## The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - » Synchronization
  - » Movement of data from one process's address space to another's.

1

## Types of Parallel Computing Models

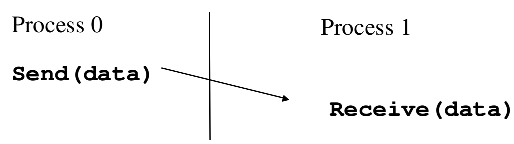
- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and MPI) is for MIMD/SPMD parallelism. HPF is an example of an SIMD interface.

2

# Cooperative Operations for Communication

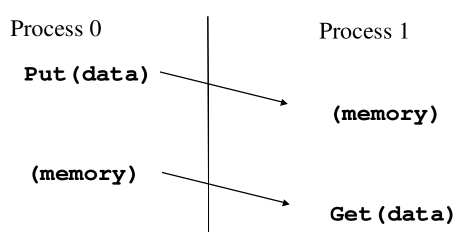
- The message-passing approach makes the exchange of data cooperative.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.




3

# One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.




4

## What is MPI?

- A *message-passing library specification*
  - » extended message-passing model
  - » not a language or compiler specification
  - » not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - » end users
  - » library writers
  - » tool developers

5

## Where Did MPI Come From?

- Early vendor systems (NX, EUI, CMMD) were not portable.
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts.
  - » Did not address the full spectrum of message-passing issues
  - » Lacked vendor support
  - » Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by vendors, library writers, and end users.
- MPI Standard (1.0) released June, 1994; many implementation efforts.
- MPI-2 Standard (1.2 and 2.0) released July, 1997.

6


3

## MPI Sources

- The Standard itself:
  - » at `http://www.mpi-forum.org`
  - » All MPI official releases, in both postscript and HTML
- Books:
  - » *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - » *MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra*, MIT Press, 1996.
  - » *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
  - » *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
- Other information on Web:
  - » at `http://www.mcs.anl.gov/mpi`
  - » pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages



## Notes on C and Fortran

- C and Fortran bindings correspond closely
- In C:
  - » mpi.h must be #included
  - » MPI functions return error codes or **`MPI_SUCCESS`**
- In Fortran:
  - » mpif.h must be included, or use MPI module
  - » All MPI calls are to subroutines, with a place for the return code in the last argument.
- C++ bindings, and Fortran-90 issues, are part of MPI-2.

## Error Handling

- By default, an error causes all processes to abort.
- The user can cause routines to return (with an error code) instead.
- A user can also write and install custom error handlers.
- Libraries might want to handle errors differently from applications.

9

## Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** is part of MPI-2, as a recommendation, but not a requirement, for implementors.

10

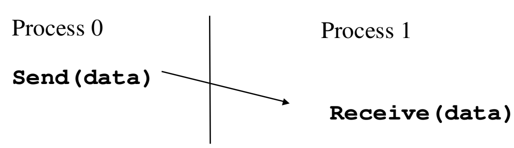
## Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - » How many processes are participating in this computation?
  - » Which one am I?
- MPI provides functions to answer these questions:
  - » **`MPI_Comm_size`** reports the number of processes.
  - » **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

11

## MPI Basic Send/Receive

- We need to fill in the details in




- Things that need specifying:
  - » How will "data" be described?
  - » How will processes be identified?
  - » How will the receiver recognize/screen messages?
  - » What will it mean for these operations to complete?

12


6

## Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

13

## MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

14

## MPI Basic (Blocking) Send

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

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

15

## MPI Basic (Blocking) Receive

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

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`status`** contains further information
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

16

# Retrieving Further Information

- **Status** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```



# Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - » this requires libraries to be aware of tags used by other libraries.
  - » this can be defeated by use of "wild card" tags.
- Contexts are different from tags
  - » no wild cards allowed
  - » allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application




9