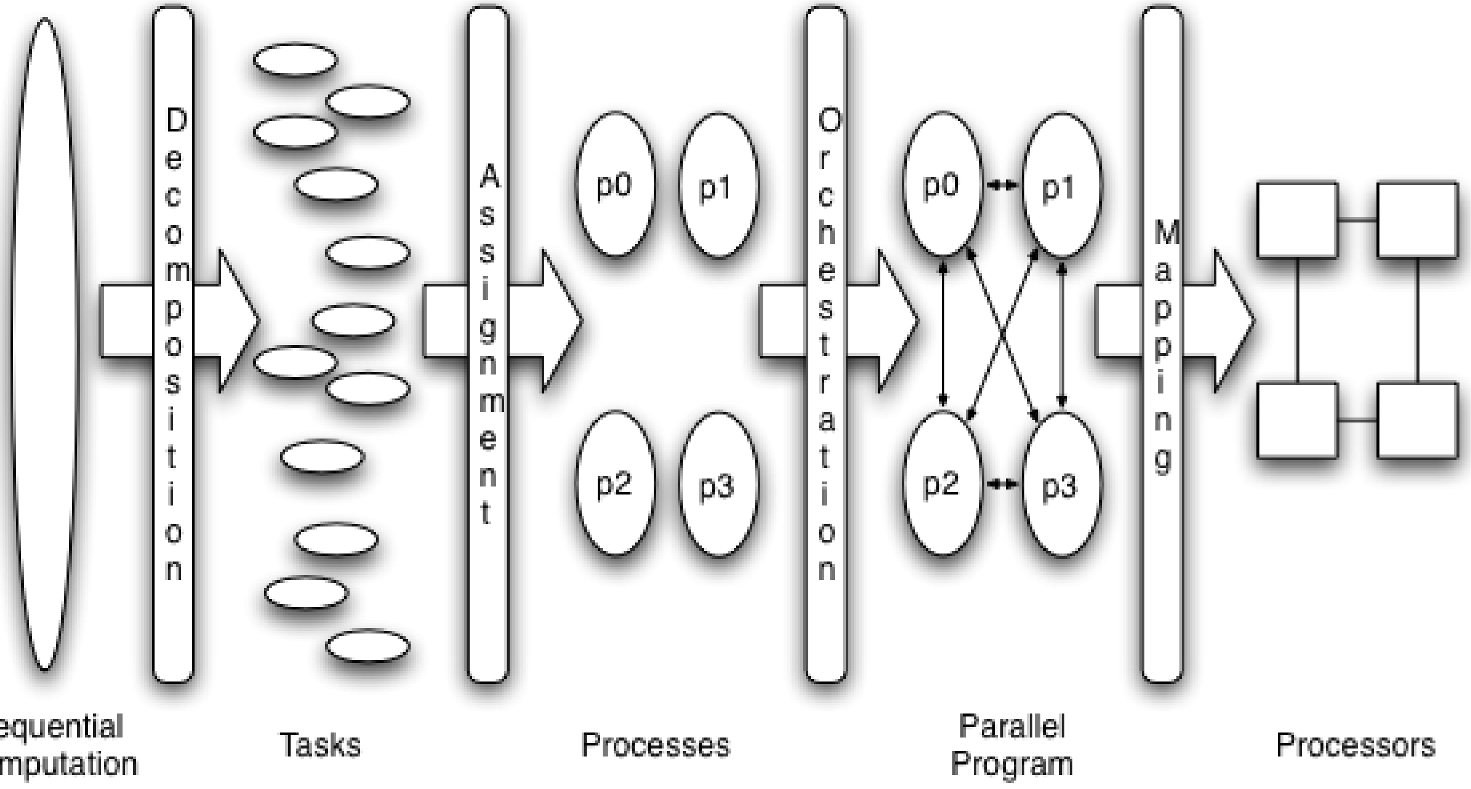
Background

- Parallel computing landscape
- The Message Passing Interface (MPI)
- “Six function MPI”
- Basic collective operations
- Intermediate point to point operations
- Profiling, debugging, tuning
- Where to learn more

Parallel Computing Landscape

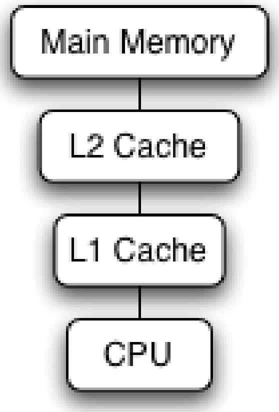
- Communicating and cooperating processes that solve large problems fast
- Parallelization process
- Programming model
    - Describes logical relationship of CPUs, memory, network
- Hardware model
    - Describes physical relationship of CPUs, memory, network
- Logical and physical relationships can be independent

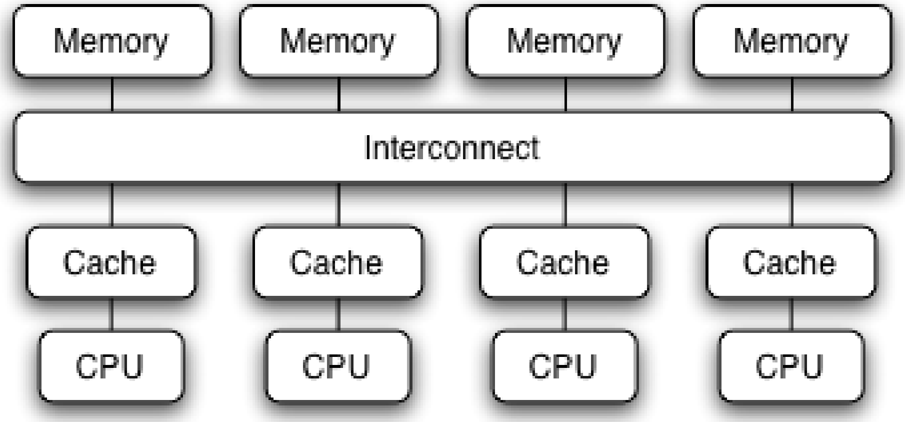
Parallelization Process

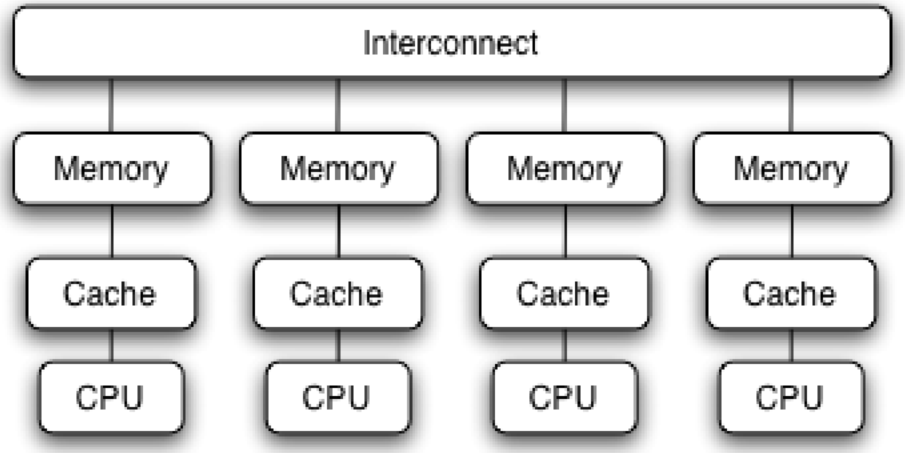


Hardware Models

- Single processor model
- Shared memory
- Distributed memory
- *All memory is distributed*





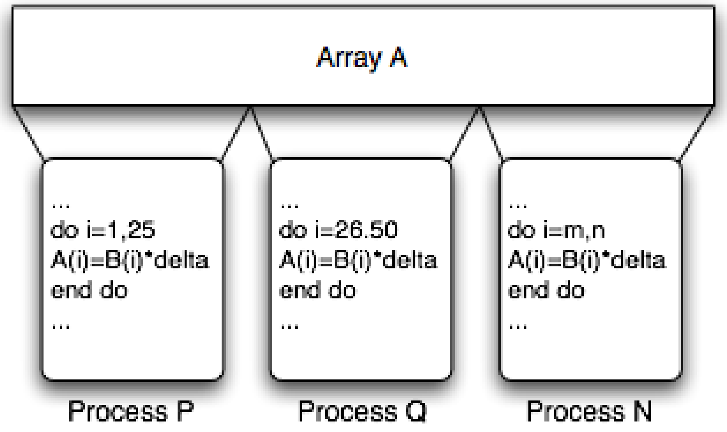


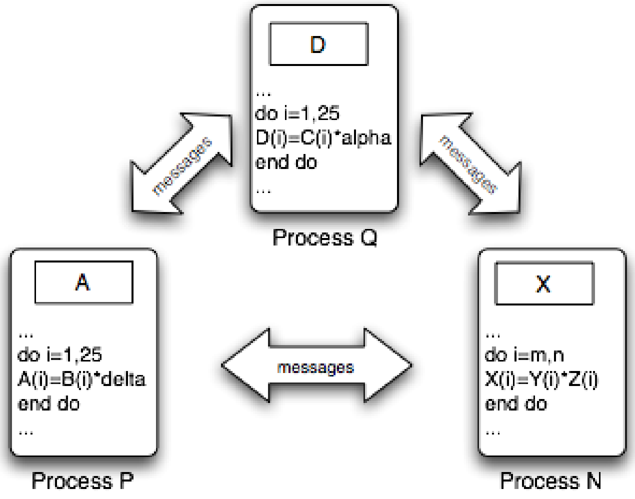
Message Passing is Pervasive

- Easier to build than shared memory
  - Pushes complexity off onto programmers
- World's biggest systems
  - RIKEN K Machine
- Clusters
  - Beowulf
  - Dark systems
- Laptops

Programming Models

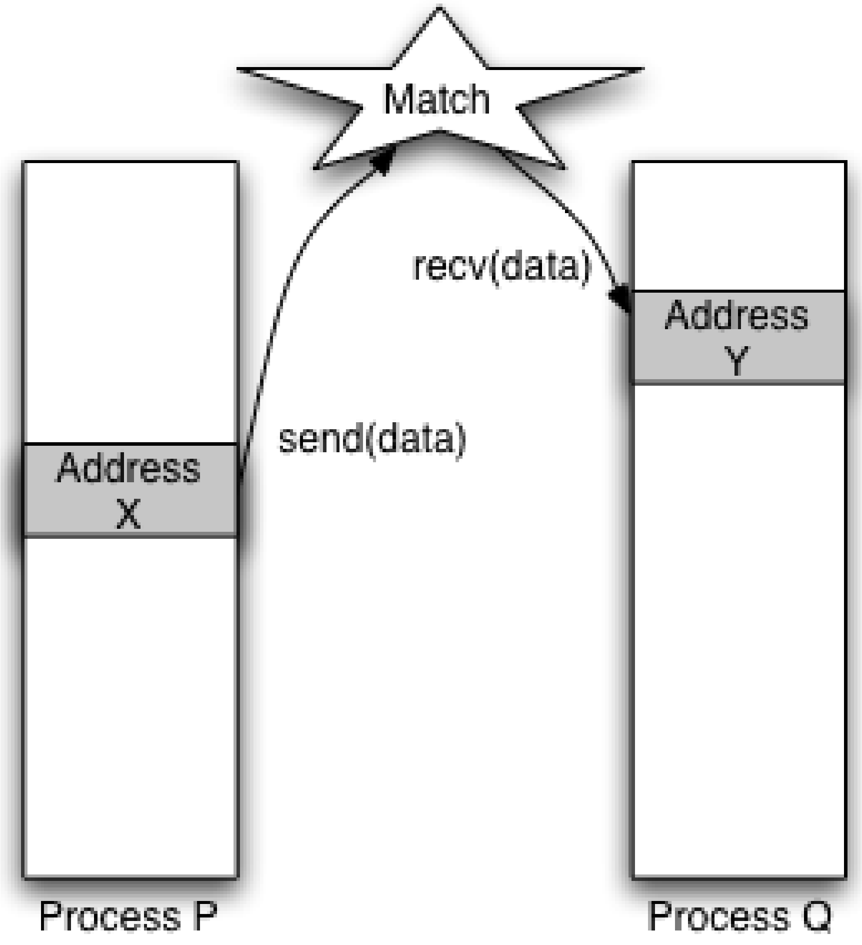
- A programmer-oriented taxonomy
  - **Data-parallel :** Same operations on different data (High-Performance Fortran)
  - **Task-parallel :** Different operations on different data
  - **SPMD :** Single program, multiple data (MPI)
- SPMD and task-parallel are essentially equivalent
  - Any task-parallel program can be expresses as SPMD





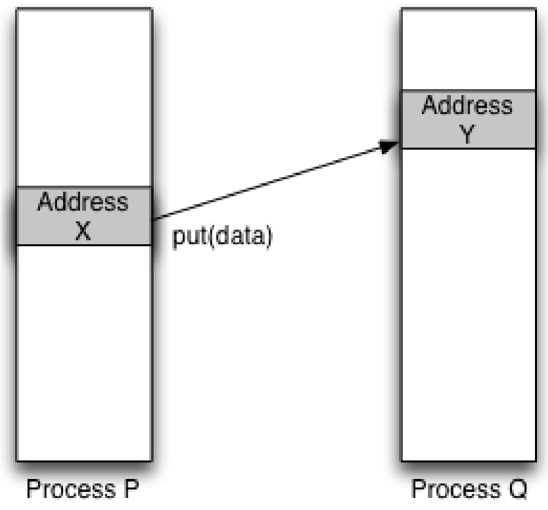
Communicating: Cooperative Operations

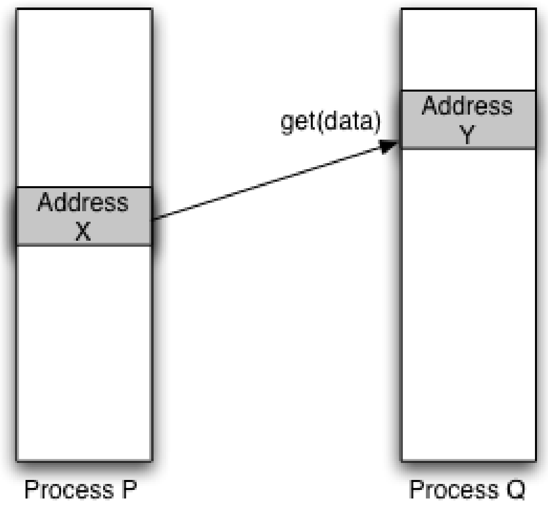
- Message-passing is an approach that makes the exchange of data cooperative
- Data must both be explicitly sent and received
- An advantage is that any change in the *receiver's* memory is made with the receiver's participation



Communicating: One-Sided Operations

- One-sided operations between parallel processes include remote memory reads and writes (gets and puts)
- Advantage: data can be accessed without waiting for another process
- Disadvantage: synchronization may be easy or difficult





Before MPI (circa 1990)

- Message passing well understood as parallel programming paradigm
  - As a model: Tony Hoare's Communicating Sequential Processes
  - Research demonstration: e.g., Caltech Cosmic Cube hypercube
- Early vendor systems were not portable
  - Intel NX
  - Thinking Machine CMMD
  - IBM EUI
- 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

Portability Addressed by Users

- Create own abstract communication calls
  - Minimal common subset of functions
- I had “mydefs.h”

```
#ifdef INTEL
#define MY_SEND ISEND
#endif

#ifdef TMC
#define MY_SEND CMMD_SEND
#endif
```

Enter MPI

- 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 (LAM/MPI, MPICH, vendors)
- MPI-2 Standard (1.2 and 2.0) released July, 1997

What is MPI?

- A *message-passing library specification*
  - Message-passing model
  - Not a compiler specification
  - Not a specific product
- Interfaces for C, C++, and Fortran 77
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Two parts: MPI-1 (1.2) and MPI-2 (2.0)

What is MPI? (cont.)

- Designed to permit (unleash?) the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

MPI Features

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols on some systems), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

MPI Features (cont’d)

- Dynamic process control
  - Allows creation and cooperative termination of processes after an MPI application has started.
  - Mechanism to establish communication between existing MPI applications, which may have been started separately.
- One-sided operations
  - Remote Memory Access (RMA) communication mechanisms
  - Communication: Put (remote write), Get (remote read) and Accumulate (remote update)
  - Support for both active and passive target synchronization.

MPI Features (cont’d)

- Parallel I/O
  - Portable interface for optimized parallel file access
  - Support for synchronous and asynchronous I/O
  - Allows for close coupling with parallel filesystems
  - Data partitioning expressed using derived datatypes.

MPI Features (cont’d)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)
- Profiling
  - Hooks allow users to intercept MPI calls to install their own tools
- Environmental
  - Inquiry
  - Error control

Features Not in MPI

- Non-message-passing concepts not included:
  - Remote memory transfers
  - Active messages
  - Threads
  - Virtual shared memory
  - Fault tolerance
- MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, etc.)

Is MPI Large or Small?

- MPI is large (MPI-1 has 128 functions, MPI-2 adds 152 functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs can be written with just 6 basic functions
- MPI is just right
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it

Where to Use MPI?

- You need a portable parallel program
- You are writing a parallel library
- You have irregular or dynamic data relationships that do not fit a data parallel model
- You care about performance

Where *not* to Use MPI

- You can use HPF or a parallel Fortran 90
- You don’t need parallelism at all
- You can use libraries (which may be written in MPI)
- You need simple threading in a slightly concurrent environment

Six-function MPI

pervasivetechnologylabs
AT INDIANA UNIVERSITY

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may

Simple MPI C Program

```
#include <stdio.h>
#include “mpi.h”

int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
   printf(“Hello world\n”);
   MPI_Finalize();
   return 0;
}
```

Simple MPI C++ Program

```
#include <iostream>
#include “mpi.h”
using namespace std;
int main(int argc, char **argv)
{
  MPI::Init(argc, argv);
  cout << “Hello world << endl;
  MPI::Finalize();
  return 0;
}
```

Simple MPI Fortran Program

```
program main
include ‘mpif.h’
integer ierr

call MPI_INIT(ierr)
print *, ‘Hello world’
call MPI_FINALIZE(ierr)

end
```

Commentary

- `#include “mpi.h”` or `#include “mpif.h”` provides basic MPI definitions and types
- All non-MPI routines are local; thus the `printf()` runs on each process
- *These simple programs assume that all processes can do output. Not all parallel systems provide this feature - MPI provides a way to handle this case.*

Commentary

- Starting MPI

```
int MPI_Init(int *argc, char **argv)

void MPI::Init(int& argc, char**& argv)

MPI_INIT(IERR)
INTEGER IERR
```

Commentary

- Exiting MPI

```
int MPI_Finalize(void)

void MPI::Finalize()

MPI_FINALIZE(IERR)
INTEGER IERR
```

C/C++ and Fortran Language Considerations

- `MPI_INIT`: The C version accepts the `argc` and `argv` variables that are provided as arguments to `main()`
- Error codes: Almost all MPI Fortran subroutines have an integer return code as their last argument. Almost all C functions return an integer error code.
- Bindings
  - **C:** All MPI names have an MPI_ prefix. Defined constants are in all capital letters. Defined types and functions have one capital letter after the prefix; the remaining letters are lowercase.
  - **C++:** All MPI functions and classes are in the MPI namespace, so instead of referring to X and MPI_X as one would in C, one writes MPI::X.

C/C++ and Fortran Language Considerations

- Bindings (cont.)
  - **Fortran:** All MPI names have an MPI_ prefix, and all characters are capitals
- Types:
  - **C:** Opaque objects are given type names
  - **C++:** Opaque objects are objects, defined by a set of MPI classes.
  - **Fortran:** Opaque objects are usually of type INTEGER (exception: binary-valued variables are of type LOGICAL)
- Inter-language interoperability is not guaranteed (e.g. Fortran calling C or vice-versa)
- Mixed language programming is OK as long as only C or Fortran uses MPI

Compiling MPI Programs

- Many MPI implementations have “wrapper” compilers which:
  - Provide all the necessary command-line flags
  - Then invoke a “real” (underlying) compiler
- Examples:
  - mpicc hello.c -o hello-c
  - mpic++ hello.cc -o hello-cxx
  - mpif77 hello.f -o hello-f77

Running MPI Programs

- On many platforms MPI programs can be started with 'mpirun'.

```
mpirun C -w hello
```

- 'mpirun' is not part of the standard, but some version of it is common with several MPI implementations. The version shown here is for the LAM implementation of MPI.
- *Just as Fortran does not specify how Fortran programs are started, MPI does not specify how MPI programs are started.*

Finding Out About the Parallel Environment

- Two of the first questions asked in a parallel program are as follows:
  1. "How many processes are there?"
  2. "Who am I?"
- "How many" is answered with MPI_COMM_SIZE
- "Who am I" is answered with MPI_COMM_RANK.
- The rank is a number between zero and (SIZE -1).

A Second MPI C Program

```
#include <stdio.h>
#include “mpi.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(“Hello world! I am %d of %d\n”,rank,size);
  MPI_Finalize();
  return 0;
}
```

A Second MPI C++ Program

```
#include <iostream>
#include “mpi.h”
using namespace std;
int main(int argc, char **argv)
{
  MPI::Init(argc, argv);
  int rank = MPI::COMM_WORLD.Get_rank();
  int size = MPI::COMM_WORLD.Get_size();
  cout << “Hello world! I am “ << rank << “ of “
       << size << endl;
  MPI::Finalize();
  return 0;
}
```

A Second MPI Fortran Program

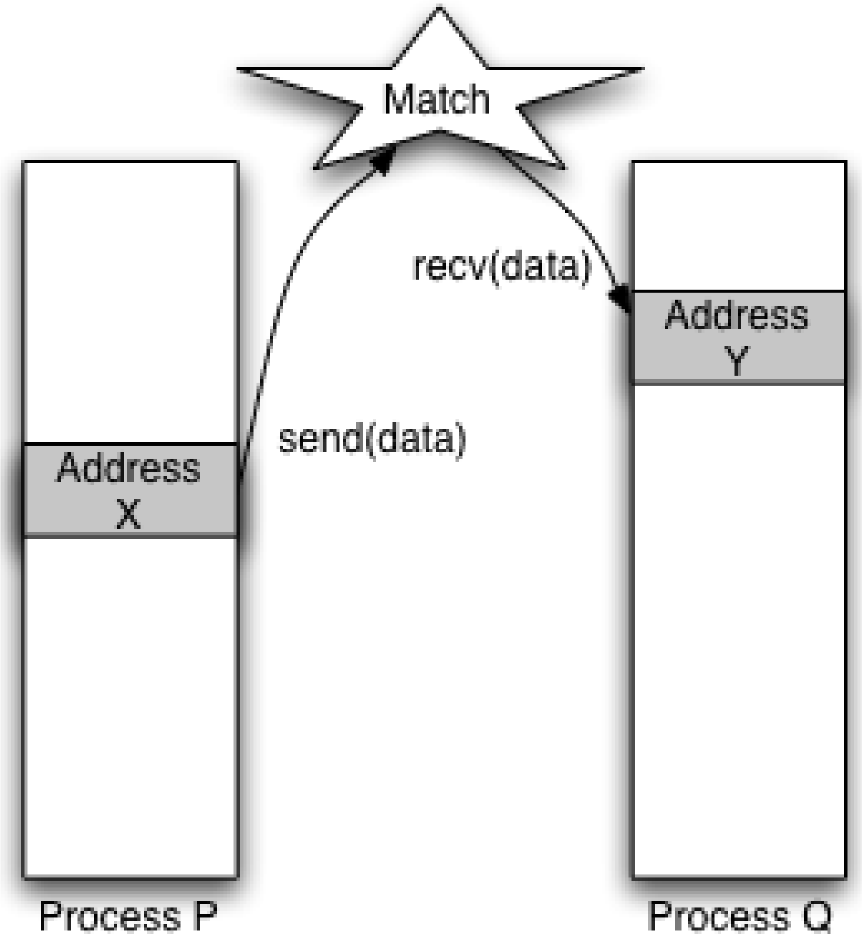
```
program main
include ‘mpif.h’
integer rank, size, ierr

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! I am ‘, rank, ‘ of ‘, size
call MPI_FINALIZE(ierr)

end
```

MPI_COMM_WORLD

- Communication in MPI takes place with respect to *communicators* (more about communicators later)
- The MPI_COMM_WORLD communicator is created when MPI is started and contains all MPI processes
- MPI_COMM_WORLD is a useful default communicator - many applications do not need to use any other

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

- To answer some of these questions, 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 column wise
    - 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 can be combined / nested.
- Specifications of elementary datatypes allow heterogeneous communication.
- Elimination of length in favor of count is clearer.
- Specifying application-oriented layout of data allows maximal use of special hardware.

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 dest, int tag) const;

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
         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 type, int source,
             int tag, MPI_Comm comm,
             MPI_Status *status)

void MPI::Comm::Recv(const void* buf, int count,
                  const MPI::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
```

Six Function MPI

- MPI is very simple. These six functions allow you to write many programs:

  MPI_INIT

  MPI_COMM_SIZE

  MPI_COMM_RANK

  MPI_SEND

  MPI_RECV

  MPI_FINALIZE

Collective Communication

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may

Collective Communications in MPI

- Coordinated communication among a group of processes, as specified by a communicator
- All collective operations are blocking
- All processes in the communicator group must call the collective operation
- Message tags are not used
- Three classes of collective operations:
  - Data movement
  - Collective computation
  - Synchronization

MPI Basic Collective Operations

- Two simple collective operations:

```
MPI_BCAST(start, count, datatype,
          root, comm)
MPI_REDUCE(start, result, count,
           datatype, operation,
           root, comm)
```

- The routine MPI_BCAST sends data from one process to all others.
- The routine MPI_REDUCE combines data from all processes returning the result to a single process.

MPI_BCAST

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer
IN	count	number of entries in buffer
IN	datatype	data type of buffer
IN	root	rank of broadcast root
IN	comm	communicator (handle)

MPI_BCAST Binding

```
int MPI_Bcast(void *buffer, int count,
             MPI_Datatype datatype, int root,
             MPI_Comm comm)

void MPI::Comm::Bcast(const void* buffer, int count,
                  const MPI::Datatype& datatype,
                  int root) const = 0;

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM,
         IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_REDUCE

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer
IN	count	number of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	reduce operation
IN	root	rank of broadcast root
IN	comm	communicator (handle)

MPI_REDUCE Binding

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)

void MPI::Comm::Reduce(const void* sendbuf,
                  void* recvbuf, int count,
                  const MPI::Datatype& datatype,
                  const MPI::Op& op,
                  int root) const = 0;

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
           ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Collective Example

```
#include "mpi.h"

int main ( int argc, char **argv )
{
   int n, i, pool_size, my_rank;
   double mypi, pi, h, sum, x, a;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == ROOT) {
      printf("Enter the number of intervals: ");
      scanf("%d",&n);
      if (n==0) n=100;
   }

   MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
```

Collective Example, Continued

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = my_rank + 1; i <= n; i += pool_size) {
       x = h * ((double)i - 0.5);
       sum += 4.0 / (1.0 + a*a);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, ROOT,
               MPI_COMM_WORLD);

    if (my_rank == ROOT) printf("\npi is approximately %.16f\n", pi);

    MPI_Finalize();

    return 0;
 }
```

Collective Example in Fortran

```
program main
include 'mpif.h'

double precision PIX
parameter (PIX = 4*atan(1.0))

double precision  mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr

f(a) = 4.d0 / (1.d0 + a*a)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
print *, "Process ", myid, " of ", numprocs, " is alive"

if (myid .eq. 0) then
   print *,"Enter the number of intervals: (0 to quit)"
   read(5,*) n
endif

call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

Collective Example in Fortran (2)

```
c check for n > 0
      IF (N.GT.0) THEN
c
c     calculate the interval size
c
        h = 1.0d0 / n
        sum  = 0.0d0

        do 20 i = myid + 1, n, numprocs
             x = h * (dble(i) - 0.5d0)
             sum = sum + f(x)
 20     continue

        mypi = h * sum
c
c     collect all the partial sums
c
        call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION,
     +                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```

Collective Example in Fortran (3)

```
c
c     process 0 prints the result
c
        if (myid .eq. 0) then
             write(6, 97) pi, abs(pi - PIX)
 97          format('  pi is approximately: ', F18.16,
     +              '  Error is: ', F18.16)
        endif

      ENDIF

      call MPI_FINALIZE(ierr)

      stop
      end
```

Another Six Function MPI

- An alternative set of six useful MPI functions:

  MPI_INIT

  MPI_COMM_SIZE

  MPI_COMM_RANK

  MPI_BCAST

  MPI_REDUCE

  MPI_FINALIZE

Synchronization

`MPI_BARRIER(comm)`

Function blocks until all processes in “comm” call it.

```
int MPI_Barrier(MPI_Comm comm)

void Intracomm::Barrier() const

MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```

MPI Collective Routines

- Some other collective routines:

MPI_ALLGATHER	MPI_ALLGATHERV	MPI_ALLREDUCE
MPI_ALLTOALL	MPI_ALLTOALLV	MPI_BCAST
MPI_GATHER	MPI_GATHERV	MPI_REDUCE
MPI_REDUCESCATTER	MPI_SCAN	MPI_SCATTER
MPI_SCATTERV		

- All versions deliver results to *all* participating processes.
- ***V*** versions allow the chunks to have different sizes.
- MPI_ALLREDUCE, MPI_REDUCE, MPI_REDUCESCATTER, and MPI_SCAN take both built-in and user-defined combination functions.

Intermediate Point to Point Communication

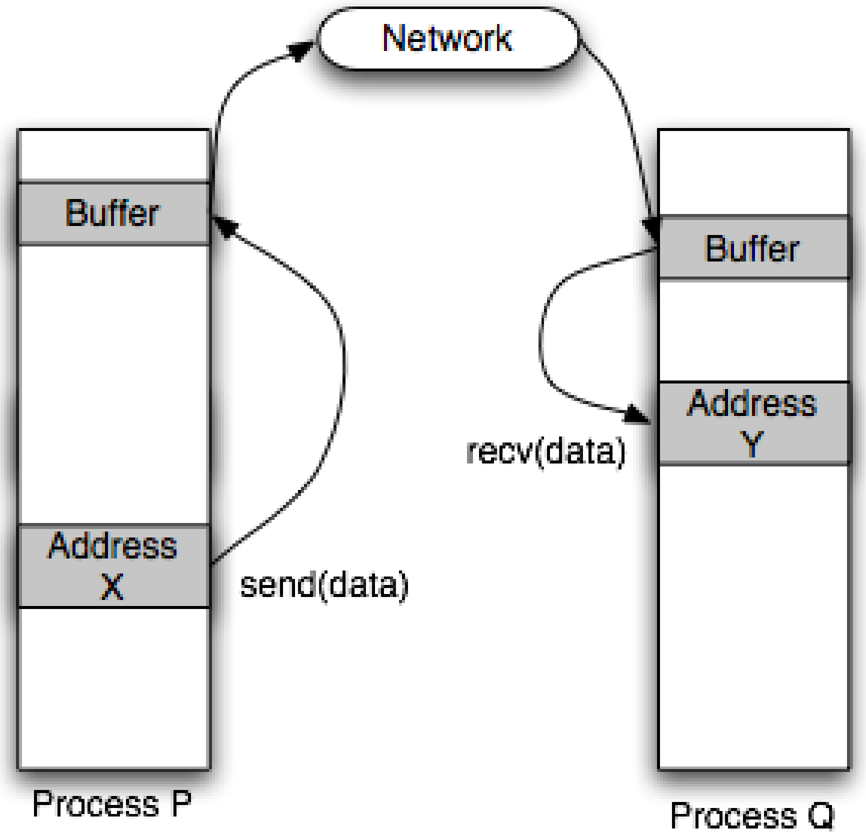
The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may

Non-Blocking Communication

- Overlap communication and computation
- Reduce data motion
- Buffer management
- Deadlock avoidance
- Allocation of (performance related) resources

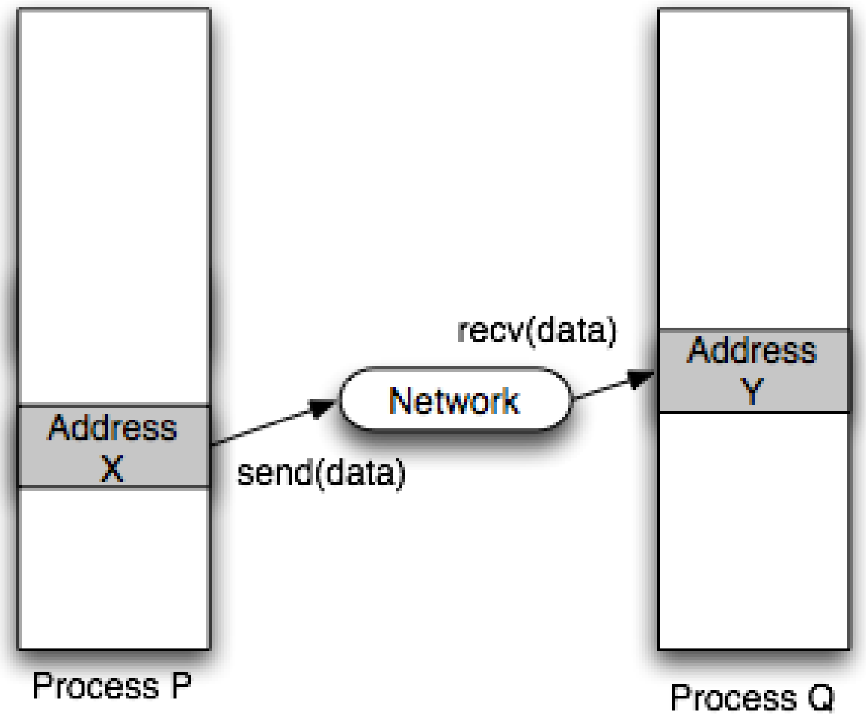
Buffered Communication

- Where does data go when you send it?



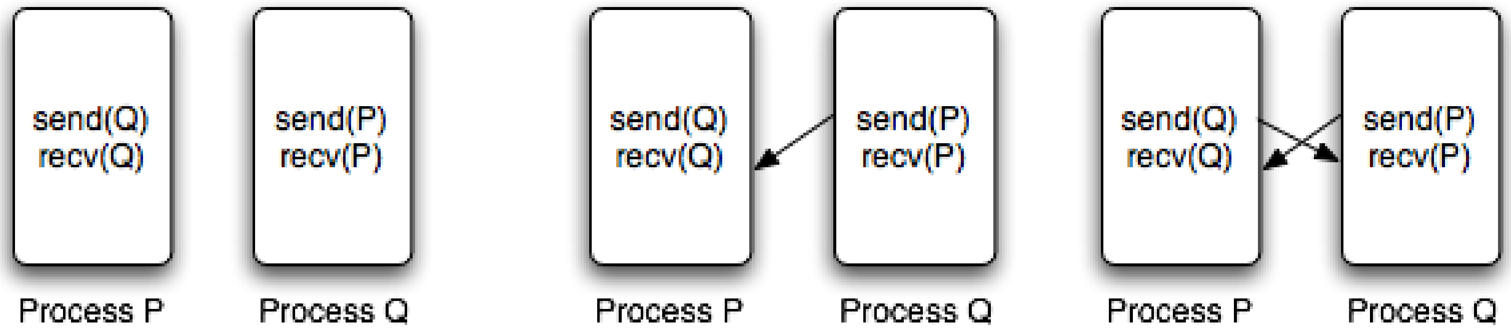
Unbuffered Communication

- Buffering can be avoided
- But we need to make sure it is safe to touch message data
  - Block until it is safe
  - Return before transfer is complete and wait/ test later



Deadlock

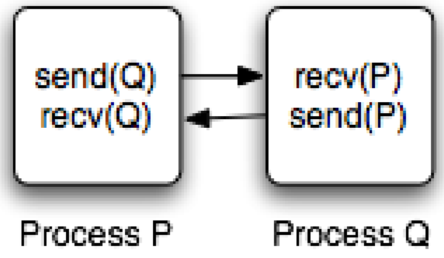
- What happens with following sequence of communication operations?

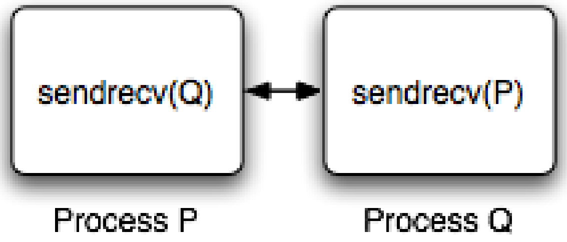


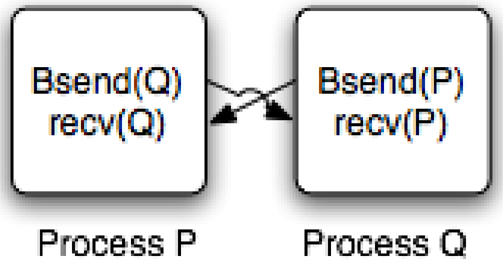
- Behavior depends on availability (and size) of buffering
  - System dependent
  - MPI implementation (LAM, Open MPI, MPICH) have diagnostics for this

Some Solutions

- Order operations properly
  - Difficult to guarantee in practice
- Use simultaneous send and receive
  - “Sendrecv”
- Use own buffers

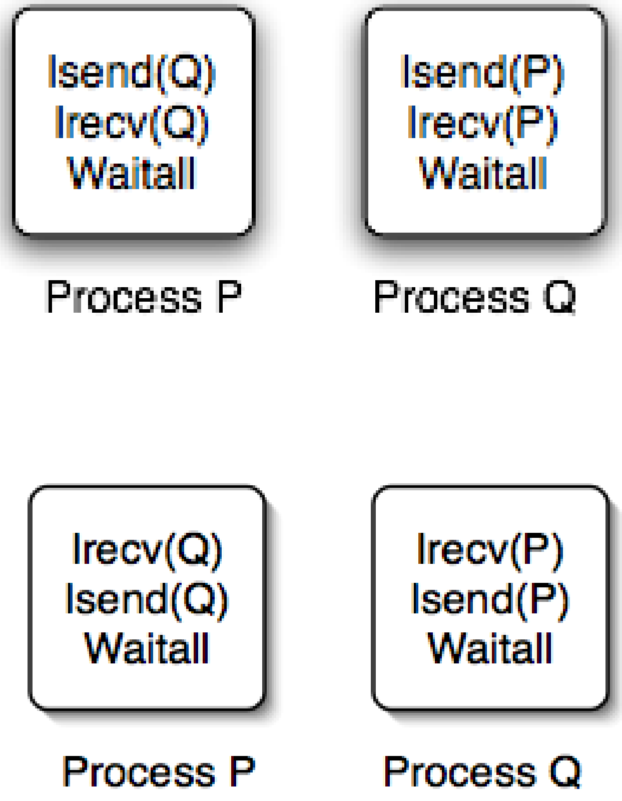






Non-Blocking Operations

- Non-blocking operations (send and receive) return immediately
- Return “request handles” that can be tested or waited on
- Where progress is made (and where communication happens) is implementation specific



MPI Basic Non-Blocking Send

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

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)
OUT	request	communication request (handle)

Bindings for Non-Blocking Send

```
int MPI_Isend(void *buf, int count,
              MPI_Datatype type, int dest,
              int tag, MPI_Comm comm
              MPI_Request *request)

Request MPI::Comm::Isend(const void* buf, int count,
                  const MPI::Datatype& datatype,
                  int dest, int tag) const;

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

MPI Basic Non-Blocking Receive

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

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	request	communication request (handle)

Bindings for Non-Blocking Receive

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

Request MPI::Comm::Irecv(const void* buf, int count,
                  const MPI::Datatype& datatype,
                  int source, int tag) const;

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
         REQUEST, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
        REQUEST, IERR
```

Communication Completion

- MPI_WAIT and MPI_TEST are used to complete a non-blocking communication.
- For a non-blocking *send* operation, indicates that sender is free to update the send buffer.
- For a non-blocking *receive* operation, indicates that the receive buffer contains the received message, and receiver is free to access it

MPI Wait

MPI_WAIT(request, status)

INOUT	request	request (handle)
OUT	status	status object (status)

Bindings for Wait

```
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)

void Request::Wait(Status& status);
void Request::Wait()

MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER  REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI Test

MPI_TEST(request, flag, status)

INOUT	request	communication request (handle)
OUT	flag	true if operation completed (logical)
OUT	status	status object (status)

Bindings for Test

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)

void Request::Test(Status& status);
void Request::Test()

MPI_WAIT(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER  REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Simple C Example

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

int main(int argc, char **argv)
{
  int i, rank, size, src, dest;
  int done = 0;
  const int tag = 4;
  double data[100];
  MPI_Request req;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  dest = size - 1;
  src  = 0;

  if (rank == src) {
      for(i = 0; i < 100; i++)
          data[i] = i;
```

Simple C Example (cont’d)

```
      MPI_Isend(data, 100, MPI_DOUBLE, dest, tag,
                MPI_COMM_WORLD, &req);
      do {
          /* Computation to mask latency */
          MPI_Test(&req, &done, &status);
      } while(!done);
      printf("Message finished sending\n");
  }
  else if (rank == dest) {
      MPI_Irecv(data, 100, MPI_DOUBLE, src, tag,
                MPI_COMM_WORLD, &req);
      /* Computation to mask latency */
      MPI_Wait(&req, &status);
      printf("Message fully received\n");
  }

  MPI_Finalize();
  return 0;
}
```

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.

Simple C Example

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

int main(int argc, char **argv)
{
  int i, rank, size, dest;
  int to, src, from, count, tag;
  int st_count, st_source, st_tag;
  double data[100];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Process %d of %d is alive\n", rank, size);

  dest = size - 1;
  src = 0;

  if (rank == src) {
    to    = dest;
    count = 100;
    tag   = 2001;
    for (i = 0; i < 100; i++)
      data[i] = i;
```

Simple C Example (cont’d)

```
    MPI_Send(data, count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD);
  }
  else if (rank == dest) {
    tag   = MPI_ANY_TAG;
    count = 100;
    from  = MPI_ANY_SOURCE;
    MPI_Recv(data, count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD,
             &status);

    MPI_Get_count(&status, MPI_DOUBLE, &st_count);
    st_source= status.MPI_SOURCE;
    st_tag= status.MPI_TAG;

    printf("Status info: source = %d, tag = %d, count = %d\n",
           st_source, st_tag, st_count);
    printf(" %d received: ", rank);
    for (i = 0; i < st_count; i++)
      printf("%lf ", data[i]);
    printf("\n");
  }

  MPI_Finalize();
  return 0;
}
```

Simple C++ Example

```
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char **argv)
{
  int i, rank, size, dest;
  int to, src, from, count, tag;
  int st_count, st_source, st_tag;
  double data[100];
  MPI::Status status;

  MPI::Init(argc, argv);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();

  cout << "Process " << rank << " of " << size << " is alive" << endl;

  dest = size - 1;
  src = 0;

  if (rank == src) {
    to    = dest;
    count = 100;
    tag   = 2001;
    for (i = 0; i < 100; i++)
      data[i] = i;
```

Simple C++ Example (cont’d)

```
    MPI::COMM_WORLD.Send(data, count, MPI::DOUBLE, to, tag);
  }
  else if (rank == dest) {
    tag   = MPI::ANY_TAG;
    count = 100;
    from  = MPI::ANY_SOURCE;
    MPI::COMM_WORLD.Recv(data, count, MPI::DOUBLE, from, tag, status);
    st_count = status.Get_count(MPI::DOUBLE);
    st_source= status.Get_source();
    st_tag= status.Get_tag();

    cout << "Status info: source = " << st_source << ", tag = " << st_tag
         << ", count = " << st_count << endl;
    cout << rank << " received: ";
    for (i = 0; i < st_count; i++)
      cout << data[i] << " ";
    cout << endl;
  }

  MPI::Finalize();
  return 0;
}
```

Simple Fortran Example

```
      program main
      include 'mpif.h'

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(100)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src = 0
C
      if (rank .eq. src) then
         to     = dest
         count  = 100
         tag    = 2001
         do 10 i=1, 100
 10         data(i) = i
```

Simple Fortran Example (cont’d)

```
         call MPI_SEND(data, count, MPI_DOUBLE_PRECISION, to,
     +                   tag, MPI_COMM_WORLD, ierr)
      else if (rank .eq. dest) then
         tag   = MPI_ANY_TAG
         count = 100
         from  = MPI_ANY_SOURCE
         call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +                  tag, MPI_COMM_WORLD, status, ierr)
         call MPI_GET_COUNT(status, MPI_DOUBLE_PRECISION,
     +                       st_count, ierr)
         st_source = status(MPI_SOURCE)
         st_tag    = status(MPI_TAG)
C
         print *, 'Status info: source = ', st_source,
     +             ' tag = ', st_tag, ' count = ', st_count
         print *, rank, ' received', (data(i),i=1,100)
      endif

      call MPI_FINALIZE(ierr)
      end
```

Profiling, Debugging, Tuning

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may

Application Profiling

- With most MPI implementations, serial profilers (gprof, oprofile, etc.) can be used to examine single process performance
- Number of tools for examining communication performance
  - MPE from Argonne National Laboratory
  - Intel Trace Analyzer (formerly Vampir)
- MPI provides a profiling layer for writing portable custom communication profiling tools

Debugging

- Number of commercial debuggers available for MPI:
  - Etnus TotalView
  - Absoft FX2
  - Portland Group PGDBG
- Few free parallel debuggers
- On many MPI implementations, try:

```
mpirun -np X xterm -e gdb application
```

Debugging Hints

- Test early, test often
  - Parallel programs are hard to debug
  - Complex parallel programs are even harder to debug
- When testing, use MPI_SSEND instead of MPI_SEND
  - The semantics of MPI_SEND specify safety of buffer re-use - nothing about message delivery
- stdio is not synchronized between processes

Where to Find More

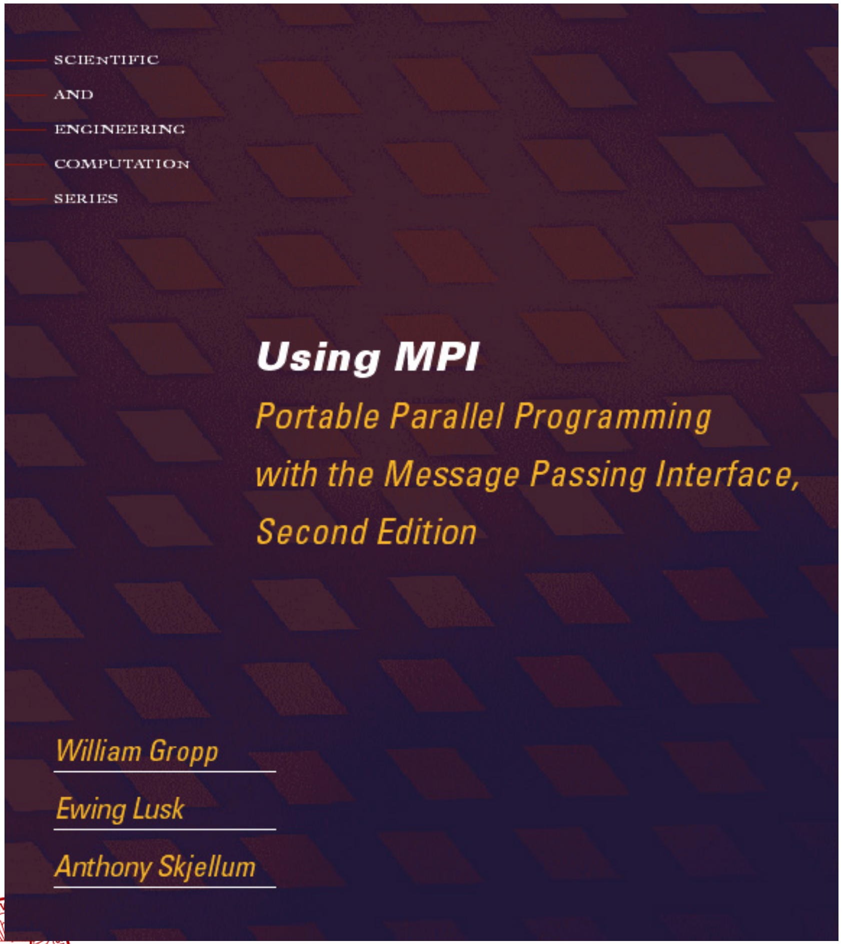
pervasivetechnologylabs
AT INDIANA UNIVERSITY

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may

The Standard

- At http://www.mpi-forum.org
  - All MPI official documents, in both postscript and HTML
  - Pointers to resources including:
    - Other talks and tutorials
    - FAQ
    - Other MPI web sites

Tutorial Material on MPI, MPI-2


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

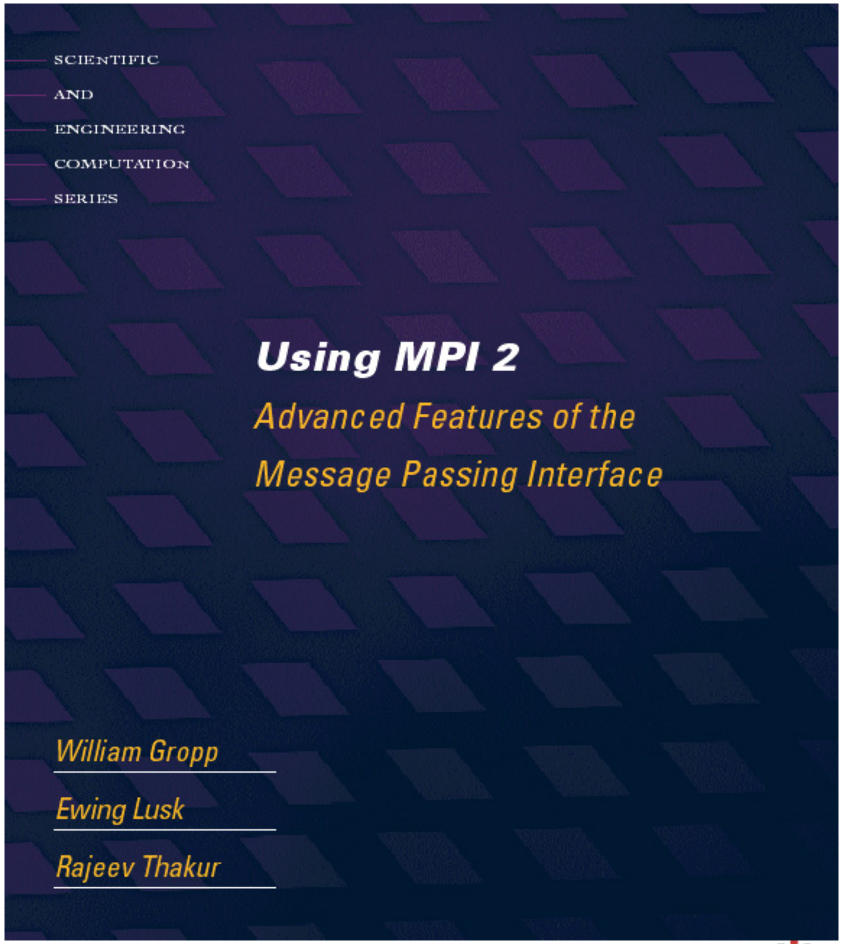
Using MPI

Portable Parallel Programming with the Message Passing Interface, Second Edition

William Gropp

Ewing Lusk

Anthony Skjellum


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

Using MPI 2

Advanced Features of the Message Passing Interface

William Gropp

Ewing Lusk

Rajeev Thakur

http://www.mcs.anl.gov/mpi/{usingmpi,usingmpi2}

More Tutorials / Information

- LAM/MPI web site
  - http://www.lam-mpi.org/tutorials/
- NCSA
  - http://webct.ncsa.uiuc.edu:8900/public/MPI/
- Old *Cluster World* "MPI Mechanic" columns
  - http://cw.squyres.com/

Books

- Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 2e, 1999
- Using MPI-2, by Gropp, Lusk, and Thakur, MIT Press, 1999
- MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - Second edition 1998
- MPI—The Complete Reference: Volume 2, The MPI Extensions, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
- Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.

Implementations

- There are many MPI implementations available from a wide variety of sources, from hardware vendors to academic institutions.
- Some popular open source implementations include:
  - Open MPI
    - http://www.open-mpi.org/
  - LAM/MPI
    - http://www.lam-mpi.org/
  - MPICH
    - http://www-unix.mcs.anl.gov/mpi/mpich/

Portable Parallel Libraries

- The idea that MPI would encourage parallel libraries turned out to be right
- Library sampler
  (From http://www.mcs.anl.gov/mpi/libraries.html)
  - **PETSc:** sparse linear systems, nonlinear equations from PDE’s, unconstrained optimization
  - **FFTW:** FFTs
  - **PGAPack:** general-purpose genetic algorithm library
  - **ScaLAPack:** parallel dense linear algebra
  - **MSG:** structured grids in Fortran
  - **PLAPack:** dense linear algebra
  - **Parallel Level 3 BLAS:** parallel basic linear algebra subroutines
  - **Aztec:** solving linear systems with Krylov methods
  - **MatheMatrix** (commercial): solving linear systems

On the Web

- HPF
  - http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm
- gprof
  - http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html
- oprofile
  - http://oprofile.sourceforge.net/
- MPE
  - http://www-unix.mcs.anl.gov/mpi/www/
- Intel Trace Analyzer
  - http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm
- TotalView
  - http://www.etnus.com/
- Absoft FX2
  - http://www.absoft.com/Products/Debuggers/fx2/fx2_debugger.html
- Portland Group PGDBG
  - http://www.pgroup.com/products/pgdbg.htm

Thanks

- Jeff Squyres
- Josh Hursey
- Brian Barrett
- Bill Gropp

INDIANENSIS UNIVERSITATIS SIGILLUM
LUX ET VERITAS
MDCCCXX

pervasivetechnologylabs
AT INDIANA UNIVERSITY