Overview

History of message passing
Parallel Computing and MPI
MPI-1 & MPI-2

Message Passing

What is message passing?
Its where tasks or processes communicate via explicit send and receive operations on fixed items of data...

As opposed to?
Shared memory where normal read/write operations can be used to SHARE data.

Message Passing

Why message pass?
So we can break problems into smaller pieces for faster performance (DMMP / MIMD)
For fault tolerance (multiple servers)
Explicit operations can be reasoned about in a formal way, i.e. CSP
Message Passing can be standardized to allow for highly portable applications
Was not always true
Is now happening to shared memory, see OpenMP

Message Passing

Data can be passed between processes on the same machine or between processes on different machines.
They might not even exist at the same time
I.e. no need for temporal coupling but this is unusual
Message Passing

Many computer systems cannot share resources in a transparent way. Even we can, we can implement message passing on top of that shared resource. This sometimes yield great performance increases. Message passing allows systems that avoid contention on resources. Compare the IBM SP-2 to the SGI Origin2000.

Message Passing

How long has it been around? Since before networks! Copy data (known as a message onto a removable media, such a a removable disk) Pass the message (i.e move the disk across the room/building/country) Read the message (mount the disk and read it)

Not that silly an example, until recently some universities in NYcity passed data via motorbike couriers on Exabyte tapes. Until ATM has finally caught up in Bandwidth.

Basics

We need two or more entities
A sender
A receiver or receivers
Some data, the message
Some means of passing the data
network, shared resource...
The two basic operations
Send and Receive
And a means of identifying each entity, and the message... maybe

Basics

The Sender passed the message by:

sending it:

send (data, somewhere, args)

The receiver gets it by asking for it:

receiving the message:

receive (some buffer, some other args)

The variations will be examined in a few slide time.

The history of message passing as we know of it

ARPA net users passed message via protocols such as IP
E-mail is a message passing system..

But, we are interested in the ‘true’ parallel computing versions..

Past message passing systems

How the facilities and functionality of modern message passing systems evolved has been influenced by a vast number of research projects and commercial implementations by vendors of MPP machines.

We will briefly discuss some of these systems in terms of which features (we now think of as standards) that they introduced and what they omitted. This will also help you understand how implementers of such systems have learned to insulate users more effectively from the increasing complexity of the underlying systems.

I C L
Vendor Message Passing Systems and Machines

First we will cover the hardware systems and their message passing systems that led to today's range of systems.

Then we will cover some portable message passing systems.

Not an exhaustive list, but long enough to show just how varied it can be.

Caltech Hypercube

The Caltech Hypercube (circa 1984) was a d-dimensional hypercube structured system with a computational node at the end of each vertex, and a single host to control the machine (known as an Intermediate Host).

The system was programmed in either C or Fortran77 and communication was based on a subroutine library known as the Crystaline Operating System (CROS).

The communications library assigned addresses to tasks depending on which node they were physically located, processes could only communicate to neighbors or the intermediate host (a total of d+1 links). The CROS terminology for a link between two nodes was that of a channel through which 8 byte message packets could be sent.

Caltech Hypercube

The system only supported collective operations (broadcast) to/from the intermediate host and the overall communication pattern was SIMD in nature.

I.e. all processes had to do the same communication routine at the same time. This lead to a single hardware path between the worker SMP nodes such as the CDC Cyber and Thinking Machines CM-2 and the later Intel iPSC and IBM Paragon machines. The former where program execution and communication occurs fully in lockstep and the latter where ordering was completely independent.

On the Hypercube under CROS, the program was free to run independently of each other but the hardware forced all the communication into lockstep.

For solving very regular problems in physics such as partial derivatives for large numbers of grid points, the structure imposed by the programming environment was an aid for producing hardware very efficient implementations.

Caltech Hypercube

Went from a fixed broadcast (i.e. no addresses specified in the send operation) to an addressed based system.

Making everybody do a send and then receive the same time... not nice.

Note the small message size of 8 bytes.

If you wanted more, you had to write your own message passing layer... nasty.

Meiko CS-1 and Occam

Transputer based multiprocessor

The transputer was a 32 bit microprocessor that had communication hardware built in.

4 high speed links

The transputer could context switch in a single cycle

I.e. multitasking-multithreading very quickly

more than one process per processor

Meiko CS-1 and Occam

Occam was a language based on the CSP specification language

CSP - Communicating Sequential Processes

CSP could be formally reasoned about

Popular target for S/W Eng projects in the early days of ESPRIT.

Many CS-1 machines were supplied under the ESPRIT ALPHA Project

Occam was a parallel language
Occam

Instructions were executed in order of blocks
blocks could be executed internally as parallel (par), sequential (seq) or
alternately (alt) which meant non-deterministically (but only one)

Seq
   do A
   do B
A happens before B always

Par
   do A
   do B
I.e. they execute together or after one another

alt
   do A
   do B
We don’t know which one executes in which order.

Occam and message passing

Message passing was built in
Based on channels between processes
   a channel is a bi-direction pipe much like unix sockets

Sending a message
   chan1 ! Data

Receiving a message
   chan1 ? buffer

Occam and message passing

Message passing was synchronous and blocking
I.e. both parties had to work together to message pass, and the send
and receiver would wait until their operation had completed completely
before the program would continue

Was easy for programmers to write code that deadlocked
lucky we had lots of ESPRIT analysis tools to help

Occam and message passing

Deadlocking code
Proc1
   chan1 ! Data
   chan1 ? Buffer
Proc 2
   chan1 ! Data
   chan1 ? Buffer

This problem happens in many systems without the MP system buffering for you.

Occam lives on

The Occam project still lives on where formally proved code is needed.
Newer versions allow recursion which was only possible by loop contracts previously
(see Occam 2 1/2)

Compilers are available for systems other than transputers such as Sparc, PowerPC etc
Kent retargetable Occam compiler project (Welsh96) from the University of Kent
at Canterbury.
SPOC - The Southampton Portable Occam Compiler from ECS, University of
Southampton.
NX and the Intel iPSC1 and the Intel Paragon

The original iPSC1 (circa 86) was a seven-dimensional hypercube structured system much like the Caltech Hypercube in terms of hardware design although it software environment known as NX1 was less like CROS and more like the "Distributed Processes environment.

NX1 The Intel Personal Super Computer for those who could afford a personal supercomputer... The comp.parallel Usenet news page was originally aimed at iPSC users. One of the largest Paragons XPS machines ever installed is at ORNL. The system was very well balanced, i.e. its computational power was proportional to its message passing performance.

Far quicker than even more modern systems like IBM SP2s (like the new ASCI Blue Pacific machine) or even many Cray T3D/E machines.

NX and the Intel iPSC1 and the Intel Paragon

This additional flexibility also added an increase in the complexity of the message passing library. Users now needed additional routines to inquire location information, and messages needed to be addressed correctly.

Additional arguments in the subroutine calls.
Messages needed to be identified on an individual basis as there was neither a fixed order of transmission and therefore receive, but the pattern of communications was no longer dictated by topology.

To assist developers, messages could be tagged or typed. Like a subject like in E-mail!

Unfortunately the original system did not permit the filtering of messages by sender identity and type at the same time, although the type could be set to the senders ID to allow for receive from sender semantics.

An additional new feature was that different length messages could be received prior to the receiving process knowing which was which and hence knowing how big a buffer memory to allocate, or even how much data was received.

If the user offered a buffer that was too small the the excess message data was discarded (truncated).

NX and the Intel iPSC1 and the Intel Paragon

The later Intel machines implemented an improved version of NX, known as NX2. In the case of the Paragon, this was implemented upon an OSF/1 micro Unix kernel.

A number of improvements were added such as interrupt driven communication which allowed an application to perform computation and be woken up when a message arrived instead of having to poll for them intermittently (leading to decreased cache performance and possible page faults as the OS calls are invoked to check for messages).

I.e. Asynchronous and non-blocking messaging

Be careful many users call non-blocking asynchronous and vice versa.

NX and the Intel iPSC1 and the Intel Paragon

The NX operating system was based on the Caltech Reactive Kernel which provided hiding of the underlying communication topology (processes where identified by a simple integer from 0 to P-1, where P was the number of processes per partition), multiple processes per node, and any to any message passing, non-synchronous messaging.

i.e. both sender and receiver do not have to be active at the same time for communication to complete) and non-blocking (i.e. no need to wait for completion)

what is now thought of as a typical set of features that define a message passing environment.
Up to this point communication had been point to point, i.e. from a single sender to a single receiver. New broadcast functions allowed the construction of global operations such as global synchronizations (barriers) and some arithmetic reduction operations.

Although NX's design inspired many of the features of current message passing systems, it also had a number of shortcomings:

- lack of more comprehensive group communication functions (to assist certain types of calculations)
- lack of message identification and filtering at the receivers end
- only one type compared to up to three used on later systems, and
- initially a high software overhead compared to simpler protocols such as active messages which had direct access to hardware.

But, it did give us the semantics of the most common message passing systems in current use.

Point to point communication under EUI was performed by sending messages to tasks directly in the same style as on the Intel iPSC systems addresses from 0 to N-1, where N was the number of tasks making up a parallel job.

The point to point system supported typed messages for both blocking and non-blocking messages.

Unlike NX, messages could be selected by the receiver on both message type and source (sender) including the use of wildcards.

To assist in handling non-blocking messages the user could check the status of a particular transfer as well as wait for completion of a named operation, any of a range of operations or all pending outstanding transfers.

EUI allowed the construction of conceptual collections of tasks into logical groups that could be addressed by a single group ID.

See PVM groups latter.

This allowed users to avoid having to list (sometimes large numbers of) tasks explicitly when passing messages in structured ways repeatedly.

The use of collective operations on these groups avoided the need for many point to point calls and allowed the system to perform these as efficiently as possible on the given hardware.

The range of operations included, barriers, data shifts, broadcasts, gather, scatter, a generalized combine and an associative reduction.

All the collective operations required all members of each group to partake in a blocking fashion.

The term 'collective' used in MPI comes from the IBM system.

Asynchronous returns from non-blocking functions required a two part status lookup. If the user interface to a non-blocking receive was as follows:

```c
void recvf (&data, sizeofbuffer, &size-received)
```

The size-received variable could not be filled in by the system until the non-blocking operation had completed, which might be while the thread that made this call was in a different program module. Thus the need to get a status handle which could be queried after the operation had completed and who's memory storage was handled by the messaging system.

```c
zero (data, ... setstatus)
```

wait (for above zero to complete)

`status` is now safe to examine, as the wait operation has completed any status data structures.
IBM Scalable Power series and EUI

The EUI project was not the first to introduce this two step strategy, this also occurred between NX1 and NX2, but the overall design used by IBM was the bases of that used by the MPI forum.

Also much of the EUI collective operations design also directly effected the design of MPI.

Thinking Machines CM5 and the CMMD Active Message Layer (AML)

The CMMD point to point library was built on top of the AML, and provided the common set of operations including:

- blocking, non-blocking, synchronous and asynchronous point to point operations with selection upon source, message type or wild cards.
- Interestingly the blocking calls were quicker than the non-blocking calls as they avoided system level copying of message data.

Another interesting point was the inclusion of a "send and then receive" operation.

This allowed for simpler coding of stencil operations and boundary exchanges in domain decomposition problems.

The system implementation of the joint send and receive operation being quicker than the two separate operations.

The send-receive operation was so useful that it was also provided for in MPI.

For example previously to exchange values two operations would be required:

<table>
<thead>
<tr>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (B, data)</td>
<td>make copy of data edge into data'</td>
</tr>
<tr>
<td>Recv (B, data)</td>
<td>Recv (A, data)</td>
</tr>
<tr>
<td>Send (A, data')</td>
<td></td>
</tr>
</tbody>
</table>

As opposed to:

<table>
<thead>
<tr>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMMD_send_and_receive (B, data)</td>
<td>CMMD_send_and_receive (A, data)</td>
</tr>
</tbody>
</table>

Note the two operations version is made more complex by the need to copy data values to prevent them being over written before they are copied into the message layer, a common complication found in many wavefront calculations.

So what did we get?

From a very restrictive system (Caltech Hypercube) to systems with multiple ways of sending a message that is addressed in a flexible way and tagged.

Receivers have multiple ways of filtering messages (using addresses, tags, channels) and can start a receive and get back to it when finished.

m4 and p4 macros

P4 was a system, that grew out of a set of fortran macros that was was developed at Argonne National Laboratory (ANL) for use on a NSF shared memory processor.

The original Macros were called Horipass that were processed at compile time by the Unix m4 preprocessing utility, and offered the user a set of monitor functions used to provide locks on critical sections of code that accessed shared data.

The use of macros avoided an additional set of stack operations if the monitors had be based on function calls.

The authors of this system co-wrote the book "Parallel Programs for Parallel Processors" which give the system its final name of p4.
m4 and p4 macros

The system was used as a base for several specialized versions

- TCGMSG for Chemistry problems
- GMD macros for solving problems on regular grids
  - GMD macros was the basis for the very successful PARMACS system developed by Rolf Hempel at GMD in Germany.

  He was at NEC, Germany. And NEC has one of the fastest MPI implementations in existence for their SX series computers.

- The system was used as a base for several specialized versions
  - TCGMSG for Chemistry problems
  - GMD macros for solving problems on regular grids
  - GMD macros was the basis for the very successful PARMACS system developed by Rolf Hempel at GMD in Germany.

m4 and p4 macros

The final p4 system was based on procedure calls and supported C as well as Fortran on a very wide range of systems including both distributed memory as well as shared memory systems.

- The programming paradigm was that of procs (processes) that formed administrative clusters, that intercommunicated by either locks or explicit message passing.

- The system provided user access to buffers so that they could avoid additional buffering by the system if they were knowledgeable enough.

- There was no non-locally blocking or asynchronous calls. I.e. the calls returned when the data was sent, and the user did not have to probe, test or wait for completion before reusing buffers.

- One globally blocking point to point call was also included p4sendr() which waited for an explicit acknowledgment from the receiver before returning, hence the r on the end of the call name to signify a rendezvous.

- The system also included a wide range of collective operations, with the ability to use the p4_global_op() call to construct user-defined operations.

- Although the p4 system was very efficient, it was not as popular as other message passing systems and was later used as a layering scheme to support other projects such as:
  - Chameleon {Gropp93}
  - BlockComm
  - MPI in the form of MPICH {Gropp94,Gropp95}.

What have we got this far?

- Non-vendor versions are more portable
- Some are more efficient than others
- Some are almost as fast as the vendor systems
- The number of features has increased
- Group communications better supported
- First portable high level libraries appearing
- Layering approach appearing
- MPI based on the CH ADE on top of p4... Or even PVM

Whats next?

- MPI standard and API
  - You will need to look at those handouts!

- What we will cover
  - Safe communications – communicators, groups
  - Message passing semantics – blocking, non blocking, local, global operations
  - Collectives, Buffering and data types
Message Passing

Introduction to MPI

What is MPI?

April 1992 was the beginning of the MPI forum
Organized at SC92
Consisted of hardware vendors, software vendors, academicians, and end users
Held 2 day meetings every 6 weeks
Created drafts of the MPI standard
This standard was to include all the functionality believed to be needed to make the message passing model a success
Final version released may, 1994

What is MPI?

A standard library specification!
Defines syntax and semantics of an extended message passing model
It is not a language or compiler specification
It is not a specific implementation
It does not give implementation specifics
Hints are offered, but implementers are free to do things however they want
Different implementations may do the same thing in a very different manner
http://www.mpi-forum.org

What is MPI

A library specification designed to support parallel computing in a distributed memory environment
Routines for cooperative message passing
There is a sender and a receiver
Point-to-point communication
Collective communication
Routines for synchronization
Derived data types for non-contiguous data access patterns
Ability to create sub-sets of processors
Ability to create process topologies

Notes

This talk is a combination of lots of different material from a host of sources including:
David Cronk & David Walker
EPCC
NCSA
FT-MPI, Open MPI, LAM and MPICH teams
What is MPI?

Continuing to grow!
- New routines have been added to replace old routines
- New functionality has been added
- Dynamic process management
- One sided communication
- Parallel I/O

Getting Started with MPI

Outline
- Introduction
- 6 basic functions
- Basic program structure
- Groups and communicators
- A very simple program
- Message passing
- A simple message passing example
- Types of programs
  - Traditional
  - Master/Slave
  - Examples
- Unsafe communication

Getting Started with MPI

MPI contains 128 routines (more with the extensions!)
Many programs can be written with just 6 MPI routines!
Upon startup, all processes can be identified by their rank, which goes from 0 to N-1 where there are N processes

6 Basic Functions

MPI_INIT (ierr)

ierr: Integer error return value. 0 on success, non-zero on failure.
This MUST be the first MPI routine called in any program.
Except for MPI_Initialized () can be called to check if MPI_Init has been called!!
Can only be called once
Sets up the environment to enable message passing

MPI_FINALIZE (ierr)

ierr: Integer error return value. 0 on success, non-zero on failure.
This routine must be called by each process before it exits
This call cleans up all MPI state
No other MPI routines may be called after MPI_FINALIZE
All pending communication must be completed (locally) before a call to MPI_FINALIZE
Basic Program Structure

```c
program main
#include "mpi.h"
integer ierr
   call MPI_INIT(ierr)
   Do some work
   call MPI_FINALIZE(ierr)
end
```

Groups and communicators

Communicators are containers that hold messages and groups of processes together with additional meta-data.

All messages are passed only within communicators.

Upon startup, there is a single set of processes associated with the communicator `MPI_COMM_WORLD`.

Groups can be created which are sub-sets of this original group, also associated with communicators.

Why do communicators exist?

To keep different message passing libraries from interfering with each other.

Allows the building of multiple layers of message passing code.

Groups and communicators

Nothing to stop message passing to the wrong layer….
MPI_COMM_RANK (comm, rank, ierr)

- comm: Integer communicator.
- rank: Returned rank of calling process
- ierr: Integer error return code

This routine returns the relative rank of the calling process, within the group associated with comm.

MPI_COMM_SIZE (comm, size, ierr)

- comm: Integer communicator identifier
- size: Upon return, the number of processes in the group associated with comm. For our purposes, always the total number of processes
- ierr: Integer error return code

This routine returns the number of processes in the group associated with comm.

A Very Simple Program
Hello World

program main
include 'mpi.h'
integer ierr, size, rank

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierr)
print *, ‘Hello World from process’, rank, ‘of’, size
call MPI_FINALIZE (ierr)
end

Hello World

> mpirun –np 4 a.out
> Hello World from 2 of 4
> Hello World from 0 of 4
> Hello World from 3 of 4
> Hello World from 1 of 4

> mpirun –np 4 a.out
> Hello World from 3 of 4
> Hello World from 1 of 4
> Hello World from 2 of 4
> Hello World from 0 of 4

Message Passing

Message passing is the transfer of data from one process to another.
This transfer requires cooperation of the sender and the receiver, but is initiated by the sender.
There must be a way to "describe" the data.
There must be a way to identify specific processes.
There must be a way to identify messages.

Data is described by a triple:
- Address: Where is the data stored
- Count: How many elements make up the message
- Datatype: What is the type of the data

Basic types (integers, reals, etc)
Derived types (good for non-contiguous data access)
Message Passing

Processes are specified by a double
Communicator: safe space to pass message
Rank: The relative rank of the specified process within the group associated with the communicator
Messages are identified by a single tag
This can be used to differentiate between different types of messages
Max tag can be looked up but must be at least 32k

MPI_SEND(buf, cnt, dtype, dest, tag, comm, ierr)

buf: The address of the beginning of the data to be sent
cnt: The number of elements to be sent
dtype: Datatype of each element
dest: The rank of the destination
tag: The message tag
comm: The communicator

Once this routine returns, the message has been copied out of the user buffer and the buffer can be reused
This may require the use of system buffers. If there are insufficient system buffers, this routine will block until a corresponding receive call has been posted
Completion of this routine indicates nothing about the designated receiver

MPI_RECV (buf, cnt, dtype, source, tag, comm, status, ierr)

buf: Starting address of receive buffer
cnt: Max number of elements to receive
dtype: Datatype of each element
source: Rank of sender (may use MPI_ANY_SOURCE)
tag: The message tag (may use MPI_ANY_TAG)
comm: Communicator
status: Status information on the received message

When this call returns, the data has been copied into the user buffer
Receiving fewer than cnt elements is ok, but receiving more is an error
Status is a structure in C (MPI_Status) and an array in Fortran (integer status(MPI_STATUS_SIZE))

MPI_STATUS

The status parameter is used to retrieve information about a completed receive
In C, status is a structure consisting of at least 3 fields: MPI_SOURCE, MPI_TAG, MPI_ERROR
status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code, respectively
In Fortran, status must be an integer array of size MPI_STATUS_SIZE
status(MPI_SOURCE), status(MPI_TAG), and status(MPI_ERROR) contain the source, tag, and error code
Send/Recv Example

program main
include 'mpi.h'
CHARACTER*20 msg
integer ierr, rank, tag, status (MPI_STATUS_SIZE)
tag = 99
call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
if (myrank .eq. 0) then
  msg = "Hello there"
call MPI_SEND (msg, 11, MPI_CHARACTER, 1, tag, &
  MPI_COMM_WORLD, ierr)
else if (myrank .eq. 1) then
  call MPI_RECV(msg, 20, MPI_CHARACTER, 0, tag, &
  MPI_COMM_WORLD, status, ierr)
endif
  call MPI_FINALIZE (ierr)
end

Types of MPI Programs

Traditional
Break the problem up into about even sized parts and distribute across
all processors
What if problem is such that you cannot tell how much work must be
done on each part?
Master/Slave
Break the problem up into many more parts than there are processors
Master sends work to slaves
Parts may be all the same size or the size may vary

Traditional Example

Compute the sum of a large array of N integers

Comm = MPI_COMM_WORLD
Call MPI_COMM_RANK (comm, rank)
Call MPI_COMM_SIZE (comm, npes)
Stride = N/npes
Start = (stride * rank) + 1
Sum = 0
DO (I = start, start+stride)
  sum = sum + array(I)
ENDDO

If (rank .eq. 0) then
  DO (I = 1, npes-1)
    call MPI_RECV(tmp, 1, MPI_INTEGER, &
    I, 1, 2, comm, status)
    sum = sum + tmp
  ENDDO
  ELSE
    MPI_SEND (sum, 1, MPI_INTEGER, &
      0, 2 comm)
  ENDIF

Unsafe Communication Patterns

Process 0 and process 1 must exchange data
Process 0 sends data to process 1 and then receives data from
process 1
Process 1 sends data to process 0 and then receives data from
process 0
If there is not enough system buffer space for either message, this
will deadlock
Any communication pattern that relies on system buffers is unsafe
Any pattern that includes a cycle of blocking sends is unsafe

Unsafe Communication Patterns

Communication Modes

Outline
Standard mode
Blocking
Non-blocking
Non-standard mode
Buffered
Synchronous
Ready
Performance issues
Point-to-Point Communication Modes

Standard Mode:

blocking:

MPI_SEND (buf, count, datatype, dest, tag, comm)

MPI_RECV (buf, count, datatype, source, tag, comm, status)

Generally ONLY use if you cannot call earlier AND there is no other work that can be done!

Standard ONLY states that buffers can be used once calls return. It is implementation dependent on when blocking calls return.

Blocking sends MAY block until a matching receive is posted. This is not required behavior, but the standard does not prohibit this behavior either.

Further, a blocking send may have to wait for system resources such as system managed message buffers.

Be VERY careful of deadlock when using blocking calls!

Point-to-Point Communication Modes (cont)

Standard Mode:

Non-blocking (immediate) sends/receives:

MPI_ISEND (buf, count, datatype, dest, tag, comm, request)

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

MPI_WAIT (request, status)

MPI_TEST (request, flag, status)

Allows communication calls to be posted early, which may improve performance, overlap computation and communication latency tolerance

Loss of visibility

* MUST either complete these calls (with wait or test) or call MPI_REQUEST_FREE

MPI_ISEND (buf, cnt, dtype, dest, tag, comm, request)

Same syntax as MPI_SEND with the addition of a request handle

Request is a handle (int in Fortran) used to check for completeness of the send

This call returns immediately

Data in buf may not be accessed until the user has completed the send operation

The send is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_IRECV(buf, cnt, dtype, source, tag, comm, request)

Same syntax as MPI_RECV except status is replaced with a request handle

Request is a handle (int in Fortran) used to check for completeness of the recv

This call returns immediately

Data in buf may not be accessed until the user has completed the receive operation

The receive is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_WAIT (request, status)

Request is the handle returned by the non-blocking send or receive call

Upon return, status holds source, tag, and error code information

This call does not return until the non-blocking call referenced by request has completed

Upon return, the request handle is freed

If request was returned by a call to MPI_ISEND, return of this call indicates nothing about the destination process

MPI_TEST (request, flag, status)

Request is a handle returned by a non-blocking send or receive call

Upon return, flag will have been set to true if the associated non-blocking call has completed. Otherwise it is set to false

If flag returns true, the request handle is freed and status contains source, tag, and error code information

If request was returned by a call to MPI_ISEND, return with flag set to true indicates nothing about the destination process
Non-blocking Communication

100 continue
if (err .lt. Delta) goto 200
do some computation
if (I = 0, npe)
set up data to send
call MPI_SEND (data, cnt, dtype, &I, tag, comm, ierr)
endif
endif
do (I = 0, npes)
if (I .ne. myrank)
set up data to recv
call MPI_RECV (data, cnt, dtype, &I, tag, comm, status, ierr)
endif
goto 100

Clearly unsafe
May run out of handles

100 continue
if (err .lt. Delta) goto 200
do some computation
if (I = 0, npe)
set up data to send
call MPI_ISEND (data, cnt, dtype, &I, tag, comm, request, ierr)
endif
endif
do (I = 0, npes)
if (I .ne. myrank)
set up data to recv
call MPI_IRECV (data, cnt, dtype, &I, tag, comm, request, ierr)
endif
goto 100

Safe, and pretty good

Point-to-Point Communication Modes

Non-standard mode communication
Only used by the sender! (MPI uses the push communication model)
Buffered mode - A buffer must be provided by the application
Synchronous mode - Completes only after a matching receive has been posted
Ready mode - May only be called when a matching receive has already been posted

Buffered Sends

MPI_BSEND (buf, count, datatype, dest, tag, comm)
MPI_IBSEND (buf, count, dtype, dest, tag, comm, req)
MPI_BUFFER_ATTACH (buff, size)
MPI_BUFFER_DETACH (buff, size)
Buffered sends do not rely on system buffers
User need not worry about calls blocking, waiting for system buffer space
The buffer is managed by MPI
The user MUST ensure there is no buffer overflow

Point-to-Point Communication Modes: Synchronous

MPI_SSEND (buf, count, datatype, dest, tag, comm)
MPI_ISSEND (buf, count, datatype, dest, tag, comm, req)
Can be started (called) at any time.
Does not complete until a matching receive has been posted and the receive operation has been started.
* Does NOT mean the matching receive has completed
Can be used in place of sending and receiving acknowledgements
Can be more efficient when used appropriately buffering may be avoided
**Point-to-Point Communication Modes: Ready Mode**

- **MPI_RSEND** (buf, count, datatype, dest, tag, comm)
- **MPI_IRSEND** (buf, count, datatype, dest, tag, comm, req)

  - **Ready Mode**
  - Should only be used with **extreme caution**.
  - Only really faster on a Paragon!

**Performance Issues**

- Non-blocking calls are almost always the way to go.
- Communication can be carried out during blocking system calls.
- Computation and communication can be overlapped if there is special purpose communication hardware.
- Less likely to have errors that lead to deadlock.
- Standard mode is usually sufficient - but buffered mode can offer advantages.
- Particularly if there are frequent, large messages being sent.
- If the user is unsure the system provides sufficient buffer space.
- Synchronous mode can be more efficient if acks are needed.
- Also tells the system that buffering is not required.
  - But, if no overlapping then non blocking is Slower due to extra data structures and book keeping!
  - Only way to know... Benchmark it!

**Point 2 point case study**

Master has a large number of 'tests' that need to ran and some average result needs to be calculated.

We will consider four things:
- Overall execution structure
- What this means for message passing
- Performance issue
- Improving the structure for better performance

**P2p example**
P2p case study

What to consider
- Do we have more work than workers?
- How big is the work?
- Is the work independent from each other?
- Are there intermediate results?
- Do they need to get shared? Stored on disk?
- Does the master need to do some work as well?

Work definition is small...

Work result needs to be at the master

Master sends data to the workers: Slaves receive work
Point 2 point case study

To consider how this looks as a parallel algorithm we need to draw it as a DAG.
If the master does work as well.

Data does not need to move?

If you do not distribute or send out all work in one go we need to loop until all the work is done.

What does this look like in terms of code?

The job of each process is defined by who they are (master or slave)

In MPI we can use RANK to define a master

RANK can also identify who the slaves are

Arrows in the graphs are data in the form of messages

Depending on if your master or worker and which arc, we know if we are sending to receiving data.
P2p case study

Master
MPI_Init ( )
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if (rank==0) {
    /* I AM MASTER */
    Do_master ( )
}
MPI_Finalize ( )

Worker
MPI_Init ( )
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if (rank!=0) {
    /* I am Worker */
    Do_worker ( rank )
}
MPI_Finalize ( )

To make the previous code work if the work does not divide up into the workers correctly you need to change the data being sent:
Special value for no-more work
you need to tell workers how much work they have they can ask for work

Master:
work has 3 pieces of work...
MPI_Send (howmuch, work..)
loop {
    MPI_Send (work[..])
}

Slave:
do_work
MPI_recv (howmuch,..)
loop (1..howmuch)
MPI_Recv (work, 1, 0, .. Status)
Presult = Do_work ( ) ...
MPI_Send (Presult..)

you need to tell workers how much work they have

you need to tell workers how much work they have

You need to do:
/* get work */
MPIRecv (work, 1, 0, status)
Presult = Do_work (work)
MPI_Send (work, 1, 0,..)

you need to tell workers how much work they have

Special value for no-more work
Loop ()
    MPI_Recv (work, 1, 0, .. Status)
    If (work==NOWORKLEFT) return ();
    Else
        Presult = Do_work (work) ...
        MPI_Send (Presult...)
}
P2p case study

they can ask for work

Master
If work-left or workers-still-working{  
MPI_Recv (what&who..)  
If what==result add it to partial result  
If work-left MPI_Send (nextwork, who..)  
Else MPI_Send (NOMOREWORK, who..)  
}

Worker
MPI_Send (Iwantsomework, 1, 0..)  
Loop {  
MPI_Recv (work, 1, 0, .. Status)  
If (work==NOWORKLEFT) return ();  
Else  
Presuit = Do_work ( ) ...  
MPI_Send (Presuit..)  
}

Collective Communication

Outline
Introduction
Barriers
Broadcasts
Gather
Scatter
All gather
Alltoall
Reduction
Performance issues

Collective Communication

Total amount of data sent must exactly match the total amount of data received
Collective routines are collective across an entire communicator and must be called in the same order from all processors within the communicator
Collective routines are all blocking
This simply means buffers can be re-used upon return
Collective routines return as soon as the calling process' participation is complete
Does not say anything about the other processors
Collective routines may or may not be synchronizing
No mixing of collective and point-to-point communication

Collective Communication

Barrier: MPI_BARRIER (comm)
Only collective routine which provides explicit synchronization
Returns at any processor only after all processes have entered the call
Collective Communication: Bcast

MPI_BCAST (buffer, count, datatype, root, comm)
- Strictly in place
- MPI-1 insists on using an intra-communicator
- MPI-2 allows use of an inter-communicator

**REMINDER**: A broadcast need not be synchronizing. Returning from a broadcast tells you nothing about the status of the other processes involved in a broadcast. Furthermore, though MPI does not require MPI_BCAST to be synchronizing, it neither prohibits synchronous behavior.

```
if (myrank == root) {
    fp = fopen (filename, 'r');
    fscanf (fp, '%d', &iters);
    fclose (fp);
    MPI_Bcast (&iters, 1, MPI_INT,
               root, MPI_COMM_WORLD);
} else {
    MPI_Recv (&iters, 1, MPI_INT,
              root, tag, MPI_COMM_WORLD,
              &status);
}
```

Collective Communication: Gather

MPI_GATHER (sendbuf, sendcount, sendtype, recvbuf, recvcounts, recvtype, root, comm)
- Receive arguments are only meaningful at the root
- Each processor must send the same amount of data
- Root can use MPI_IN_PLACE for sendbuf:
  data is assumed to be in the correct place in the recvbuf

```
P1 = root
P2
P3
P4
```

Collective Communication: Gatherv

MPI_GATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
- Vector variant of MPI_GATHER
- Allows a varying amount of data from each proc
- Allows root to specify where data from each proc goes
- No portion of the receive buffer may be written more than once
- MPI_IN_PLACE may be used by root.

```
1   2    3    4 counts
9   7    4    0 displs
P1 = root
P2
P3
P4
```

Collective Communication: Gatherv (cont)

```
100 100 100 100
```

REMINDER: OOPS!
Collective Communication: Scatter

MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
Opposite of MPI_GATHER
Send arguments only meaningful at root
Root can use MPI_IN_PLACE for recvbuf

B (on root)

Collective Communication: Scatterv (cont)

MPI_SCATTERV (sendbuf, scounts, displs, sendtype, recvbuf, recvcount, recvtype)
Opposite of MPI_GATHERV
Send arguments only meaningful at root
Root can use MPI_IN_PLACE for recvbuf
No location of the sendbuf can be read more than once

Collective Communication: Allgatherv

MPI_ALLGATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)
Same as MPI_GATHERV, except all processors get the result
MPI_IN_PLACE may be used for sendbuf of all processors
Equivalent to a gather followed by a bcast
Collective Communication: 

**MPI_ALLTOALLV**

- Same as MPI_ALLTOALL, but the vector variant
- Can specify how many blocks to send to each processor, location of blocks to send, how many blocks to receive from each processor, and where to place the received blocks

**MPI_ALLTOALLW**

- Same as MPI_ALLTOALLV, except different datatypes can be specified for data scattered as well as data gathered
- Can specify how many blocks to send to each processor, location of blocks to send, how many blocks to receive from each processor, and where to place the received blocks
- Displacements are now in terms of bytes rather than types

Collective Communication: Reduction

- Global reduction across all members of a group
- Can use predefined operations or user defined operations
- Can be used on single elements or arrays of elements
- Counts and types must be the same on all processors
- Operations are assumed to be associative
- User defined operations can be different on each processor, but not recommended

**MPI_REDUCE**

- Combines elements on an element by element basis into sendbuf on root
- Results of the reduction are returned to root in recvbuf
- MPI_IN_PLACE can be used for sendbuf on root

```plaintext
REAL a(n), b(n,m), c(m)
REAL sum(m)
DO j=1,m
  sum(j) = 0.0
  DO i = 1,n
    sum(j) = sum(j) + a(i)*b(i,j)
  ENDDO
ENDDO
CALL MPI_REDUCE(sum, c, m, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```

**MPI_ALLREDUCE**

- Same as MPI_REDUCE, except all processors get the result

**MPI_REDUCE_SCATTER**

- Acts like it does a reduce followed by a scatterv

```plaintext
CALL MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```
Collective Communication: Reduction - user defined ops

MPI_OP_CREATE (function, commute, op)
If commute is true, operation is assumed to be commutative
Function is a user defined function with 4 arguments
invec: input vector
inoutvec: input and output value
len: number of elements
datatype: MPI_DATATYPE
Returns invec[i] op inoutvec[i], i = 0..len-1
MPI_OP_FREE (op)

Collective Communication: Performance Issues
Collective operations should have much better performance than simply sending messages directly
Broadcast may make use of a broadcast tree (or other mechanism)
All collective operations can potentially make use of a tree (or other) mechanism to improve performance
Important to use the simplest collective operations which still achieve the needed results
Use MPI_IN_PLACE whenever appropriate
Reduces unnecessary memory usage and redundant data movement

Case study again
In the previous example we sent all the work out using point to point calls
Received all the results using point to pint calls.
Could use collectives

Case study
Broadcast or scatter?

Case study
If broadcast all nodes get the same set of work
the workers have to understand what work they are doing
CASE STUDY

If scatter then custom work per worker can be sent

What Else is There

- Lots of other routines
- Derived datatypes
- Process groups and communicators
- Process topologies
- Profiling
- MPI-2
  - Parallel I/O
  - Dynamic process management
  - One sided communication

Communicators and Groups

- Many MPI users are only familiar with MPI_COMM_WORLD
- A communicator can be thought of as a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
  - For many applications (dual level parallelism) maintaining different groups is appropriate
  - Groups allow collective operations to work on a subset of processes
  - Information can be added onto communicators to be passed into routines

Communicators and Groups (cont)

- While we think of a communicator as spanning processes, it is actually unique to a process
- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes
  - An intracommunicator is used for communication within a single group
  - An intercommunicator is used for communication between disjoint groups
Communicators and Groups (cont)

There are 3 distinct groups
- MPI_COMM_WORLD
- Comm1
- Comm2

P3 is a member of all 3 groups and may have different ranks in each group (say 0, 3, & 4)

P2 wants to send a message to P1 it can use
- MPI_COMM_WORLD (intracommunicator)
- An intercommunicator (covered later)

P2 wants to send a message to P3 it can use
- MPI_COMM_WORLD (send to rank 0)
- Comm1 (send to rank 3)

Group Management
- All group operations are local
- Groups are initially not associated with communicators
- Groups can only be used for message passing within a communicator
- We can access groups, construct groups, and destroy groups

Group Accessors
- MPI_GROUP_SIZE (group, size)
  - MPI_Group group
  - int size
  - This routine returns the number of processes in the group

- MPI_GROUP_RANK (group, rank)
  - MPI_Group group
  - int rank
  - This routine returns the rank of the calling process

Group Accessors (cont)
- MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)
  - MPI_Group group1, group2
  - int n, *ranks1, *ranks2
  - This routine takes an array of n ranks (ranks1) which are ranks of processes in group1. It returns in ranks2 the corresponding ranks of the processes as they are in group2
  - MPI_UNDEFINED is returned for processes not in group2

Groups Accessors (cont)
- MPI_GROUP_COMPARE (group1, group2, result)
  - MPI_Group group1, group2
  - int result
  - This routine returns the relationship between group1 and group2
  - MPI_IDENT: If group1 and group2 contain the same processes, ranked the same way, this routine returns MPI_IDENT
  - MPI_SIMILAR: If group1 and group2 contain the same processes, but ranked differently, this routine returns MPI_SIMILAR
  - MPI_UNEQUAL: Otherwise this routine returns MPI_UNEQUAL
Group Constructors

Group constructors are used to create new groups from existing groups. Base group is the group associated with MPI_COMM_WORLD. Group creation is a local operation and no communication is needed. Following group creation, no communicator is associated with the group and no communication is possible with the new group.

Group Constructors (cont)

MPI_COMM_GROUP (comm, group)
MPI_Comm comm
MPI_Group group
This routine returns in group the group associated with the communicator comm.

Set Operations

- MPI_GROUP_UNION(group1, group2, newgroup)
- MPI_GROUP_INTERSECTION(group1, group2, newgroup)
- MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

Set Operations (cont)

Let group1 = \{a,b,c,d,e,f,g\} and group2 = \{d,g,a,c,h,l\}
- MPI_Group_union(group2, group1, newgroup)
  - Newgroup = \{d,g,a,c,h,l,b,e,f\}
- MPI_Group_intersection(group2, group1, newgroup)
  - Newgroup = \{d,g,a,c\}
- MPI_Group_difference(group1, group2, newgroup)
  - Newgroup = \{b,e,f\}
Group Constructors (cont)

MPI_GROUP_INCL(group, n, ranks, newgroup)
MPI_Group group, *newgroup
int n, *ranks
This routine creates a new group that consists of all the n processes with ranks ranks[0]..ranks[n-1]
The process with rank i in newgroup has rank ranks[i] in group

Group Constructors (cont)

MPI_GROUP_EXCL(group, n, ranks, newgroup)
MPI_Group group, *newgroup
int n, *ranks
This routine creates a new group that consists of all the processes in group after deleting processes with ranks ranks[0]..ranks[n-1]
The ordering in newgroup is identical to the ordering in group

Group Constructors (cont)

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
MPI_Group group, *newgroup
int n, ranges[3][3]
Ranges is an array of triplets consisting of start rank, end rank, and stride
Each triplet in ranges specifies a sequence of ranks to be included in newgroup
The ordering in newgroup is as specified by ranges

Group Constructors (cont)

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)
MPI_Group group, *newgroup
int n, ranges[3][3]
Ranges is an array of triplets consisting of start rank, end rank, and stride
Each triplet in ranges specifies a sequence of ranks to be excluded from newgroup
The ordering in newgroup is identical to that in group

Communicator Management

Communicator access operations are local, thus requiring no interprocess communication
Communicator constructors are collective and may require interprocess communication
All the routines in this section are for intracomunicators, intercommunicators will be covered separately
Communicator Accessors

- **MPI_COMM_SIZE (comm, size)**
  Returns the number of processes in the group associated with comm

- **MPI_COMM_RANK (comm, rank)**
  Returns the rank of the calling process within the group associated with comm

- **MPI_COMM_COMPARE (comm1, comm2, result)** returns:
  - MPI_IDENT if comm1 and comm2 are handles for the same object
  - MPI_CONGRUENT if comm1 and comm2 have the same group attribute
  - MPI_SIMILAR if the groups associated with comm1 and comm2 have the same members but in different rank order
  - MPI_UNEQUAL otherwise

Communicator Constructors

- **MPI_COMM_DUP (comm, newcomm)**
  This routine creates a duplicate of comm
  newcomm has the same fixed attributes as comm
  Defines a new communication domain
  Useful to library writers and library users

- **MPI_COMM_CREATE (comm, group, newcomm)**
  This is a collective routine, meaning it must be called by all processes in the group associated with comm
  This routine creates a new communicator which is associated with group
  MPI_COMM_NULL is returned to processes not in group
  All group arguments must be the same on all calling processes
  group must be a subset of the group associated with comm

- **MPI_COMM_SPLIT (comm, color, key, newcomm)**
  MPI_Comm comm, newcomm
  int color, key
  This routine creates as many new groups and communicators as there are distinct values of color
  The rankings in the new groups are determined by the value of key, ties are broken according to the ranking in the group associated with comm
  MPI_UNDEFINED is used as the color for processes to not be included in any of the new groups

Communication Constructors

- **MPICOMM**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

  Both process a and j are returned MPI_COMM_NULL
  3 new groups are created
  \{i, c, d\}
  \{k, b, e, g, h\}
  \{f\}

Destructors

The communicators and groups from a process' viewpoint are merely handles
Like all handles in MPI, there is a limited number available – YOU CAN RUN OUT
MPI_GROUP_FREE (group)
MPI_COMM_FREE (comm)
Intercommunicators

Intercommunicators are associated with 2 groups of disjoint processes
Intercommunicators are associated with a remote group and a local group
A communicator is either intra or inter, never both

Intercommunicator Accessors

MPI_COMM_TEST_INTER (comm, flag)
This routine returns true if comm is an intercommunicator, otherwise, false
MPI_COMM_REMOTE_SIZE(comm, size)
This routine returns the size of the remote group associated with intercommunicator comm
MPI_COMM_REMOTE_GROUP(comm, group)
This routine returns the remote group associated with intercommunicator comm

Intercommunicator Constructors

The communicator constructors described previously will return an intercommunicator if the are passed intercommunicators as input
MPI_COMM_DUP: returns an intercommunicator with the same groups as the one passed in
MPI_COMM_CREATE: each process in group A must pass in group the same subset of group A (A1). Same for group B (B1). The new communicator has groups A1 and B1 and is only valid on processes in A1 and B1
MPI_COMM_SPLIT: As many new communicators as there are distinct pairs of colors are created

Communication Constructors

Processes a, j, l, o, and u would all have MPI_COMM_NULL returned in newcomm
newcomm1 would be associated with 2 groups: {e, i, d} and {t, n}
newcomm2 would be associated with 2 groups: {k, b, c, g, h} and {v, m, p, r, q}
newcomm3 would be associated with 2 groups: {f} and {s}
Intercommunicator Constructors

MPI_INTERCOMM_CREATE (local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)
This routine is called collectively by all processes in 2 disjoint groups.
All processes in a particular group must provide matching local_comm and local_leader arguments.
The local leaders provide a matching bridge_comm (a communicator through which they can communicate), in remote_leader the rank of the other leader within bridge_comm, and the same tag.
The bridge_comm, remote_leader, and tag are significant only at the leaders.
There must be no pending communication across bridge_comm that may interfere with this call.

Intercommunicators

MPI_INTERCOMM_MERGE (intercomm, high, newintracomm)
This routine creates an intracommunicator from a union of the two groups associated with intercomm.
High is used for ordering. All processes within a particular group must pass the same value in for high (true or false).
The new intracommunicator is ordered with the high processes following the low processes.
If both groups pass the same value for high, the ordering is arbitrary.

Attribute Caching

It is possible to cache attributes to be associated with a communicator.
This cached information is process specific.
The same attribute can be cached with multiple communicators.
Many attributes can be cached with a single communicator.
This is most commonly used in libraries.

Selected References

MPI - The Complete Reference Volume 1, The MPI Core
MPI - The Complete Reference Volume 2, The MPI Extensions
USING MPI: Portable Parallel Programming with the Message-Passing Interface
Using MPI-2: Advanced Features of the Message-Passing Interface