

# An Introduction to MPI

## Parallel Programming with the Message Passing Interface

William Gropp

Ewing Lusk

Argonne National Laboratory

# Outline

- Background
  - The message-passing model
  - Origins of MPI and current status
  - Sources of further MPI information
- Basics of MPI message passing
  - Hello, World!
  - Fundamental concepts
  - Simple examples in Fortran and C
- Extended point-to-point operations
  - non-blocking communication
  - modes

# Outline (continued)

- Advanced MPI topics
  - Collective operations
  - More on MPI datatypes
  - Application topologies
  - The profiling interface
- Toward a portable MPI environment

# Companion Material

- Online examples available at <http://www.mcs.anl.gov/mpi/tutorials/perf>
- <ftp://ftp.mcs.anl.gov/mpi/mpiexmpl.tar.gz> contains source code and run scripts that allows you to evaluate your own MPI implementation

# 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.

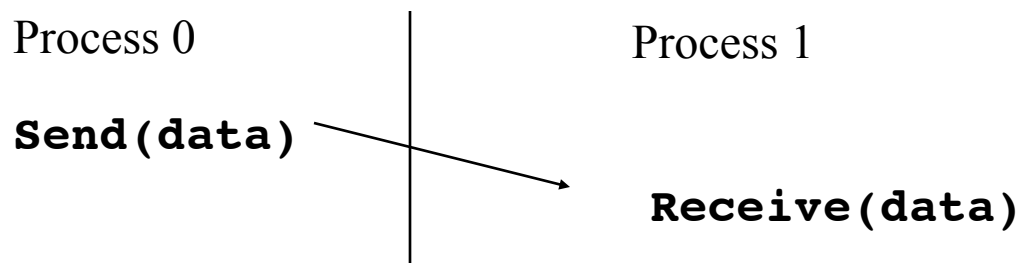
# Types of Parallel Computing Models

- 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.

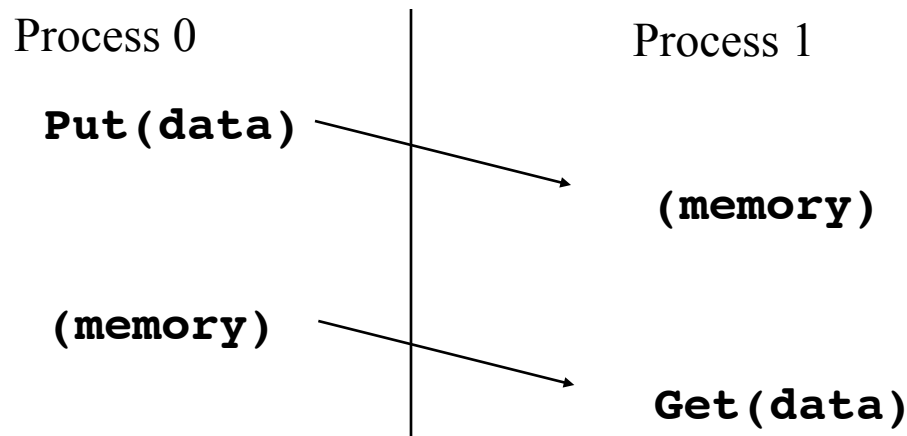
# Cooperative Operations for Communication

- 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.



# 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.





# 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

# 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.
  - *MPI: The Complete Reference Vol 1 and 2*, MIT Press, 1998(Fall).
- 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

# Why Use MPI?

- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- MPI was explicitly designed to enable libraries...
- ... which may eliminate the need for many users to learn (much of) MPI

# A Minimal MPI Program (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

# 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 (MPI-2)
  - 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.
  - In C++, exceptions are thrown (MPI-2)
- A user can also write and install custom error handlers.
- Libraries might want to handle errors differently from applications.

# 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.
- We use `mpirun(cs dept)` or `aprun (Cray)` in this class

# 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



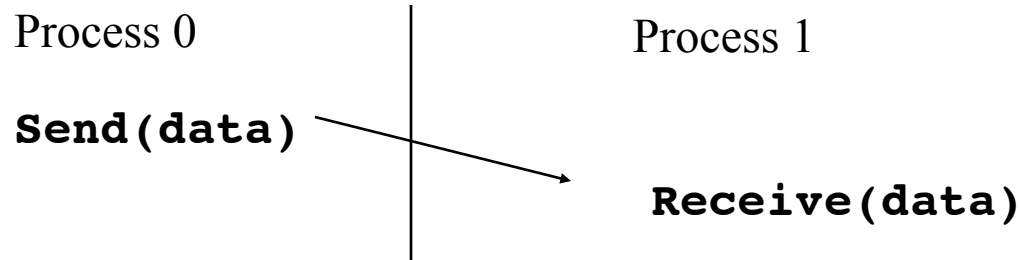
# Better Hello (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

# MPI Basic Send/Receive

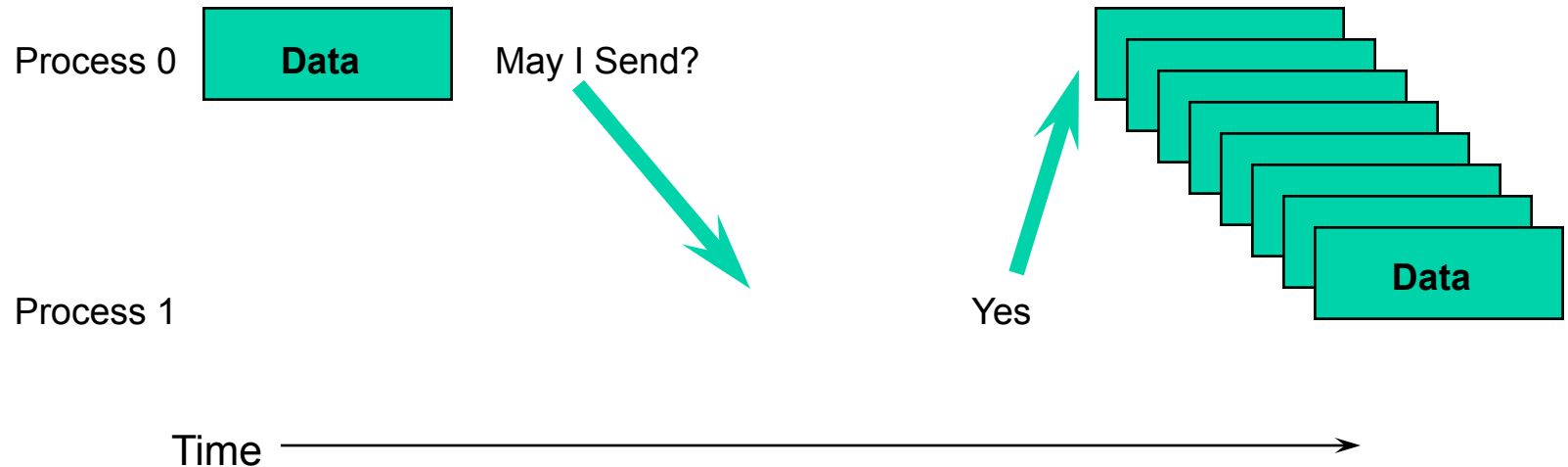
- We need to fill in the details in



- 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?

# What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

# 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`**.

# MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI\_INT, MPI\_DOUBLE\_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

# 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.

# 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.

# 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.



# 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)
```

# Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

# 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
- Use `MPI_Comm_split` to create new communicators

# MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI\_INIT**
  - **MPI\_FINALIZE**
  - **MPI\_COMM\_SIZE**
  - **MPI\_COMM\_RANK**
  - **MPI\_SEND**
  - **MPI\_RECV**
- Point-to-point (send/recv) isn't the only way...

# Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **MPI\_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI\_REDUCE** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

# Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
<b>Send (1)</b>	<b>Send (0)</b>
<b>Recv (1)</b>	<b>Recv (0)</b>

- This is called “unsafe” because it depends on the availability of system buffers

# Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

Process 0

Process 1

---

**Send (1)**

**Recv (0)**

**Recv (1)**

**Send (0)**

- Use non-blocking operations:

Process 0

Process 1

---

**Isend (1)**

**Isend (0)**

**Irecv (1)**

**Irecv (0)**

**Waitall**

**Waitall**

# Toward a Portable MPI Environment

- MPICH is a high-performance portable implementation of MPI (1).
- It runs on MPP's, clusters, and heterogeneous networks of workstations.
- In a wide variety of environments, one can do:  

```
configure  
make  
mpicc -mpitrace myprog.c  
mpirun -np 10 myprog  
upshot myprog.log
```

to build, compile, run, and analyze performance.



# Extending the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/ Fortran-90; interlanguage issues

# When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

# When *not* to use MPI

- Regular computation matches HPF
  - But see PETSc/HPF comparison ([ICASE 97-72](#))
- Solution (e.g., library) already exists
  - <http://www.mcs.anl.gov/mpi/libraries.html>
- Require Fault Tolerance
  - Sockets
- Distributed Computing
  - CORBA, DCOM, etc.

# Summary

- The parallel computing community has cooperated on the development of a standard for message-passing libraries.
- There are many implementations, on nearly all platforms.
- MPI subsets are easy to learn and use.
- Lots of MPI material is available.