Ateneo de Manila University

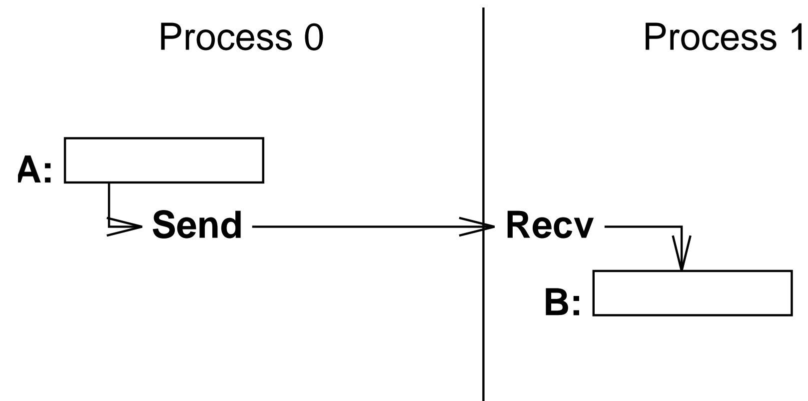
Parallel Programming with MPI: Point to Point Communication

Ateneo High Performance Computing Group
1st Semester
http://www.math.admu.edu.ph/ahpc/
william.s.yu@ieee.org

Point to Point Communication

Sending and Receiving Messages

- Basic message passing process



- Questions:
  - To whom is data sent?
  - Where is the data?
  - How much of the data is sent?
  - What type of the data is sent?
  - How does the receiver identify it?

Current Message-Passing

- A typical send might look like:

```
send(dest, address, length)
```

  - `dest` is an integer identifier representing the process to receive the message.
  - (`address, length`) describes a contiguous area in memory containing the message to be sent.

Traditional Buffer Specification

Sending and receiving only a contiguous array of bytes:

- Hides the real data structure from hardware which might be able to handle it directly
- Requires pre-packing of dispersed data
  - Rows of a matrix stored columnwise
  - General collections of structures
- Prevents communications between machines with different representations (even lengths) for same data type, except if user works this out

Generalizing the Buffer Description

- Specified in MPI by *starting address*, *datatype*, and *count*, where datatype is:
  - Elementary (all C and Fortran datatypes)
  - Contiguous array of datatypes
  - Strided blocks of datatypes
  - Indexed array of blocks of datatypes
  - General structure
- Datatypes are constructed recursively.
- Specifications of elementary datatypes allows heterogeneous communication.
- Elimination of length in favor of count is clearer.
- Specifying application-oriented layout of data allows maximal use of special hardware.

MPI C Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int

MPI C Datatypes (cont.)

MPI datatype	C datatype
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI C++ Datatypes

MPI datatype	C++ datatype
MPI::CHAR	signed char
MPI::SHORT	signed short int
MPI::INT	signed int
MPI::LONG	signed long int
MPI::UNSIGNED_CHAR	unsigned char
MPI::UNSIGNED_SHORT	unsigned short int
MPI::UNSIGNED	unsigned int

MPI C++ Datatypes (cont.)

MPI datatype	C++ datatype
MPI::UNSIGNED_LONG	unsigned long int
MPI::FLOAT	float
MPI::DOUBLE	double
MPI::LONG_DOUBLE	long double
MPI::BYTE	
MPI::PACKED	

MPI Fortran Datatypes

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Generalizing the Process Identifier

- `destination` has become `(rank, group)`.
- Processes are named according to their rank in the group
- Groups are enclosed in “communicators”
- `MPI_ANY_SOURCE` wildcard permitted in a receive.

Providing Safety

- MPI provides support for safe message passing (e.g. keeping user and library messages separate)
- Safe message passing
  - Communicators also contain "contexts"
  - Contexts can be envisioned as system-managed tags
- Communicators can be thought of as (`group, system-tag`)
- `MPI_COMM_WORLD` contains a "context" and the "group of all known processes"
- Collective and point-to-point messaging is kept separate by "context"

Identifying the Message

- MPI uses the word “tag”
- Tags allow programmers to deal with the arrival of messages in an orderly way
- MPI tags are guaranteed to range from 0 to 32767
- The range will always start with 0
- The upper bound may be larger than 32767. Section 7.1.1 of the standard discusses how to determine if an implementation has a larger upper bound
- `MPI_ANY_TAG` can be used as a wildcard value

MPI Basic Send/Receive

- Thus the basic (blocking) send has become:

```
MPI_SEND(start, count, datatype, dest, tag, comm)
```

- And the receive has become:

```
MPI_RECV(start, count, datatype, source, tag, comm,
         status)
```

- The source, tag, and count of the message actually received can be retrieved from `status`.

- For now, `comm` is `MPI_COMM_WORLD` or `MPI::COMM_WORLD`

MPI Procedure Specification

- MPI procedures are specified using a language independent notation.
- Procedure arguments are marked as

  IN: the call uses but does not update the argument

  OUT: the call may update the argument

  INOUT: the call both uses and updates the argument
- MPI functions are first specified in the language-independent notation
- ANSI C and Fortran 77 realizations of these functions are the language *bindings*

MPI Basic Send

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

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (nonnegative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

Bindings for Send

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm)

void MPI::Comm::Send(const void* buf, int count,
                     const MPI::Datatype& datatype,
                     int dent, int tag) const;

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COM, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERR
```

MPI Basic Receive

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

OUT	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (nonnegative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	src	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

Bindings for Receive

```
int MPI_Recv(void *buf, int count, MPI_Datatype
             datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)

void MPI::Comm::Recv(void *buf, int count, const
                     Datatype & datatype, int source,
                     int tag, Status & status) const;

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
         STATUS, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
         STATUS(MPI_STATUS_SIZE), IERR
```

Getting Information About a Message

- The (non-opaque) `status` object contains information about a message

```
/* In C */
MPI_Status status;
MPI_Recv(..., &status);

recvd_tag    = status.MPI_TAG;
recvd_source = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);

/* In C++ */
MPI::Status status;
MPI::COMM_WORLD.Recv(..., status);

recvd_tag = status.Get_tag();
recvd_source = status.Get_source();
recvd_count = status.Get_count(datatype);
```

Getting Information About a Message (cont'd)

- The fields `status.MPI_TAG` and `status.MPI_SOURCE` are primarily of use when `MPI_ANY_TAG` and/or `MPI_ANY_SOURCE` is used in the receive
- The function `MPI_GET_COUNT` may be used to determine how much data of a particular type was received.

Copyright © 2000-2001 by William Emmanuel S. Yu and Jeff Squires. This material may be distributed only subject to the terms and conditions set forth in the Open Content License, v1.0 or later (the latest version is presently available at http://opencontent.org/opl.shtml).