Parallel Programming With MPI

William Gropp

Argonne National Laboratory
Mathematics and
Computer Science Division

Overview

- Introduction to MPI
  - What it is
  - Where it came from
  - Basic MPI communication
    - Some simple examples
  - More advanced MPI communication
  - A non-trivial exercise
  - Looking to the future: some features from MPI-2
- Building programs using MPI libraries
  - PETSc
    - Poisson solver with *no* MPI
  - pnetCDF
    - High performance parallel I/O

Models for Parallel Computation

- Shared memory (load, store, lock, unlock)
- Message Passing (send, receive, broadcast, ...)
- Transparent (compiler works magic)
- Directive-based (compiler needs help)
- Others (BSP, OpenMP, ...)
- Task farming (scientific term for large transaction processing)

Why Yet Another Parallel Programming Approach?

- Distributed memory (shared nothing) systems
  - Common, easier to build, dominate high-end computing (over 329 of top 500; all 1998 Gordon Bell Prize finalists; most highest-performing applications)
- Performance depends on managing memory use
  - Goal of many parallel programming models is to simplify programming by hiding details of memory locality and management (parallel programming for the masses)
- Support for modular programming

Message Passing Features

- Parallel programs consist of separate processes, each with its own address space
  - Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
  - Programmer manages memory motion
- Collective operations
  - On arbitrary set of processes
- Data distribution
  - Also managed by programmer
    - Message passing model doesn't get in the way
    - It doesn't help either

Types of Parallel Computing Models

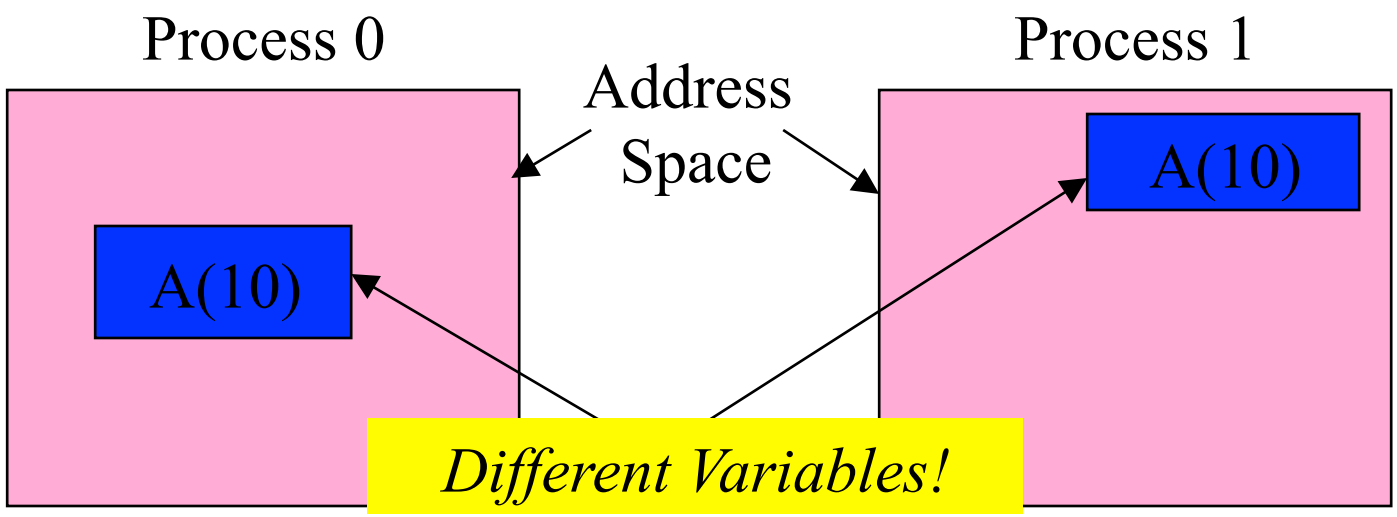
- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program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

Comparison with Other Models

- Single process (address space) model
  - OpenMP and threads in general
  - Fortran 90/95 and compiler-discovered parallelism
  - System manages memory and (usually) thread scheduling
  - Named variables refer to the *same* storage
- Single name space model
  - HPF
  - Data distribution part of the language, but programs still written as if there is a single name space

The Distributed Memory or "Shared-Nothing" Model



- Integer A(10)

  …

  print *, A

- Integer A(10)

  do i=1,10

  A(i) = i

  enddo

  …

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
- Message passing 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

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, and tool developers

Where Did MPI Come From?

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by:
  - vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - portability library writers: PVM, p4
  - users: application scientists and library writers
  - finished in 18 months

Novel Features of MPI

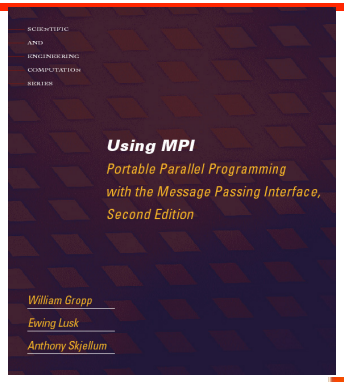
- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

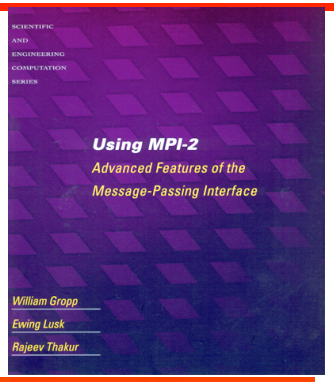
MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- 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

Books on MPI

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 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.





Programming With MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
    - MPI module for Fortran 90 (optional)
- First Program:
  - Create 4 processes in a simple MPI job
  - Write out process number
  - Write out some variables (illustrate separate name space)

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

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;
}
```

Hello (Fortran)

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

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Hello (C++)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size <<
            "\n";
    MPI::Finalize();
    return 0;
}
```

Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- I/O not part of MPI-1
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
    - A consequence of the requirement that non-MPI statements execute independently

Running MPI Programs

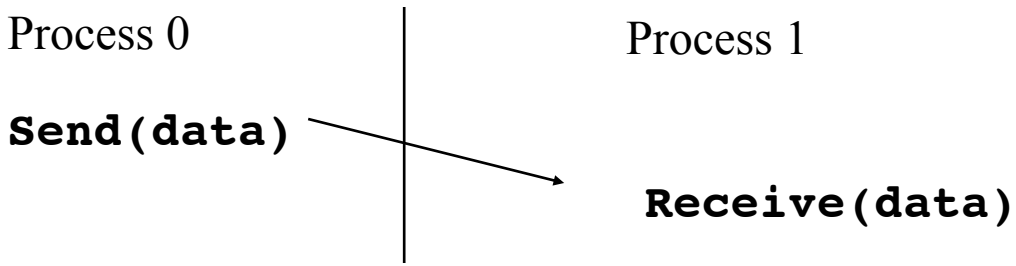
- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
  - Many implementations provided

    `mpirun -np 4 a.out`

    to run an MPI 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.
- Many parallel systems use a *batch* environment to share resources among users
  - The specific commands to run a program on a parallel system are defined by the environment installed on the parallel computer

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?

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- 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 send or receive 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)
  - 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, in particular ones for subarrays

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.

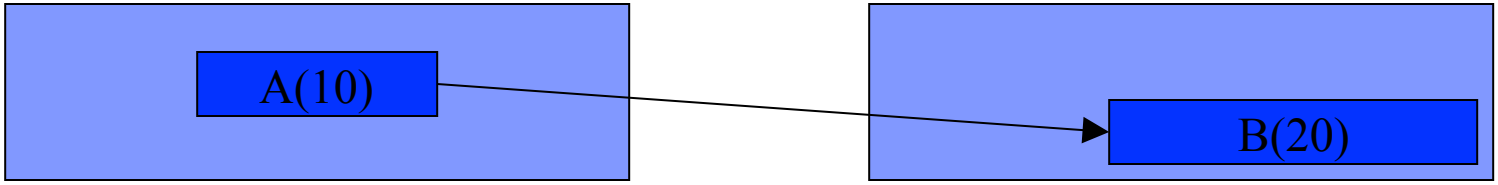
MPI Basic (Blocking) Receive

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

- Waits until a matching (both **`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`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Send-Receive Summary

- Send to matching Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

- Datatype
  - Basic for heterogeneity
  - Derived for non-contiguous
- Contexts
  - Message safety for libraries
- Buffering
  - Robustness and correctness

A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( "Received %d\n", buf );
  }

  MPI_Finalize();
  return 0;
}
```

A Simple MPI Program (Fortran)

```
      program main
      include 'mpif.h'
      integer rank, buf, ierr, status(MPI_STATUS_SIZE)

      call MPI_Init(ierr)
      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
C Process 0 sends and Process 1 receives
      if (rank .eq. 0) then
         buf = 123456
         call MPI_Send( buf, 1, MPI_INTEGER, 1, 0,
     *                  MPI_COMM_WORLD, ierr )
      else if (rank .eq. 1) then
         call MPI_Recv( buf, 1, MPI_INTEGER, 0, 0,
     *                  MPI_COMM_WORLD, status, ierr )
         print *, "Received ", buf
      endif
      call MPI_Finalize(ierr)
      end
```

A Simple MPI Program (C++)

```
#include “mpi.h”
#include <iostream>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI::Init(argv, argc);
  rank = MPI::COMM_WORLD.Get_rank();

  // Process 0 sends and Process 1 receives
  if (rank == 0) {
    buf = 123456;
    MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
  }
  else if (rank == 1) {
    MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
    std::cout << “Received “ << buf << “\n”;
  }

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

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)
```

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;
Comm.Recv(..., MPI::ANY_SOURCE, MPI::ANY_TAG, ...,
          status )
recvd_tag   = status.Get_tag();
recvd_from  = status.Get_source();
recvd_count = status.Get_count( datatype );
```

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

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.
- Use

  mpirun –np # -nolocal a.out

  for your clusters, e.g.

  mpirun –np 3 –nolocal cpi

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**

Another Approach to Parallelism

- *Collective* routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations…

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

Example: PI in C - 1

```
#include "mpi.h"
#include <math.h>
  #include <stdio.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)  {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
```

Example: PI in C - 2

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is .16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

Example: PI in Fortran - 1

```
      program main
      include 'mpif.h'
      integer done, n, myid, numprocs, i, rc
      double pi25dt, mypi, pi, h, sum, x, z
      data done/.false./
      data PI25DT/3.141592653589793238462643/
      call MPI_Init(ierr)
      call MPI_Comm_size(MPI_COMM_WORLD,numprocs, ierr )
      call MPI_Comm_rank(MPI_COMM_WORLD,myid, ierr)
      do while (.not. done)
        if (myid .eq. 0) then
         print *,"Enter the number of intervals: (0 quits)"
         read *, n
        endif
        call MPI_Bcast(n, 1, MPI_INTEGER, 0,
     *                 MPI_COMM_WORLD, ierr )
        if (n .eq. 0) goto 10
```

Example: PI in Fortran - 2

```
      h   = 1.0 / n
      sum = 0.0
      do i=myid+1,n,numprocs
        x = h * (i - 0.5)
        sum += 4.0 / (1.0 + x*x)
      enddo
      mypi = h * sum
      call MPI_Reduce(mypi, pi, 1, MPI_DOUBLE_PRECISION,
     *                MPI_SUM, 0, MPI_COMM_WORLD, ierr )
      if (myid .eq. 0) then
          print *, "pi is approximately “, pi,
     *       “, Error is “, abs(pi - PI25DT)
    enddo
 10 continue
      call MPI_Finalize( ierr )
    end
```

Example: PI in C++ - 1

```
#include "mpi.h"
#include <math.h>
#include <iostream>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI::Init(argc, argv);
  numprocs = MPI::COMM_WORLD.Get_size();
  myid     = MPI::COMM_WORLD.Get_rank();
  while (!done)  {
    if (myid == 0) {
      std::cout << "Enter the number of intervals: (0 quits) ";
      std::cin >> n;;
    }
    MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0 );
    if (n == 0) break;
```

Example: PI in C++ - 2

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,
                           MPI::SUM, 0);
    if (myid == 0)
      std::cout << "pi is approximately " << pi <<
            ", Error is " << fabs(pi - PI25DT) << "\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
  - 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.

Alternative Set of 6 Functions

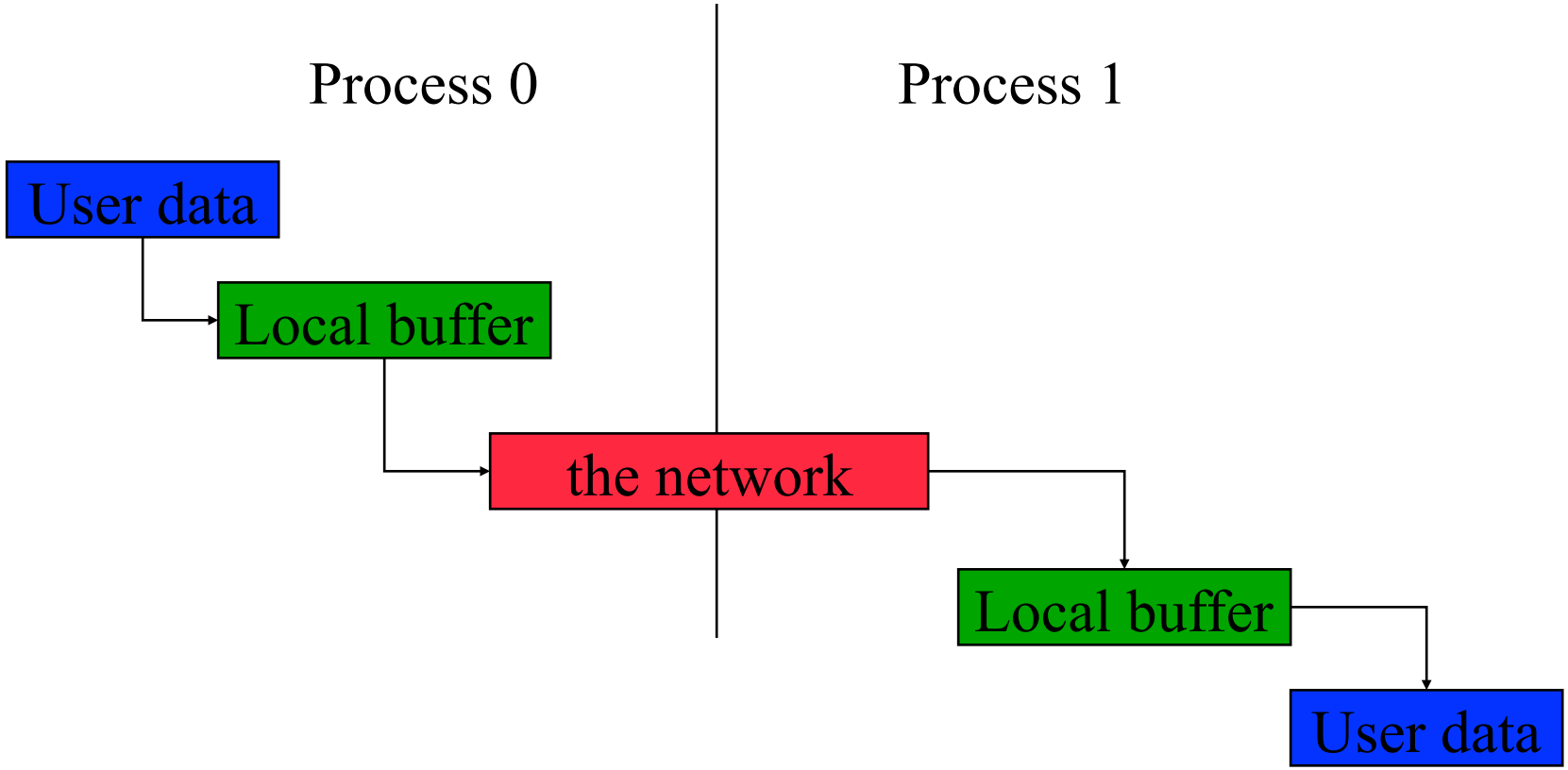
- Using collectives:
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_BCAST**
  - **MPI_REDUCE**

More on Message Passing

- Message passing is a simple programming model, but there are some special issues
  - Buffering and deadlock
  - Deterministic execution
  - Performance

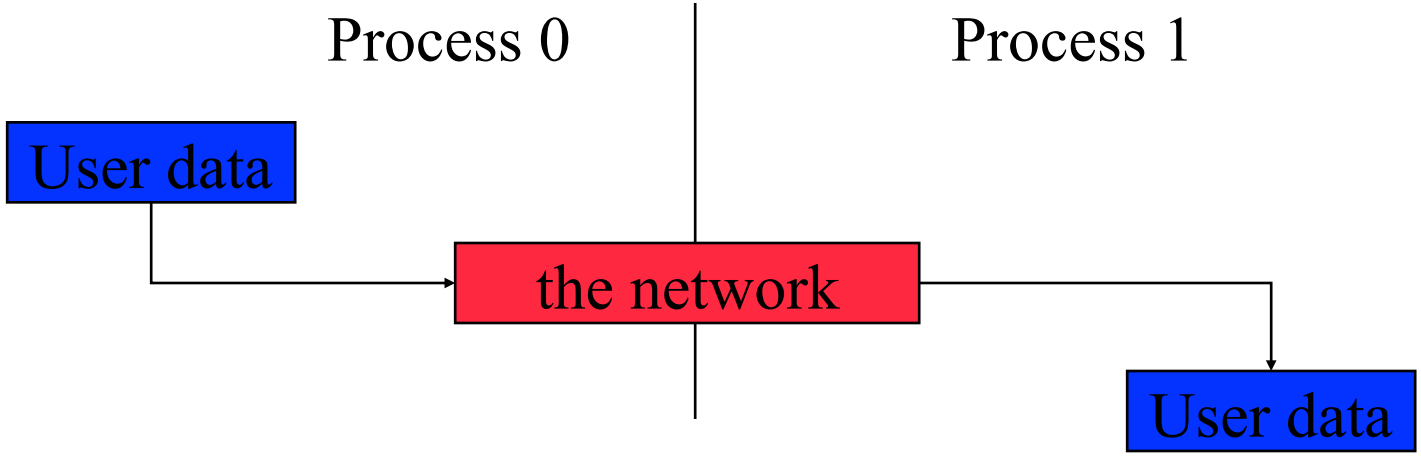
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

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 this code?

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This is called "unsafe" because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	**`Recv(0)`**
`Recv(1)`	**`Send(0)`**

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	**`Sendrecv(0)`**

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	**`Bsend(0)`**
`Recv(1)`	**`Recv(0)`**

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI_Request request;
MPI_Status status;
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);
MPI_Wait(&request, &status);
```

(each request must be Waited on)

- One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

MPI's Non-blocking Operations (Fortran)

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
integer request
integer status(MPI_STATUS_SIZE)
call MPI_Isend(start, count, datatype,
    dest, tag, comm, request,ierr)
call MPI_Irecv(start, count, datatype,
    dest, tag, comm, request, ierr)
call MPI_Wait(request, status, ierr)
```

(Each request must be waited on)

- One can also test without waiting:

```
call MPI_Test(request, flag, status, ierr)
```

MPI's Non-blocking Operations (C++)

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI::Request request;
MPI::Status  status;

request = comm.Isend(start, count,
                     datatype, dest, tag);

request = comm.Irecv(start, count,
                     datatype, dest, tag);

request.Wait(status);
```

(each request must be Waited on)

- One can also test without waiting:

```
flag = request.Test( status );
```

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
  array_of_statuses)
MPI_Waitany(count, array_of_requests,
  &index, &status)
MPI_Waitsome(count, array_of_requests,
  array_of indices, array_of_statuses)
```

- There are corresponding versions of **`test`** for each of these.

Multiple Completions (Fortran)

- It is sometimes desirable to wait on multiple requests:

```
call MPI_Waitall(count, array_of_requests,
  array_of_statuses, ierr)
call MPI_Waitany(count, array_of_requests,
  index, status, ierr)
call MPI_Waitsome(count, array_of_requests,
  array_of indices, array_of_statuses, ierr)
```

- There are corresponding versions of **test** for each of these.

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in any mode.

Other Point-to Point Features

- **`MPI_Sendrecv`**
- **`MPI_Sendrecv_replace`**
- **`MPI_Cancel`**
  - Useful for multibuffering
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance

MPI_Sendrecv

- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
  - More general than "send left"

Process 0	Process 1
`SendRecv(1)`	**`SendRecv(0)`**

MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

Synchronization

- `MPI_Barrier( comm )`
- Blocks until all processes in the group of the communicator `comm` call it.
- Almost never required in a parallel program
  - Occasionally useful in measuring performance and load balancing

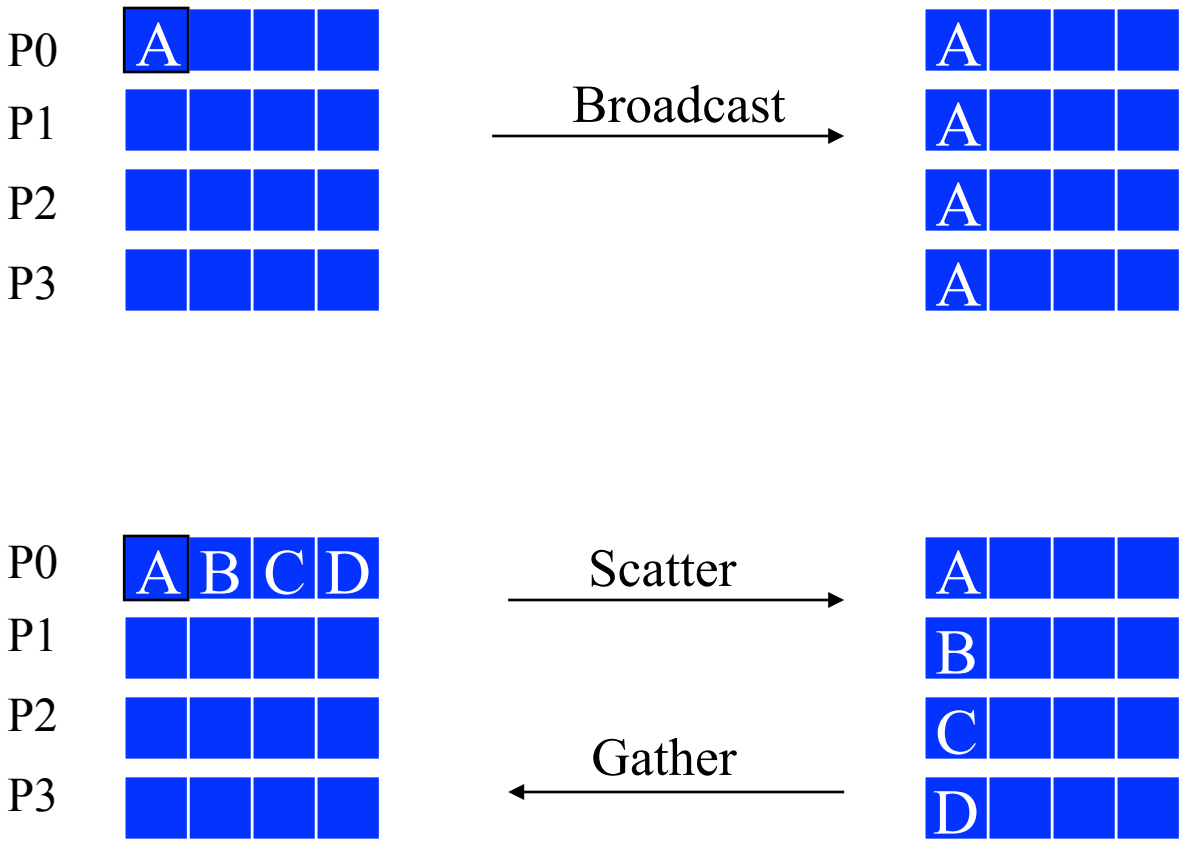
Synchronization (Fortran)

- `MPI_Barrier( comm, ierr )`
- Blocks until all processes in the group of the communicator `comm` call it.

Synchronization (C++)

- `comm.Barrier();`
- Blocks until all processes in the group of the communicator `comm` call it.

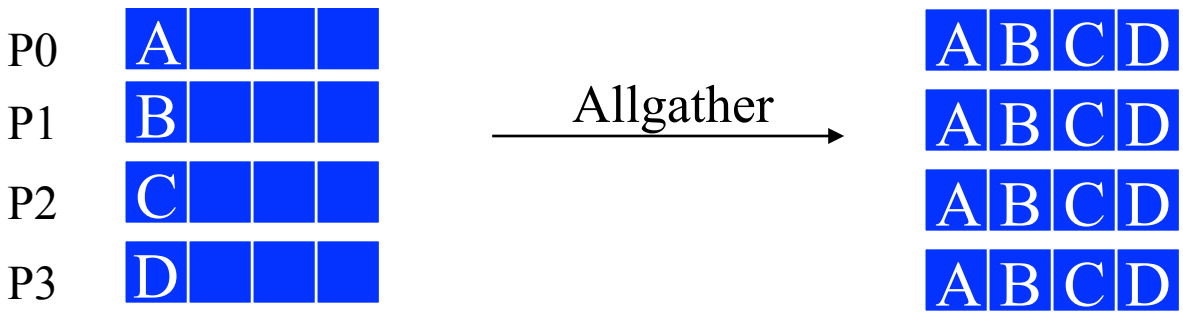
Collective Data Movement

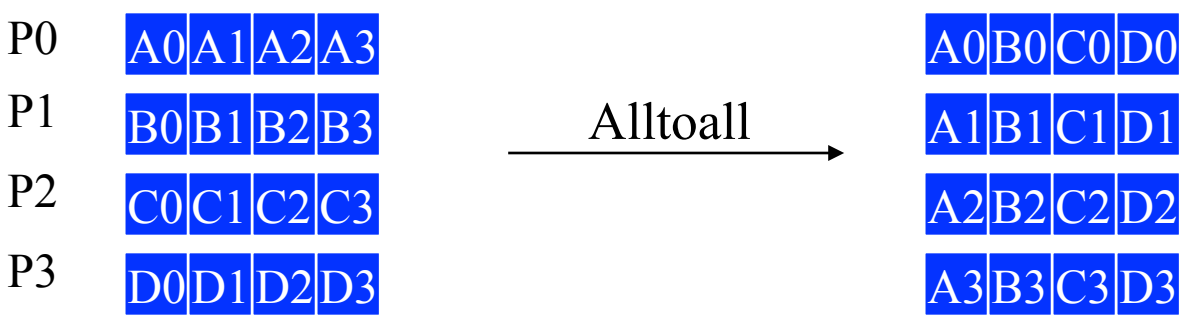


Comments on Broadcast

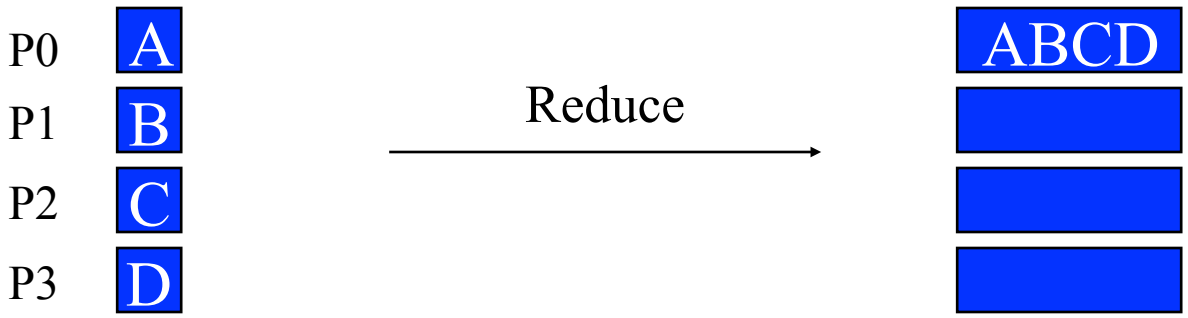
- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a "multi-send"
  - "root" argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive
- Example of orthogonallity of the MPI design: MPI_Recv need not test for "multisend"

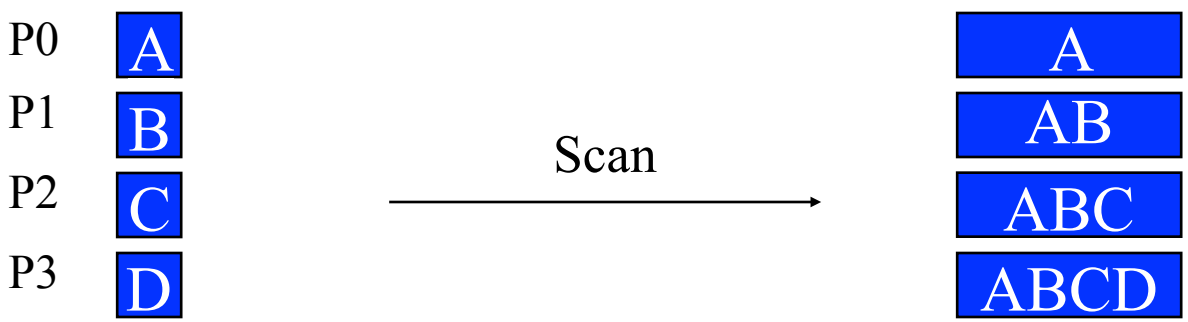
More Collective Data Movement





Collective Computation





MPI Collective Routines

- Many Routines: **`Allgather`**, **`Allgatherv`**, **`Allreduce`**, **`Alltoall`**, **`Alltoallv`**, **`Bcast`**, **`Gather`**, **`Gatherv`**, **`Reduce`**, **`Reduce_scatter`**, **`Scan`**, **`Scatter`**, **`Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.
- MPI-2 adds **`Alltoallw`**, **`Exscan`**, intercommunicator versions of most routines

MPI Built-in Collective Computation Operations

• **MPI_MAX**	Maximum
• **MPI_MIN**	Minimum
• **MPI_PROD**	Product
• **MPI_SUM**	Sum
• **MPI_LAND**	Logical and
• **MPI_LOR**	Logical or
• **MPI_LXOR**	Logical exclusive or
• **MPI_BAND**	Binary and
• **MPI_BOR**	Binary or
• **MPI_BXOR**	Binary exclusive or
• **MPI_MAXLOC**	Maximum and location
• **MPI_MINLOC**	Minimum and location

The Collective Programming Model

- One style of higher level programming is to use *only* collective routines
- Provides a "data parallel" style of programming
  - Easy to follow program flow

What MPI Functions are in Use?

- For simple applications, these are common:
  - Point-to-point communication
    - MPI_Irecv, MPI_Isend, MPI_Wait, MPI_Send, MPI_Recv
  - Startup
    - MPI_Init, MPI_Finalize
  - Information on the processes
    - MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name
  - Collective communication
    - MPI_Allreduce, MPI_Bcast, MPI_Allgather

Understanding and Predicting Performance

- Not all programs will run faster in parallel
  - The benefit of additional processors may be outweighed by the cost of moving data between them
- A typical cost model is

$$T = \frac{T_p}{p} + T_s + T_c$$

This term is zero for p=1

$$T_c = \text{communication overhead}$$

$$T_s = \text{serial (non-parallizable) fraction}$$

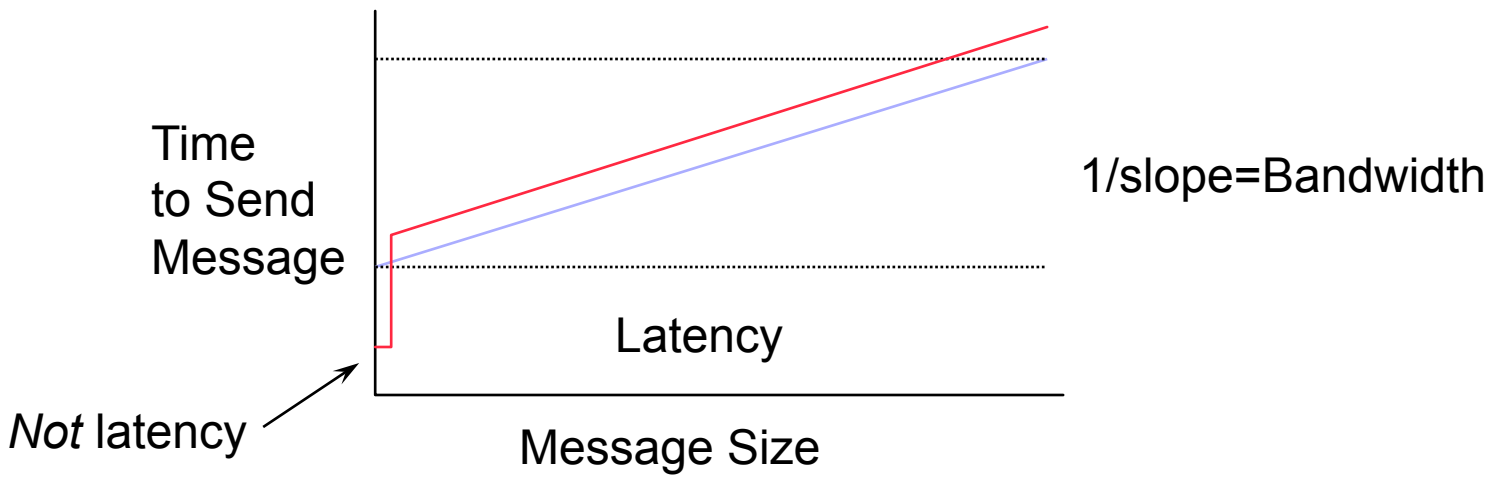
$$T_p = \text{parallel fraction}$$

Latency and Bandwidth

- Simplest model $s + r\,n$
- s includes both hardware (gate delays) and software (context switch, setup)
- r includes both hardware (raw bandwidth of interconnection and memory system) and software (packetization, copies between user and system)
- Head-to-head and pingpong values may differ

Interpreting Latency and Bandwidth

- Bandwidth is the inverse of the slope of the line
  time = latency + (1/rate) size_of_message
- For performance estimation purposes, latency is the limit(n→0) of the time to send n bytes
- Latency is sometimes described as "time to send a message of zero bytes". This is true *only* for the simple model. The number quoted is sometimes misleading.



Timing MPI Programs (C)

- The elapsed (wall-clock) time between two points in an MPI program can be computed using **MPI_Wtime**:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "time is %f\n", t2 - t1 );
```

- The value returned by a single call to **MPI_Wtime** has little value.
- Times in general are local, but an implementation might offer synchronized times. See attribute **MPI_WTIME_IS_GLOBAL**.

Timing MPI Programs (Fortran)

- The elapsed (wall-clock) time between two points in an MPI program can be computed using **MPI_Wtime**:

```
double precision t1, t2
t1 = MPI_Wtime()
...
t2 = MPI_Wtime()
print *, 'time is ', t2 - t1
```

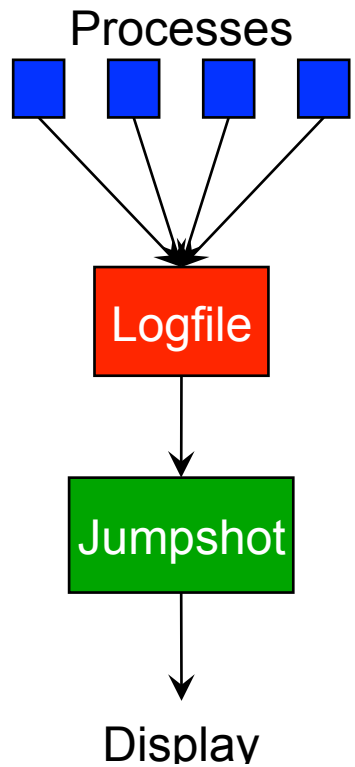
- The value returned by a single call to **MPI_Wtime** has little value.
- Times in general are local, but an implementation might offer synchronized times. See attribute **MPI_WTIME_IS_GLOBAL**.

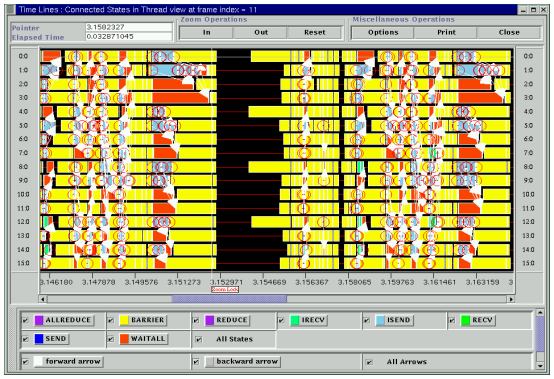
Measuring Performance

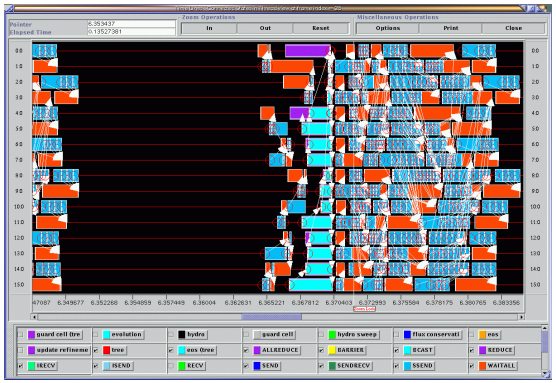
- Using MPI_Wtime
  - timers are *not* continuous — use MPI_Wtick to find resolution
- MPI_Wtime is local unless the MPI_WTIME_IS_GLOBAL attribute is true
  - MPI attributes are an advanced topic – ask me afterwards if you are interested
- MPI Profiling interface provides a way to easily instrument the MPI calls in an application
- Many performance measurement tools exist for MPI programs — take advantage of them

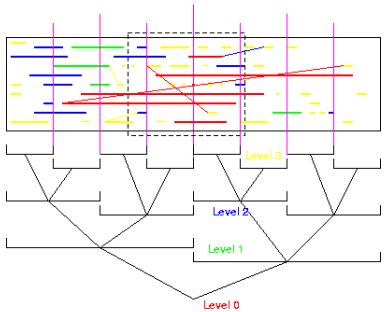
Performance Visualization with Jumpshot

- For detailed analysis of parallel program behavior, timestamped events are collected into a log file during the run.
- A separate display program (Jumpshot) aids the user in conducting a post mortem analysis of program behavior.

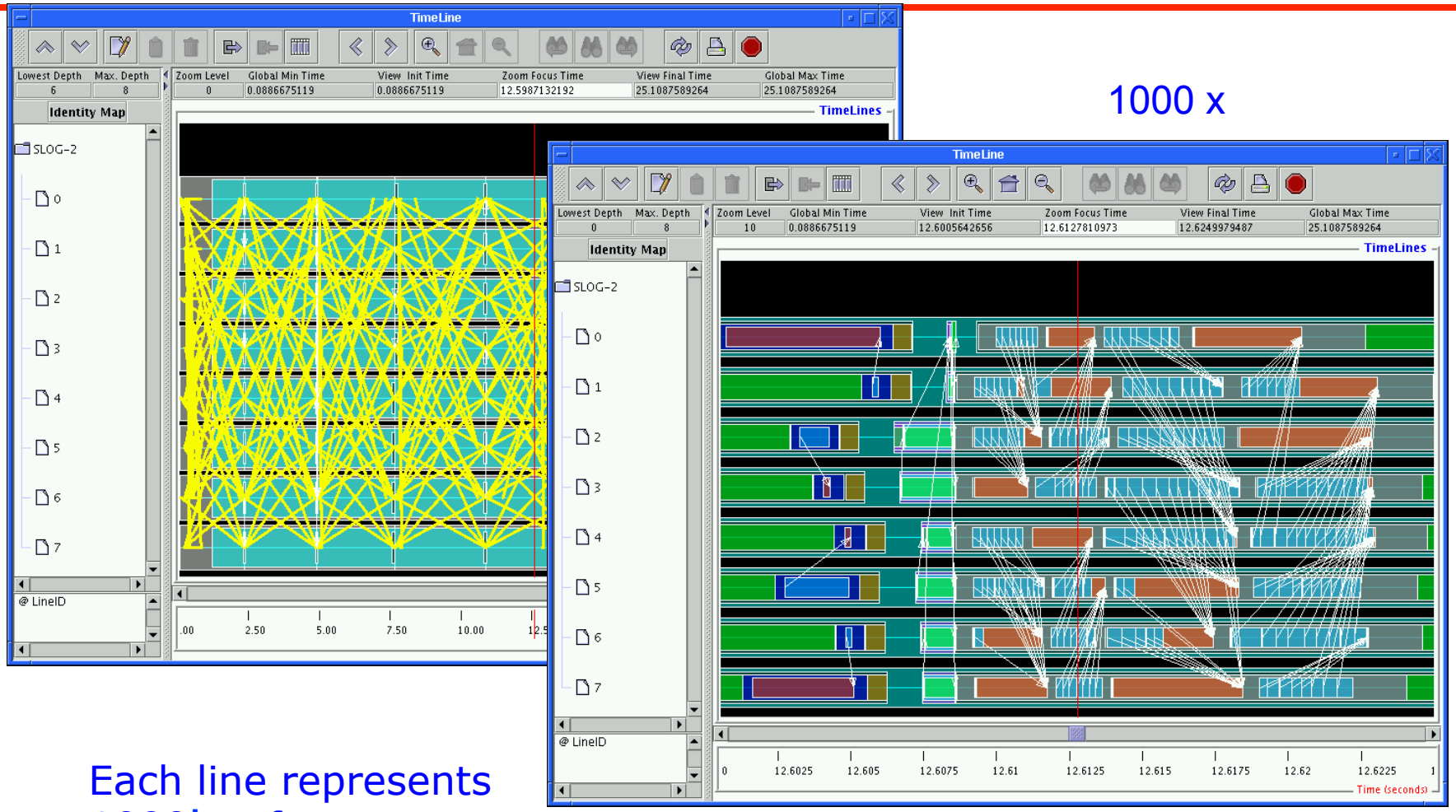








Using Jumpshot to look at FLASH at multiple Scales



1000 x

Each line represents 1000's of messages

Detailed view shows opportunities for optimization

Implementing Master/Worker Algorithms

- Many algorithms have one or more master processes that send tasks and receive results from worker processes
- Because there is one (or a few) controlling processes, the master can become a bottleneck

Skeleton Master Process

```
do while(.not. Done)
   ! Get results from anyone
   call MPI_Recv( a,…, status, ierr )
   ! If this is the last data item,
   ! set done  to .true.
   ! Else send more work to them
   call MPI_Send(b,…,status(MPI_SOURCE),&
                    … , ierr )
enddo
```

- Not included:
  - Sending initial work to all processes
  - Deciding when to set done

Skeleton Worker Process

- ```
  Do while (.not. Done)
     ! Receive work from master
     call MPI_Recv( a, …, status, ierr )
     … compute for task
     ! Return result to master
     call MPI_Send( b, …, ierr )
  enddo
  ```
- Not included:
  - Detection of termination (probably message from master)
  - An alternative would be a test for a nonblocking barrier (which MPI doesn't have)
  ```

Problems With Master/ Worker

- Worker processes have nothing to do while waiting for the next task
- Many workers may try to send data to the master at the same time
  - Could be a problem if data size is very large, such as 20-100 MB
- Master may fall behind in servicing requests for work if many processes ask in a very short interval
- Presented with many requests, master may not evenly respond

Spreading out communication

- Use double buffering to overlap request for more work with work

```
Do while (.not. Done)
    ! Receive work from master
    call MPI_Wait( request, status, ierr )
    ! Request MORE work
    call MPI_Send( …, send_work, …, ierr )
    call MPI_IRecv( a2, …, request, ierr )
    … compute for task
    ! Return result to master (could also be nb)
    call MPI_Send( b, …, ierr )
enddo
```

- MPI_Cancel
  - Last Irecv may never match; remove it with MPI_Cancel
  - You must still complete the request with MPI_Test or MPI_Wait, or MPI_Request_free.

Limiting Memory Demands on Master

- Using MPI_Ssend and MPI_Issend to encourage limits on memory demands
  - MPI_Ssend and MPI_Issend do *not* specify that the data itself doesn't move until the matching receive is issued, but that is the easiest (and most common) way to implement the synchronous send operations
  - Replace MPI_Send in worker with
    - MPI_Ssend for blocking
    - MPI_Issend for nonblocking (even less synchronization)

Distributing work further

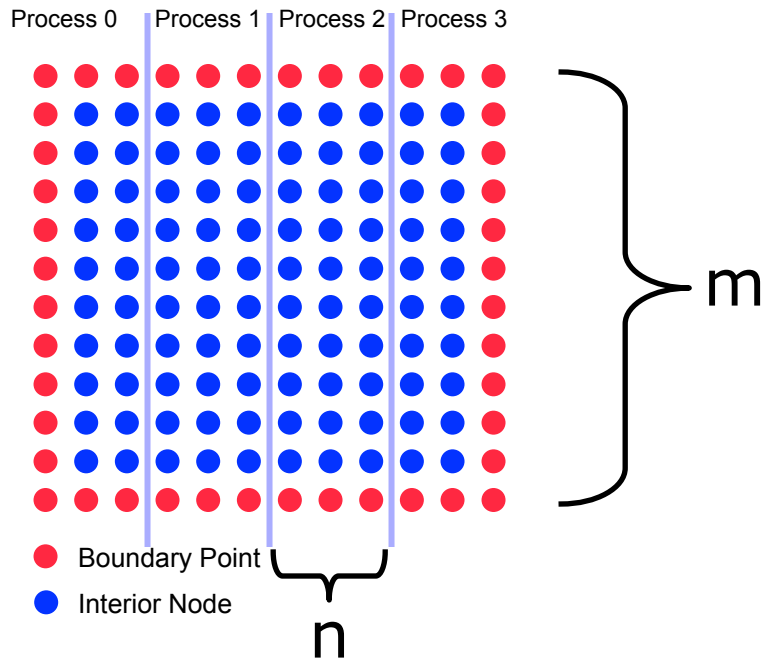
- Use multiple masters, workers select a master to request work from at random
- Keep more work locally
- Use threads to implement work stealing (but you must be use a *thread-safe* implementation of MPI)

Mesh-Based Computations in MPI

- First step: decompose the problem
- Second step: arrange for the communication of data
- Example: "Jacobi" iteration
  - Represents the communication in many mesh-based computations
  - Not a good algorithm (we'll see better ways later)
  - Not the best decomposition (more scalable decompositions are more complex to program — unless you use higher-level libraries)

Jacobi Iteration (Fortran Ordering)

- Simple parallel data structure



- Processes exchange columns with neighbors
- Local part declared as xlocal(m,0:n+1)

Send and Recv (Fortran)

- Simplest use of send and recv

```
integer status(MPI_STATUS_SIZE)

call MPI_Send( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                    left_nbr, 0, ring_comm, ierr )
call MPI_Recv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, &
                    right_nbr, 0, ring_comm, status, ierr )
call MPI_Send( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                    right_nbr, 0, ring_comm, ierr )
call MPI_Recv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                    left_nbr, 0, ring_comm, status, ierr )
```

Performance of Simplest Code

- Very poor performance on SP2
  - Rendezvous sequentializes sends/receives
- OK performance on T3D (implementation tends to buffer operations)

Better to start receives first (Fortran)

- Irecv, Isend, Waitall - ok performance

```
integer statuses(MPI_STATUS_SIZE,4), requests(4)

call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION,&
                left_nbr, ring_comm, requests(2), ierr )
call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION,&
                right_nbr, ring_comm, requests(4), ierr )
call MPI_Isend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                right_nbr, ring_comm, requests(1), ierr )
call MPI_Isend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                left_nbr, ring_comm, requests(3), ierr )
call MPI_Waitall( 4, requests, statuses, ierr )
```

MPI and Threads

- Symmetric Multiprocessors (SMPs) are a common building block of many parallel machines
- The preferred programming model for SMPs with *threads*
  - POSIX ("pthreads")
  - OpenMP
  - sproc (SGI)
  - compiler-managed parallelism

Thread Interfaces

- POSIX "pthreads"
- Windows
  - Kernel threads
  - User threads called "fibers"

Library-based

Invoke a routine in a separate thread

- Java
  - First major language with threads *in* the language
  - Provides memory synchronization model: methods (procedures) declared "synchronized" executed by one thread at a time
  - (don't mention Ada, which has tasks)
- OpenMP
  - Mostly directive-based parallel loops
  - Some thread features (lock/unlock)
  - http://www.openmp.org

Threads and MPI

- **`MPI_Init_thread(&argc,&argv,required,&provided)`**
  - Thread modes:
    - MPI_THREAD_SINGLE — One thread (MPI_Init)
    - MPI_THREAD_FUNNELED — One thread making MPI calls – most common case when MPI combined with OpenMP
    - MPI_THREAD_SERIALIZED — One thread at a time making MPI calls
    - MPI_THREAD_MULTIPLE — Free for all
- Coexist with compiler (thread) parallelism for SMPs
- MPI could have defined the same modes on a communicator basis (more natural, and MPICH will do this through attributes)

What's in MPI-2

- Extensions to the message-passing model
  - Dynamic process management
  - One-sided operations (remote memory access)
  - Parallel I/O
  - Thread support
- Making MPI more robust and convenient
  - C++ and Fortran 90 bindings
  - External interfaces, handlers
  - Extended collective operations
  - Language interoperability

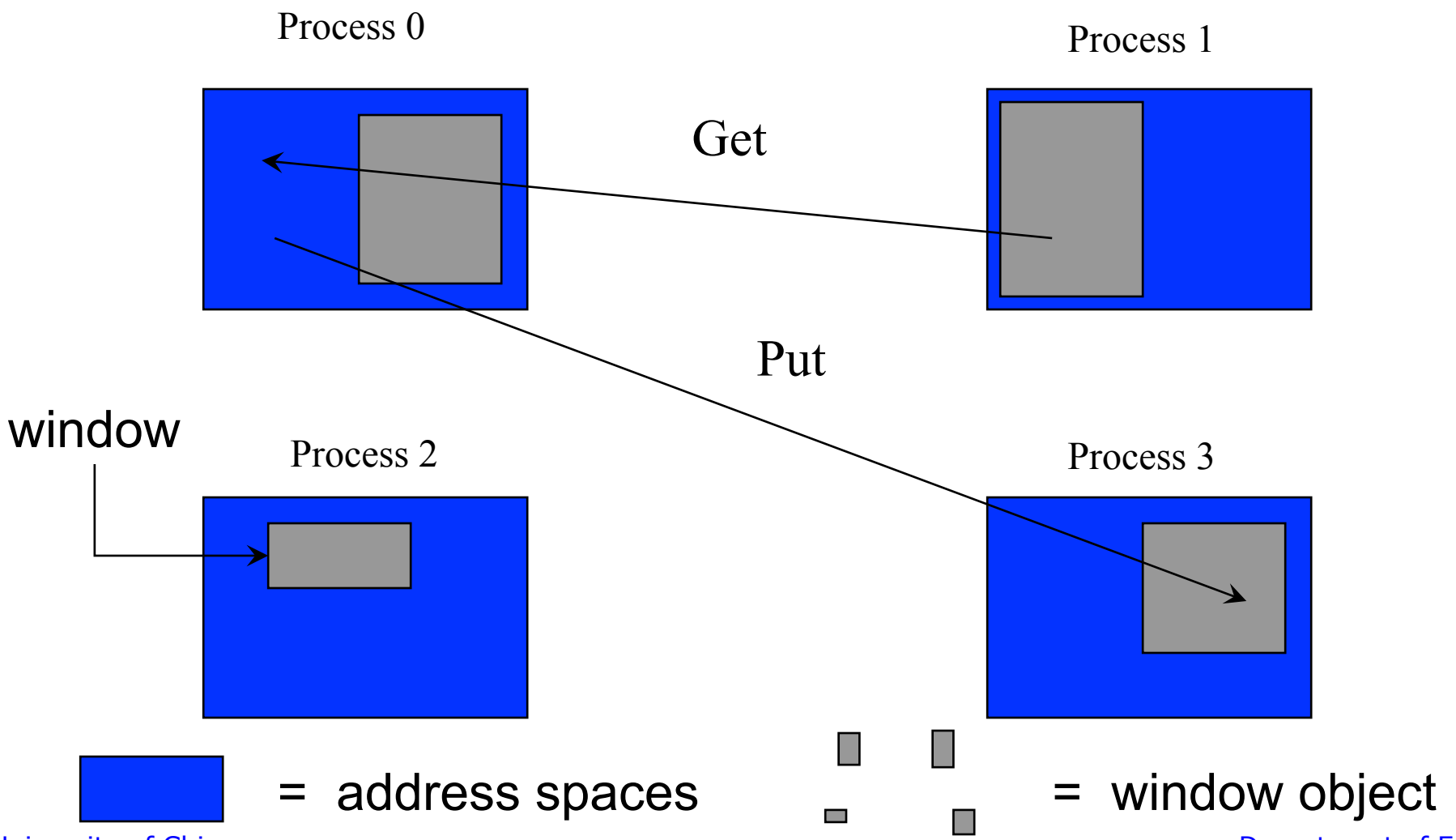
MPI as a Setting for Parallel I/O

- Writing is like sending and reading is like receiving
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- I.e., lots of MPI-like machinery
- We will discuss a high-level approach to using MPI-IO

One-Sided Operations in MPI-2 (also called Remote Memory Access)

- Synchronization is separate from data movement.
- Balancing efficiency and portability across a wide class of architectures
  - shared-memory multiprocessors
  - NUMA architectures
  - distributed-memory MPP's, clusters
  - Workstation networks
- Retaining "look and feel" of MPI-1
- Dealing with subtle memory behavior issues: cache coherence, sequential consistency

Remote Memory Access Windows and Window Objects



Why Use RMA?

- Performance
- May allow more dynamic, asynchronous algorithms
- But Get/Put is not Load/Store
  - Synchronization is exposed in MPI one-sided operations

Basic RMA Functions for Communication

- **`MPI_Win_create`** exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- **`MPI_Win_free`** deallocates window object

- **`MPI_Put`** moves data from local memory to remote memory
- **`MPI_Get`** retrieves data from remote memory into local memory
- **`MPI_Accumulate`** updates remote memory using local values
- Data movement operations are non-blocking
- Subsequent synchronization on window object needed to ensure operation is complete

RMA Functions for Synchronization

- Multiple ways to synchronize:
- **`MPI_Win_fence`** – barrier across all processes participating in window, allows BSP-like model
- **`MPI_Win_{start, complete, post, wait}`**
  - involves groups of processes, such as nearest neighbors in grid
- **`MPI_Win_{lock, unlock}`** – involves single other process
  - Not to be confused with lock,unlock used with threads

Comparing RMA and Point-to-Point Communication

- The following example shows how to achieve the same communication pattern using point-to-point and remote memory access communication
- Illustrates the issues, not an example of where to use RMA

Point-to-point

```
/* Create communicator for separate context for processes 0 and 1 */
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_split( MPI_COMM_WORLD, rank <= 1, rank, &comm );
/* Only processes 0 and 1 execute the rest of this */
if (rank > 1) return;
/* Process 0 sends and Process 1 receives */
if (rank == 0) {
    MPI_Isend( outbuf, n, MPI_INT, 1, 0, comm, &request );
}
else if (rank == 1) {
    MPI_Irecv( inbuf, n, MPI_INT, 0, 0, comm, &request );
}
/* Allow other operations to proceed (communication or
   computation) */
…
/* Complete the operation */
MPI_Wait( &request, &status );
/* Free communicator */
MPI_Comm_free( &comm );
```

RMA

```
/* Create memory window for separate context for processes 0
and 1 */
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_split( MPI_COMM_WORLD, rank <= 1, rank, &comm );
if (rank == 0)
    MPI_Win_create( NULL, 0, sizeof(int),
                    MPI_INFO_NULL, comm, &win );
else if (rank == 1 )
    MPI_Win_create( inbuf, n * sizeof(int), sizeof(int),
                    MPI_INFO_NULL, comm, &win );
/* Only processes 0 and 1 execute the rest of this */
if (rank > 1) return;
/* Process 0 puts into process 1 */
MPI_Win_fence( 0, win );
if (rank == 0)
    MPI_Put( outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win );
/* Allow other operations to proceed (communication or
computation) */
…
/* Complete the operation */
MPI_Win_fence( 0, win );
/* Free the window */
MPI_Win_free( &win );
```

Using MPI_Win_fence

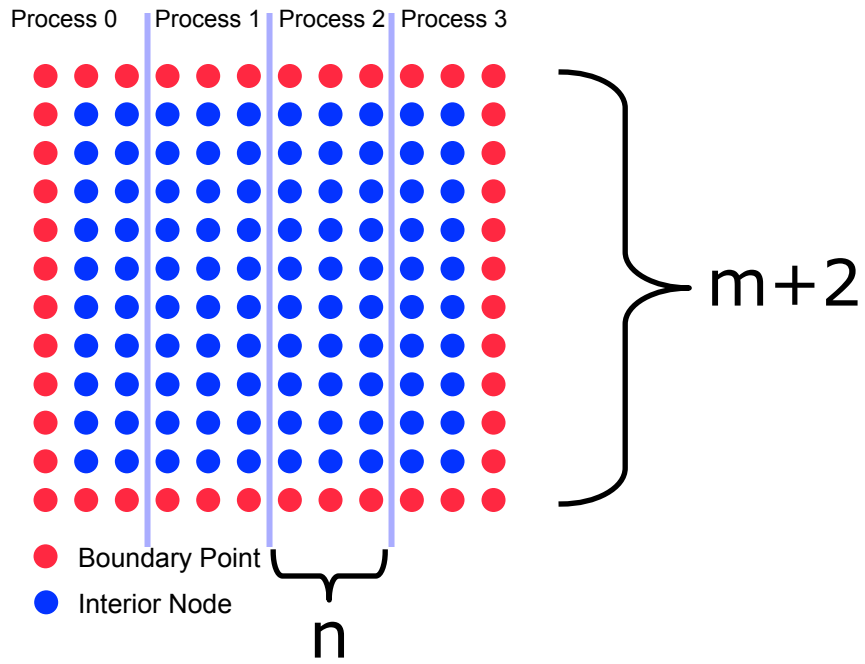
```
MPI_Win_create( A, ..., &win );
MPI_Win_fence( 0, win );
if (rank == 0) {
    /* Process 0 puts data into many local windows */
    MPI_Put( ... , win );
    MPI_Put( ... , win );
}
/* This fence completes the MPI_Put operations initiated
   by process 0 */
MPI_Win_fence( 0, win );
/* All processes initiate access to some window to extract data */
MPI_Get( ... , win );
/* The following fence completes the MPI_Get operations */
MPI_Win_fence( 0, win );
/* After the fence, processes can load and store into A, the local window */
A[rank] = 4;
printf( "A[%d] = %d\n", 0, A[0] );
MPI_Win_fence( 0, win );
/* We need a fence between stores and RMA operations */
MPI_Put( ... , win );
/* The following fence completes the preceding Put */
MPI_Win_fence( 0, win );
```

Example of MPI RMA: Ghost Point Exchange

- Multiparty data exchange
- Jacobi iteration in 2 dimensions
  - Model for PDEs, Sparse matrix-vector products, and algorithms with surface/volume behavior
  - Issues are similar to unstructured grid problems (but harder to illustrate)

Jacobi Iteration (Fortran Ordering)

- Simple parallel data structure



- Processes exchange columns with neighbors
- Local part declared as xlocal(m,0:n+1)

Ghostpoint Exchange with RMA

```
subroutine exchng1( a, m, s, e, win, left_nbr, right_nbr )
use mpi
integer m, s, e
double precision a(0:m+1,s-1:e+1)
integer win, left_nbr, right_nbr
integer ierr
integer(kind=MPI_ADDRESS_KIND) left_ghost_disp, right_ghost_disp

call MPI_WIN_FENCE( 0, win, ierr )
! Put left edge into left neighbor's right ghost cells
right_ghost_disp = 1 + (m+2)*(e-s+2)
call MPI_PUT( a(1,s), m, MPI_DOUBLE_PRECISION, &
              left_nbr, right_ghost_disp, m, &
              MPI_DOUBLE_PRECISION, win, ierr )
! Put bottom edge into right neighbor's left ghost cells
left_ghost_disp = 1
call MPI_PUT( a(1,e), m, MPI_DOUBLE_PRECISION, &
              right_nbr, left_ghost_disp, m, &
              MPI_DOUBLE_PRECISION, win, ierr )
call MPI_WIN_FENCE( 0, win, ierr )
return
end
```

MPI-2 Status Assessment

- All MPP vendors now have MPI-1. Free implementations (MPICH, LAM) support heterogeneous workstation networks.
- MPI-2 implementations are being undertaken now by all vendors.
  - Multiple complete MPI-2 implementations available
- MPI-2 implementations appearing piecemeal, with I/O first.
  - I/O available in most MPI implementations
  - One-sided available in some (e.g .NEC and Fujitsu, parts from SGI and HP, parts coming soon from IBM)
  - MPI RMA an important part of the spectacular results on the Earth Simulator
  - Most of dynamic and one sided in LAM, WMPI, MPICH2

MPICH Goals

- Complete MPI implementation
- Portable to all platforms supporting the message-passing model
- High performance on high-performance hardware
- As a research project:
  - exploring tradeoff between portability and performance
  - removal of performance gap between user level (MPI) and hardware capabilities
- As a software project:
  - a useful free implementation for most machines
  - a starting point for vendor proprietary implementations

MPICH2

- All-new implementation is our vehicle for research in
  - Thread safety and efficiency (e.g., avoid thread locks)
  - Optimized MPI datatypes
  - Optimized Remote Memory Access (RMA)
  - High Scalability (64K MPI processes and more)
  - Exploiting Remote Direct Memory Access (RDMA) capable networks (Myrinet)
  - All of MPI-2, including dynamic process management, parallel I/O, RMA
- Parallel Environments
  - Clusters
  - IBM BG/L
  - New interconnect technologies (Infiniband)
  - Cray Red Storm
  - Others
  - Many vendors start with MPICH in crafting custom, optimized MPI's

MPICH-2 Status and Schedule

- Supports all of MPI-1 and all of MPI-2, including all of MPI-IO, active-target RMA, dynamic processes, passive-target RMA on many platforms
- Improved performance
- New algorithms for collective operations
- Improved robustness
- Process manager interface
  - Supports multiple process managers
  - Includes the MPD-based manager (provides scalable startup)
- Multiple devices implemented
  - Sockets, shared memory, Infiniband
  - Many groups already using MPICH2 for their MPI implementations

Getting MPICH for Your Cluster

- MPICH1:
  - www.mcs.anl.gov/mpi/mpich
- MPICH2:
  - www.mcs.anl.gov/mpi/mpich2

High-Level Programming With MPI

- MPI was designed from the beginning to support libraries
- Many libraries exist, both open source and commercial
- Sophisticated numerical programs can be built using libraries
  - Scalable I/O of data to a community standard file format
  - Solve a PDE (e.g., PETSc)

Higher Level I/O Libraries

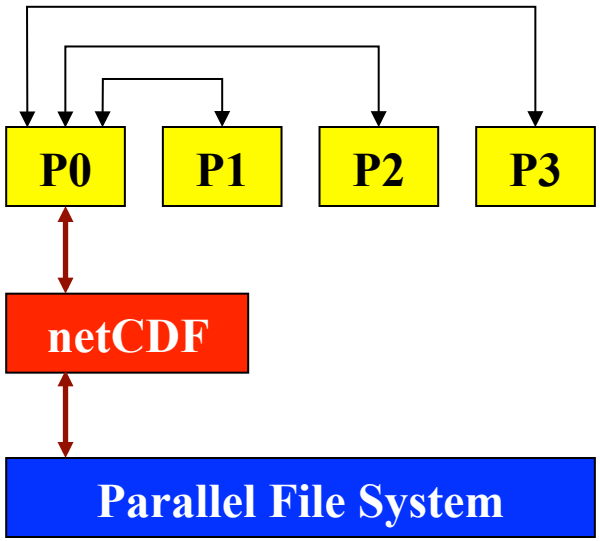
- Scientific applications work with structured data and desire more self-describing file formats
- netCDF and HDF5 are two popular "higher level" I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel machines, these should be built on top of MPI-IO

Parallel netCDF (PnetCDF)

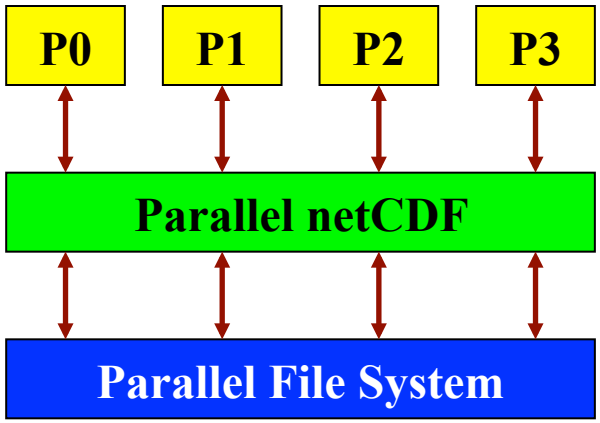
- Collaboration between NWU and ANL as part of the Scientific Data Management SciDAC
- netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
- Popular in both fusion and climate communities
- Parallel netCDF is an effort to
  - Very similar API to netCDF
  - Tuned for better performance in today's computing environments
  - Retains the file format so netCDF and PnetCDF applications can share files

I/O in netCDF

- Original netCDF
  - Parallel read
    - All processes read the file independently
    - No possibility of collective optimizations
  - Sequential write
    - Parallel writes are carried out by shipping data to a single process



- PnetCDF
  - Parallel read/write to shared netCDF file
  - Built on top of MPI-IO which utilizes optimal I/O facilities of the parallel file system and MPI-IO implementation
  - Allows for MPI-IO hints and datatypes for further optimization



PnetCDF Example Part 1

```
int main(int argc, char *argv[])
{
  double temps[512*512/NR_PROCS];
  int status, lon_id, lat_id, temp_id, dim_id[2],
      dim_off[2], dim_size[2];
  status = ncmpi_create(MPI_COMM_WORLD, "foo",
           NC_CLOBBER, MPI_INFO_NULL, &file_id);
  status = ncmpi_def_dim(file_id, "longitude",
           512, &lon_id);
  status = ncmpi_def_dim(file_id, "latitude",
           512, &lat_id);
  dim_id[0] = lon_id; dim_id[1] = lat_id;
  status = ncmpi_def_var(file_id, "temp",
           NC_DOUBLE, 2, dim_id, &temp_id);
```

PnetCDF Example Part 2

```
    /* leave define mode, enter coll. data mode */
    status = ncmpi_enddef(file_id);

    partition_problem_space(dim_off, dim_size);

    /* perform calculations until time to write */

    /* each proc. writes its part. collectively */
    status = ncmpi_put_vara_double_all(file_id,
             temp_id, dim_off, dim_size, temps);

    status = ncmpi_close(file_id);
}
```

More Information on PnetCDF

- Parallel netCDF web site:

  http://www.mcs.anl.gov/parallel-netcdf/

- Parallel netCDF mailing list:

  Mail to majordomo@mcs.anl.gov with the body "subscribe parallel-netcdf"

- The SDM SciDAC web site:

  http://sdm.lbl.gov/sdmcenter/

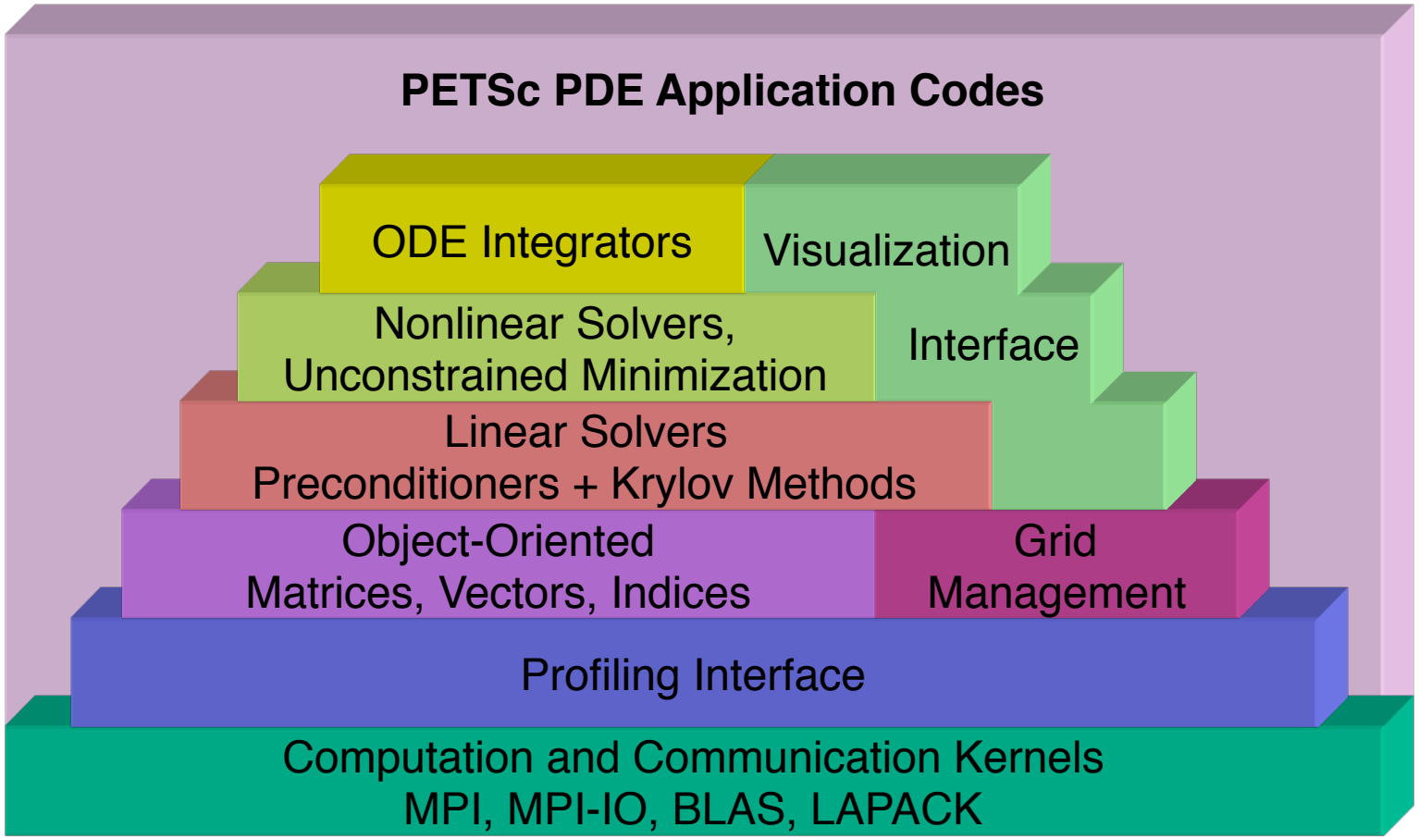
The PETSc Library

- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution
- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)

Hello World in PETSc

```
#include "petsc.h"
int main( int argc, char *argv[] )
{
  int rank;
  PetscInitialize( &argc, &argv, 0, 0 );
  MPI_Comm_rank( PETSC_COMM_WORLD, &rank );
  PetscSynchronizedPrintf( PETSC_COMM_WORLD,
       "Hello World from rank %d\n", rank );
  PetscSynchronizedFlush( PETSC_COMM_WORLD );
  PetscFinalize( );
  return 0;
}
```

Structure of PETSc



PETSc Numerical Components

Nonlinear Solvers		
Newton-based Methods		Other
Line Search	Trust Region	

Time Steppers			
Euler	Backward Euler	Pseudo Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-STAB	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwartz	Block Jacobi	Jacobi	ILU	ICC	LU (Sequential only)	Others

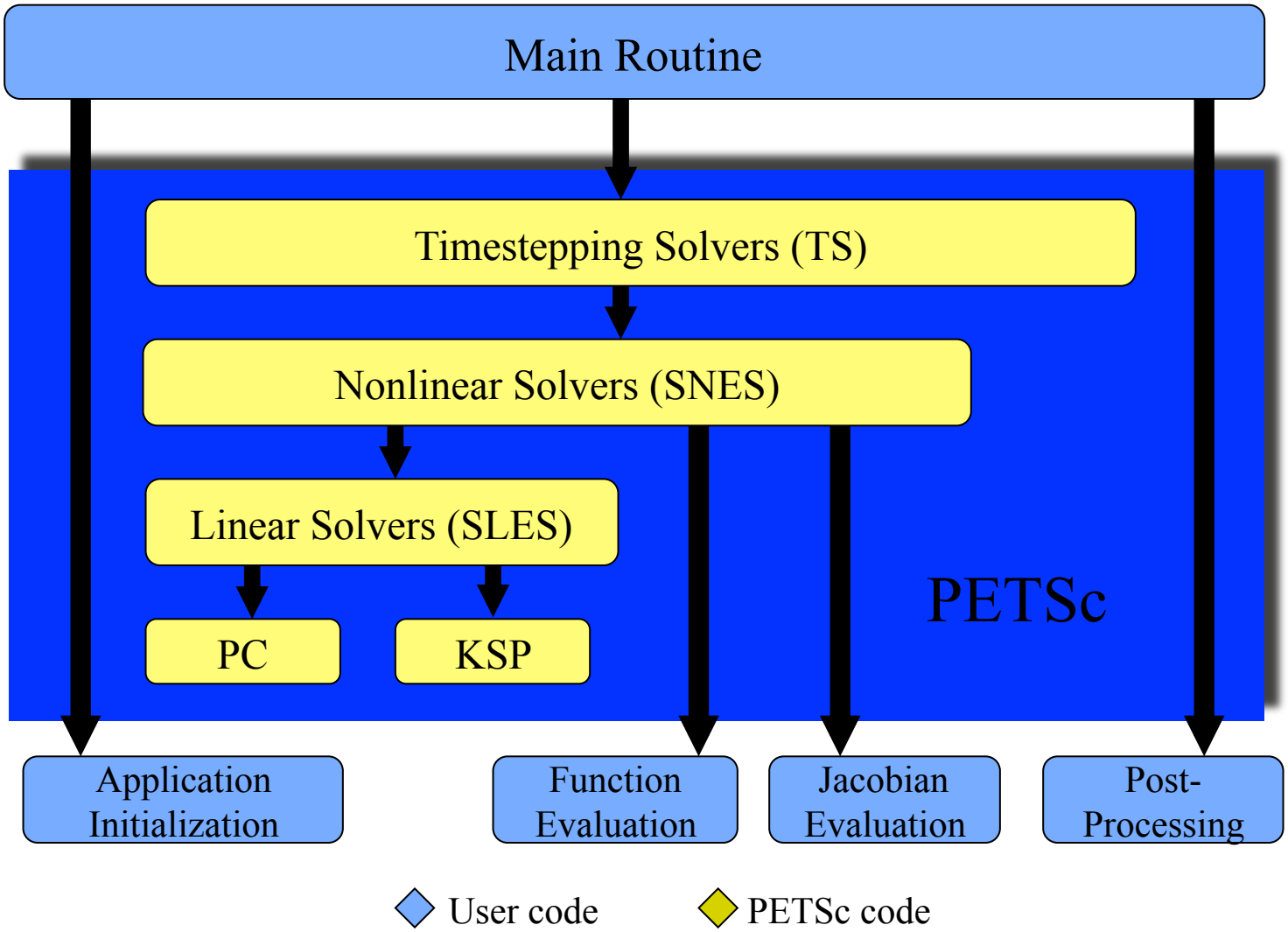
Matrices					
Compressed Sparse Row (AIJ)	Blocked Compressed Sparse Row (BAIJ)	Block Diagonal (BDIAG)	Dense	Matrix-free	Other

Distributed Arrays

Vectors

Index Sets			
Indices	Block Indices	Stride	Other

Flow of Control for PDE Solution



Poisson Solver in PETSc

- The following 7 slides show a complete 2-d Poisson solver in PETSc. Features of this solver:
  - Fully parallel
  - 2-d decomposition of the 2-d mesh
  - Linear system described as a sparse matrix; user can select many different sparse data structures
  - Linear system solved with any user-selected Krylov iterative method and preconditioner provided by PETSc, including GMRES with ILU, BiCGstab with Additive Schwarz, etc.
  - Complete performance analysis built-in
- Only 7 slides of code!

Solve a Poisson Problem with Preconditioned GMRES

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#include <math.h>
#include "petscsles.h"
#include "petscda.h"
extern Mat FormLaplacianDA2d( DA, int );
extern Vec FormVecFromFunctionDA2d( DA, int, double (*)(double,double) );
/* This function is used to define the right-hand side of the
   Poisson equation to be solved */
double func( double x, double y ) {
    return sin(x*M_PI)*sin(y*M_PI); }

int main( int argc, char *argv[] )
{
    SLES      sles;
    Mat       A;
    Vec       b, x;
    DA        grid;
    int       its, n, px, py, worldSize;

    PetscInitialize( &argc, &argv, 0, 0 );
```

PETSC “objects” hide details of distributed data structures and function parameters

```
/* Get the mesh size.  Use 10 by default */
n = 10;
PetscOptionsGetInt( PETSC_NULL, "-n", &n, 0 );
/* Get the process decomposition.  Default it the same as without
   DAs */
px = 1;
PetscOptionsGetInt( PETSC_NULL, "-px", &px, 0 );
MPI_Comm_size( PETSC_COMM_WORLD, &worldSize );
py = worldSize / px;
```

PETSc provides routines to access parameters and defaults

```
/* Create a distributed array */
DACreate2d( PETSC_COMM_WORLD, DA_NONPERIODIC, DA_STENCIL_STAR,
                    n, n, px, py, 1, 1, 0, 0, &grid );
```

```
/* Form the matrix and the vector corresponding to the DA */
A = FormLaplacianDA2d( grid, n );
b = FormVecFromFunctionDA2d( grid, n, func );
VecDuplicate( b, &x );
```

PETSc provides routines to create, allocate, and manage distributed data structures

```
    SLESCreate( PETSC_COMM_WORLD, &sles );
    SLESSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
    SLESSetFromOptions( sles );
    SLESSolve( sles, b, x, &its );

    PetscPrintf( PETSC_COMM_WORLD, "Solution is:\n" );
    VecView( x, PETSC_VIEWER_STDOUT_WORLD );
    PetscPrintf( PETSC_COMM_WORLD, "Required %d iterations\n", its );

    MatDestroy( A ); VecDestroy( b ); VecDestroy( x );
    SLESDestroy( sles ); DADestroy( grid );
    PetscFinalize( );
    return 0;
}
```

PETSc provides routines that solve systems of sparse linear (and nonlinear) equations

PETSc provides coordinated I/O (behavior is as-if a single process), including the output of the distributed “vec” object

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#include "petsc.h"
#include "petscvec.h"
#include "petscda.h"

/* Form a vector based on a function for a 2-d regular mesh on the
   unit square */
Vec FormVecFromFunctionDA2d( DA grid, int n,
                             double (*f)( double, double ) )
{
    Vec    V;
    int    is, ie, js, je, in, jn, i, j;
    double h;
    double **vval;

    h = 1.0 / (n + 1);
    DACreateGlobalVector( grid, &V );

    DAVecGetArray( grid, V, (void **)&vval );
```

```
/* Get global coordinates of this patch in the DA grid */
DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
ie = is + in - 1;
je = js + jn - 1;

 for (i=is ; i<=ie ; i++) {
            for (j=js ; j<=je ; j++){
               vval[j][i] = (*f)( (i + 1) * h, (j + 1) * h );
            }
   }
   DAVecRestoreArray( grid, V, (void **)&vval );

   return V;
}
```

Almost the uniprocess code

Creating a **Sparse** Matrix, Distributed Across All Processes

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#include "petscsles.h"
#include "petscda.h"

/* Form the matrix for the 5-point finite difference 2d Laplacian
   on the unit square. n is the number of interior points along a
   side */
Mat FormLaplacianDA2d( DA grid, int n )
{
    Mat    A;
    int    r, i, j, is, ie, js, je, in, jn, nelm;
    MatStencil cols[5], row;
    double     h, oneByh2, vals[5];

    h = 1.0 / (n + 1); oneByh2 = 1.0 / (h*h);

    DAGetMatrix( grid, MATMPIAIJ, &A );
    /* Get global coordinates of this patch in the DA grid */
    DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
    ie = is + in - 1;
    je = js + jn - 1;
```

Creates a parallel distributed matrix using compressed sparse row format

```
for (i=is; i<=ie; i++) {
        for (j=js; j<=je; j++){
            row.j = j; row.i = i; nelm = 0;
            if (j - 1 > 0) {
                vals[nelm]   = oneByh2;
                cols[nelm].j = j - 1; cols[nelm++].i = i;}
            if (i - 1 > 0) {
                vals[nelm]   = oneByh2;
                cols[nelm].j = j;     cols[nelm++].i = i - 1;}
            vals[nelm]   = - 4 * oneByh2;
            cols[nelm].j = j;         cols[nelm++].i = i;
            if (i + 1 < n - 1) {
                vals[nelm]   = oneByh2;
                cols[nelm].j = j;     cols[nelm++].i = i + 1;}
            if (j + 1 < n - 1) {
                vals[nelm]   = oneByh2;
                cols[nelm].j = j + 1; cols[nelm++].i = i;}
            MatSetValuesStencil( A, 1, &row, nelm, cols, vals,
                INSERT_VALUES );
        }
    }

    MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);

    return A;
}
```

Just the usual code for setting the elements of the sparse matrix (the complexity comes, as it often does, from the boundary conditions

Full-Featured PDE Solver

- Command-line control of Krylov iterative method (choice of algorithms *and* parameters)
- Integrated performance analysis
- Optimized parallel sparse-matrix operations

- Question: How many MPI calls used in example?

Setting Solver Options at Runtime

- -ksp_type [cg,gmres,bcgs,tfqmr,…]
- -pc_type [lu,ilu,jacobi,sor,asm,…]

- -ksp_max_it <max_iters>
- -ksp_gmres_restart <restart>
- -pc_asm_overlap <overlap>
- -pc_asm_type [basic,restrict,interpolate,none]
- etc ...

Other Libraries

- Many libraries exist
  - Freely available libraries
    - PETSc, ScaLAPACK, FFTW, HDF5, DOUG, GeoFEM, MP_SOLVE, MpCCI, PARASOL, ParMETIS, Prometheus, PSPASES, PLAPACK, S+, TCGMSG-MPI, Trilinos, SPRNG, TAO, …
  - Commercially supported libraries
    - E.g., NAG, IBM PESSL, IMSL, …
  - More at www.mcs.anl.gov/mpi/libraries.html
- These provide the building blocks for building large, complex, and efficient programs

Some Final Comments

- It isn't how fast you can program something *easy*
- It *is* how fast you can program what you *need*
- Libraries give MPI an advantage over other parallel programming models
- Libraries provide a way to build custom high-level programming environments
  - Many exist — use them if possible
  - Otherwise, create your own!

Conclusion

- MPI provides a well-developed, efficient and portable model for programming parallel computers
- Just as important, MPI provides features that enable and encourage the construction of software *components*.
- Parallel applications can often be quickly built using these components
- Even when no available component meets your needs, using component-oriented design can simplify the process of writing and debugging an application with MPI.