init ()
 - » You keep them until you call MPI_Finalize
 - » Or you halt..



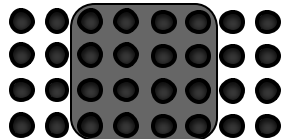
MPI-2: Process Management

- » In MPI-1
 - » You get a set of process after you call MPI_Init ()
 - » You keep them until you call MPI_Finalize
 - » Or you halt..



MPI-2: Process Management

- » In MPI-1
 - » You get a set of process after you call MPI_Init ()
 - » You keep them until you call MPI_Finalize
 - » Or you halt..



MPI-2: Process Management

- » How do you add more nodes to an already running MPI application ??
- » How can you drop some of the nodes ??
- » How could we couple two or more applications ??
- » How would we handle a node failure ??

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

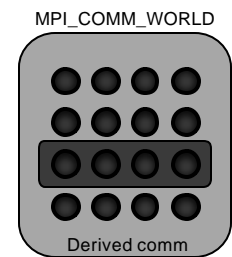
- » Group of parents collectively call spawn
 - » Launches a new set of children processes
 - » Children processes become an MPI job
 - » An **inter**-communicator is created between parents and children
- » Parents and children can then use MPI functions to pass messages

Types of Communicators

- » **Intracommunicator**
 - » "Normal" communicator
 - » MPI_COMM_WORLD is an intracommunicator
 - » One group of processes
- » **Intercommunicator**
 - » Two groups of processes: local and remote
 - » Always communicate relative to remote group
- » Both types can be used with MPI_SEND / MPI_RECV

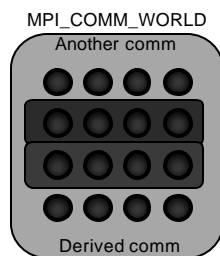
Continue Previous Example

- » MPI_COMM_WORLD and one derived communicator
- » Both are intracomms



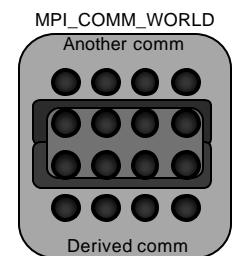
Continue Previous Example

- » MPI_COMM_WORLD and one derived communicator
- » Both are intracomms
- » Create another derived communicator
- » Now have 2 groups



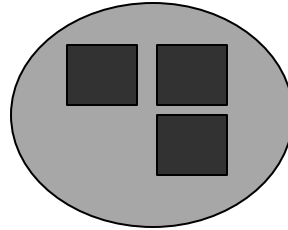
Continue Previous Example

- » MPI_COMM_WORLD and one derived communicator
- » Both are intracomms
- » Create another derived communicator
- » Now have 2 groups
- » Create intercomm from the two groups

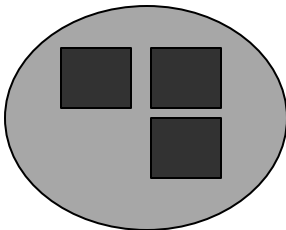


**Dynamic Processes:
Spawn**

Spawn Example

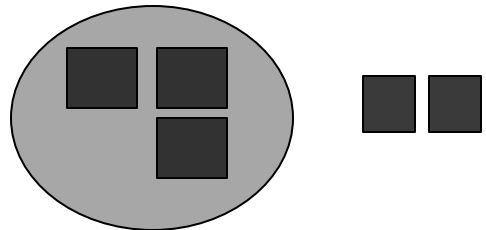


Spawn Example



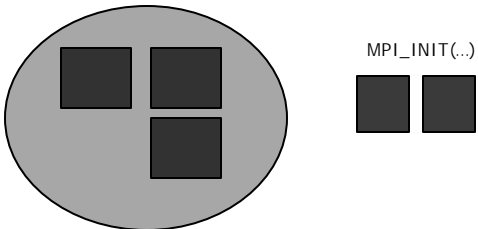
Parents call `MPI_COMM_SPAWN`

Spawn Example



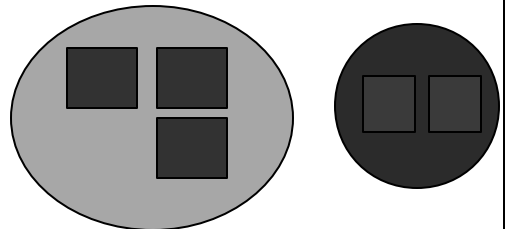
Two processes are launched

Spawn Example



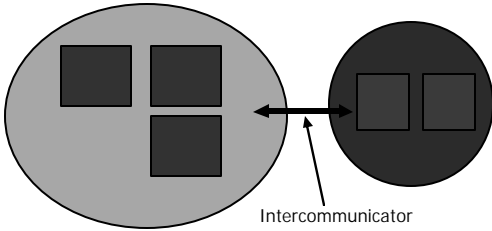
Children processes call `MPI_INIT`

Spawn Example



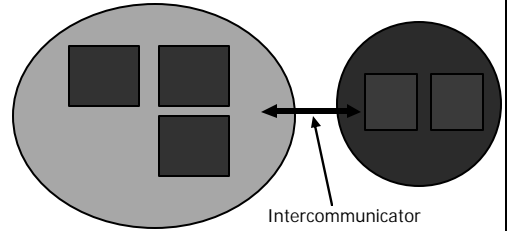
Children create their own `MPI_COMM_WORLD`

Spawn Example



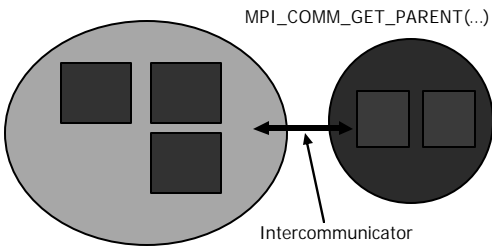
An intercommunicator is formed between parents and children

Spawn Example



Intercommunicator is returned from MPI_COMM_SPAWN

Spawn Example



Children call MPI_COMM_GET_PARENT to get intercommunicator

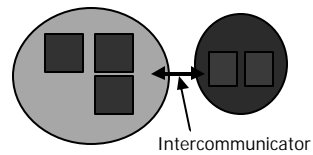
MPI-2: Process Management

- » 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 ()

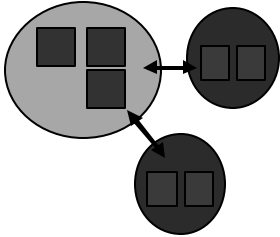
MPI-2: Process Management

- » 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

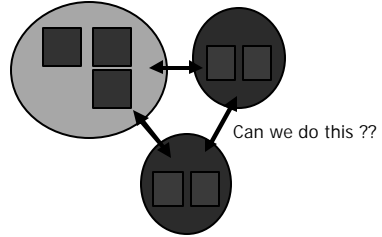
MPI-2: Process Management



MPI-2: Process Management



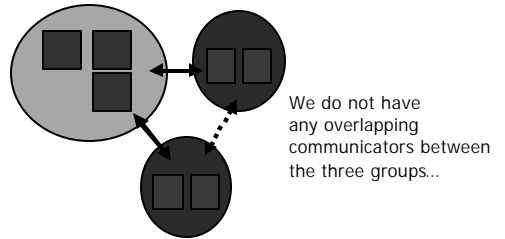
MPI-2: Process Management



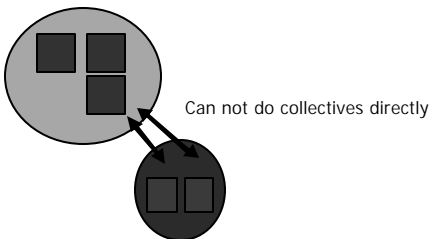
MPI-2: Process Management

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

MPI-2: Process Management



MPI-2: Process Management



**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)

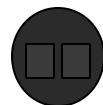
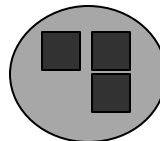
Server Side

- » Accept an incoming connection
 - » MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
 - » comm is a **intra**communicator; local group
 - » newcomm is an **inter**communicator; both groups

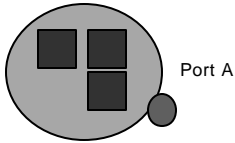
Client Side

- » 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 **intra**communicator; local group
 - » newcomm is an **inter**communicator; both groups

Connect / Accept Example

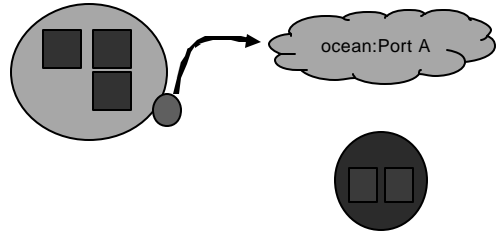


Connect / Accept Example



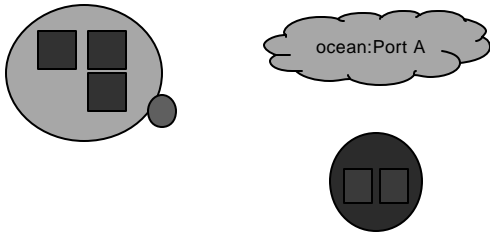
Server calls `MPI_OPEN_PORT`

Connect / Accept Example



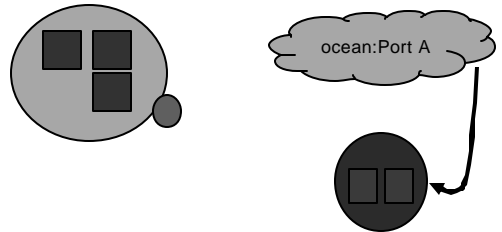
Server calls `MPI_PUBLISH_NAME("ocean", info, port_name)`

Connect / Accept Example



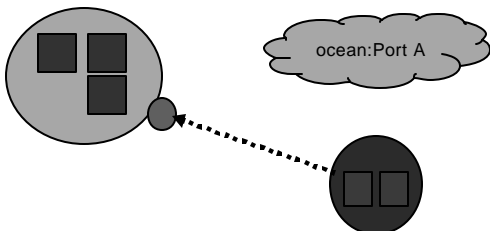
Server blocks in `MPI_COMM_ACCEPT("Port A", ...)`

Connect / Accept Example



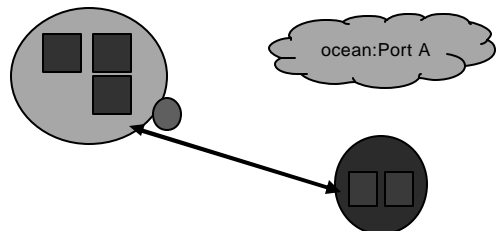
Client calls `MPI_LOOKUP_NAME("ocean", ...)`, gets "Port A"

Connect / Accept Example



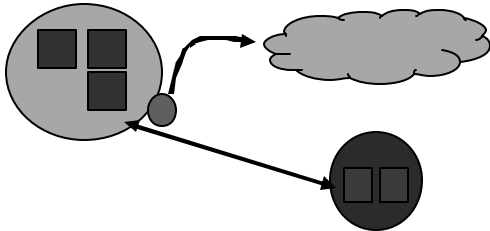
Client calls `MPI_COMM_CONNECT("Port A", ...)`

Connect / Accept Example



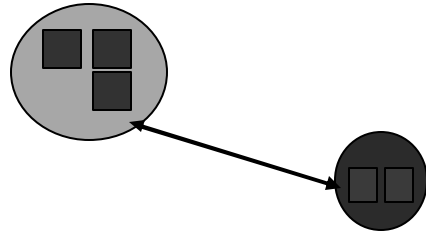
Intercommunicator formed; returned to both sides

Connect / Accept Example



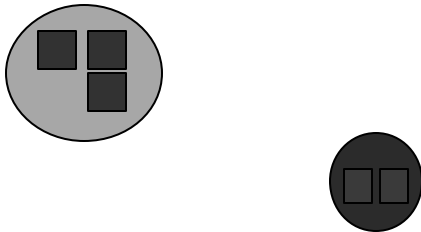
Server calls `MPI_UNPUBLISH_NAME("ocean", ...)`

Connect / Accept Example



Server calls `MPI_CLOSE_PORT`

Connect / Accept Example



Both sides call `MPI_COMM_DISCONNECT`

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

- » 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 **inter**communicators
 - » Hence, can use collectives on the communicators returned by SPAWN, ACCEPT, CONNECT
- » However -- **beware!**
 - » **Intra**communicator collectives are "familiar"
 - » **Inter**communicator 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

- » Remote Memory Access (RMA)
 - » Is fast
 - » Can allow for simple program design
 - » An operation specifies all the send and receive arguments together

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

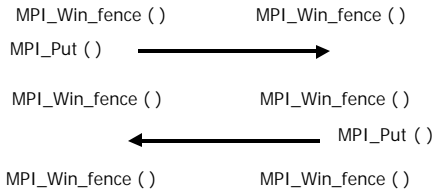
MPI-2: Single sided communications

- » 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

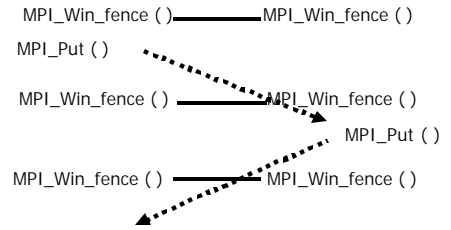
MPI-2: Single sided communications

- » `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-2: Single sided communications



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_Irecv followed by a MPI_Waitall and a MPI_Barrier

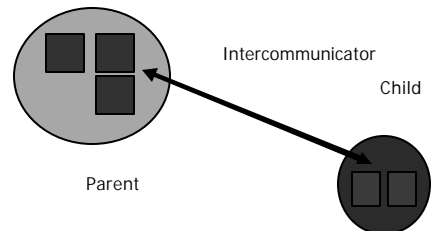
MPI-2: Single sided communications

- » For fine grain control between pairs of processes only there is a more complex interface
 - » MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait, MPI_Win_lock, MPI_Win_unlock
- » Close to some of the Posix threads interface

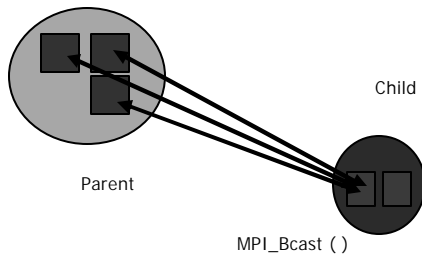
MPI-2: Intercommunicator collectives

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

MPI-2: Intercommunicator collectives



MPI-2: Intercommunicator collectives



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

MPI-1: 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
 - » Pardy (do it without the source available)

MPI-1: Profiling

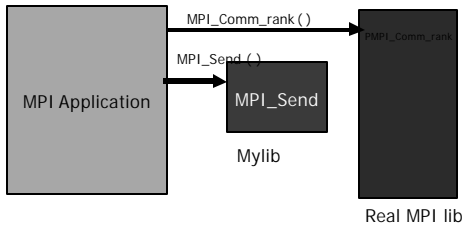
- » MPI-1 provides a built in method of doing this
 - » Known as the profiling interface
 - » Allows instrumentation or replacing of only the required calls at run time
 - » Relies on weak linking in most implementations

MPI-1: Profiling

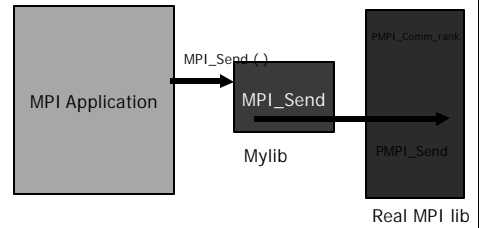
- » How it works
 - » All MPI calls have a non-profiled version called:
 - » PMPI_<CALL>
 - » The standard MPI_<CALL> links to this PMPI_<CALL> unless a replacement MPI_<CALL> is provided.

MPI-1: Profiling

- » Mylib has MPI_Send () implemented



MPI-1: Profiling

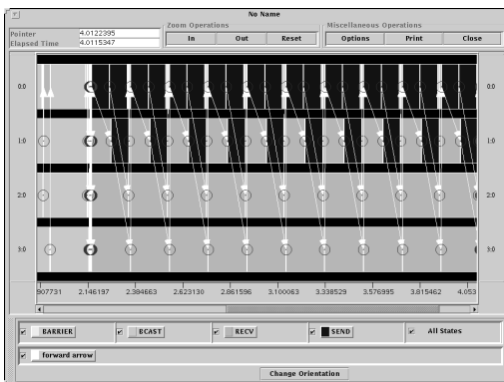


MPI-1: Profiling

- » 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+

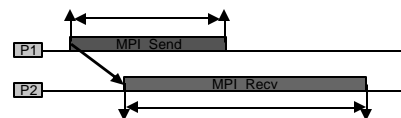
MPI-1: Profiling

- » 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.
 - » There are several log storage format (clog, slog, epilog) as well as tools to gather and interpret the logs (KOJAK, ...)



PMPI

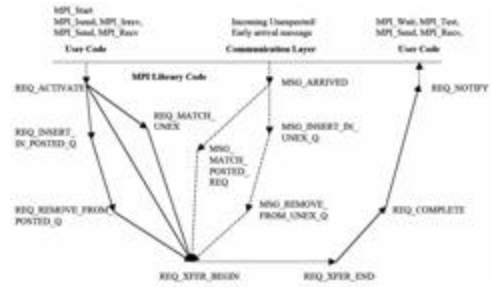
- » High level profiling interface
- » Allow detection of starting and ending events for all the MPI functions.
- » Most of the time this information is not enough to understand the behaviour of the MPI application



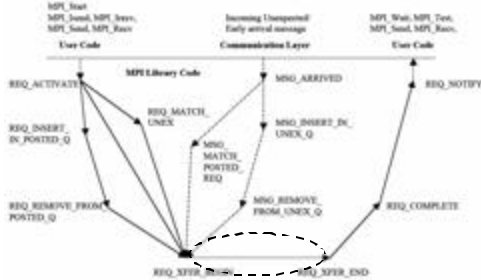
Peruse

- » An MPI extension for revealing unexposed implementation information
- » Similar to PAPI (who expose processor counters)
- » A set of events tracing all the lifetime of an MPI request
- » Extension for file access and collective communications

Peruse



Peruse



MPI Thread support

MPI Thread

- » Several level of thread support
- » MPI_Init_thread(int* argc, char*** argv, int required, int* provided)
 - » MPI_THREAD_SINGLE - no thread support
 - » MPI_THREAD_FUNNELED - the process can have threads but only the main thread can call MPI functions
 - » MPI_THREAD_SERIALIZED - the process can be multi-threaded, all threads can call MPI functions but only one at the time. The synchronization is managed by the user.
 - » MPI_THREAD_MULTIPLE - all threads can call MPI functions without restrictions.

MPI Thread

- » MPI_Query_thread : return the current level of thread support
- » MPI_Is_thread_main: return true if the current thread is the one who called MPI_Init. Useful for the case when the provided thread support is MPI_THREAD_FUNNELED.

POSIX Threads

Process vs. Thread

- » A process is a collection of virtual memory space, code, data, and system resources.
- » A thread (lightweight process) is a slice of executable code that is to be serially executed within a process.
- » A process can have several threads. They all share the same memory space at the user level, and most of the information stored at the kernel level.

Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process's global variables and resources.

Possible to create more efficient applications ?

Process vs. Thread

- » Multithreaded applications must avoid two threading problems: deadlocks and races.
- » A deadlock occurs when each thread is waiting for the other to do something.
- » A race condition occurs when one thread finishes before another on which it depends, causing the former to use a bogus value because the latter has not yet supplied a valid one.

The key is synchronization

- » Synchronization = gaining access to a shared resource.
- » Synchronization REQUIRE cooperation.

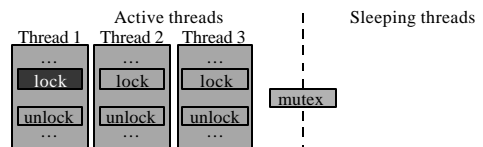
POSIX Thread

- » What's POSIX ?
 - » Widely used UNIX specification
 - » Most of the UNIX flavor operating systems
 - » Some degree of compatibility is available on Windows OS.

POSIX is the Portable Operating System Interface, the open operating interface standard accepted world-wide. It is produced by IEEE and recognized by ISO and ANSI.

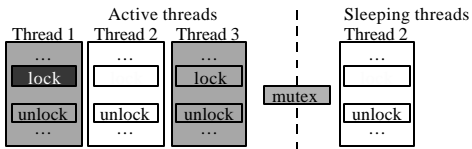
Mutual exclusion

- » Simple lock primitive with 2 states: lock and unlock
- » Only one thread can lock the mutex.
- » Several politics: FIFO, random, recursive



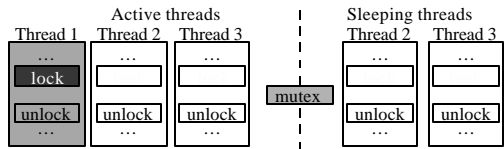
Mutual exclusion

- » Simple lock primitive with 2 states: lock and unlock
- » Only one thread can lock the mutex.
- » Several politics: FIFO, random, recursive



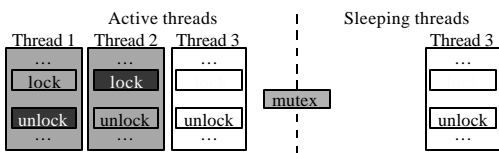
Mutual exclusion

- » Simple lock primitive with 2 states: lock and unlock
- » Only one thread can lock the mutex.
- » Several politics: FIFO, random, recursive



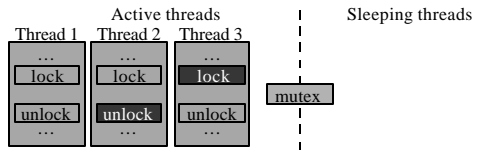
Mutual exclusion

- » Simple lock primitive with 2 states: lock and unlock
- » Only one thread can lock the mutex.
- » Several politics: FIFO, random, recursive



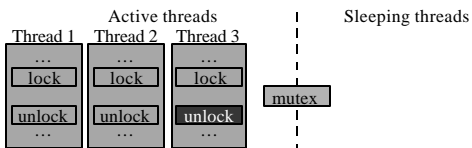
Mutual exclusion

- » Simple lock primitive with 2 states: lock and unlock
- » Only one thread can lock the mutex.
- » Several politics: FIFO, random, recursive



Mutual exclusion

- » Simple lock primitive with 2 states: lock and unlock
- » Only one thread can lock the mutex.
- » Several politics: FIFO, random, recursive



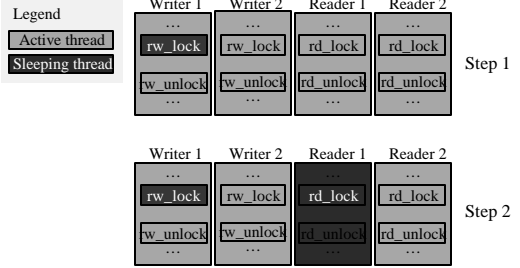
Mutual exclusion

- » Spin vs. sleep ?
- » What's the desired lock grain ?
 - » Fine grain – spin mutex
 - » Coarse grain – sleep mutex
- » Spin mutex: use CPU cycles and increase the memory bandwidth, but when the mutex is unlock the thread continue his execution immediately.

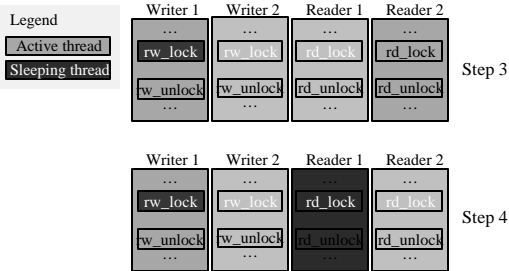
Shared/Exclusive Locks

- » ReadWrite Mutual exclusion
- » Extension used by the reader/writer model
- » 4 states: write_lock, write_unlock, read_lock and read_unlock.
- » multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock.
- » if one thread holds an exclusive lock, no threads may hold a shared lock.

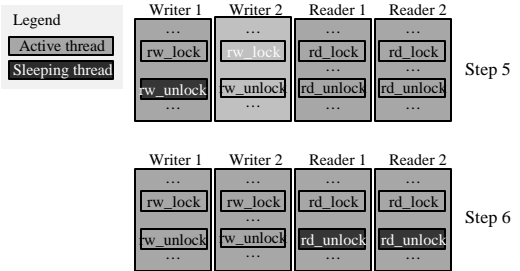
Shared/Exclusive Locks



Shared/Exclusive Locks

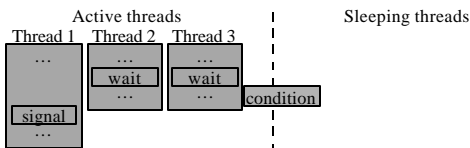


Shared/Exclusive Locks



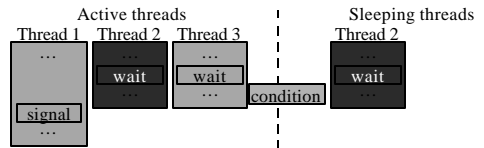
Condition Variable

- » Block a thread while waiting for a condition
- » Condition_wait / condition_signal
- » Several thread can wait for the same condition, they all get the signal



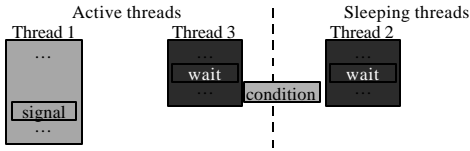
Condition Variable

- » Block a thread while waiting for a condition
- » Condition_wait / condition_signal
- » Several thread can wait for the same condition, they all get the signal



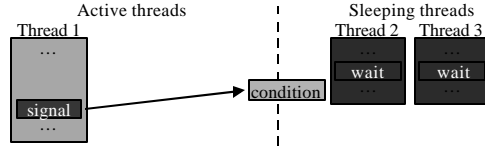
Condition Variable

- » Block a thread while waiting for a condition
- » Condition_wait / condition_signal
- » Several thread can wait for the same condition, they all get the signal



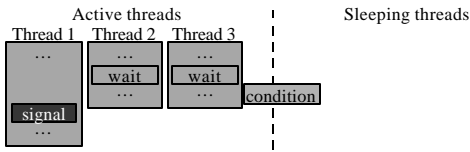
Condition Variable

- » Block a thread while waiting for a condition
- » Condition_wait / condition_signal
- » Several thread can wait for the same condition, they all get the signal



Condition Variable

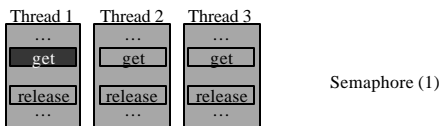
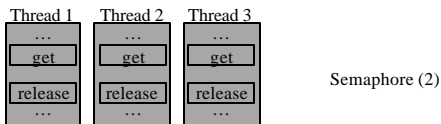
- » Block a thread while waiting for a condition
- » Condition_wait / condition_signal
- » Several thread can wait for the same condition, they all get the signal



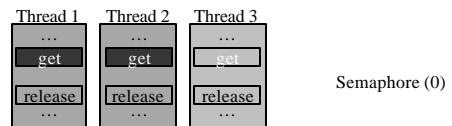
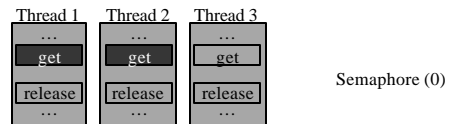
Semaphores

- » simple counting mutexes
- » The semaphore can be hold by as many threads as the initial value of the semaphore.
- » When a thread get the semaphore it decrease the internal value by 1.
- » When a thread release the semaphore it increase the internal value by 1.

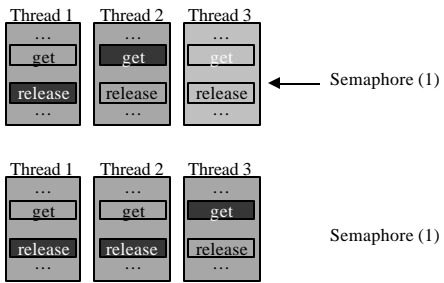
Semaphores



Semaphores



Semaphores



Atomic instruction

- » Is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and the operation is safe from interference by other CPUs
 - » TestAndSet
 - » CompareAndSwap
 - » DoubleCompareAndSwap
 - » Atomic increment
 - » Atomic decrement