Introduction to MPI

Darius Buntinas

Mathematics and Computer Science Division

Argonne National Laboratory

Argonne, IL, USA

Outline

- Quick overview of MPI basics
- Impact of buffering and ordering of messages on performance
- Derived datatypes, collective communication
- Overview of MPI-2
- Parallel I/O with examples

Models of Parallel Programming

- Shared memory (load, store, lock, unlock, …)
- Message Passing (send, receive, broadcast, …)
- Transparent (compiler works magic …)
- Directive-based (compiler needs help ...), e.g., OpenMP

The Message-Passing Model

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

What is MPI?

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

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 message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by:
  - vendors: 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.

Hello (C)

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

int main( argc, argv )
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;
}
```

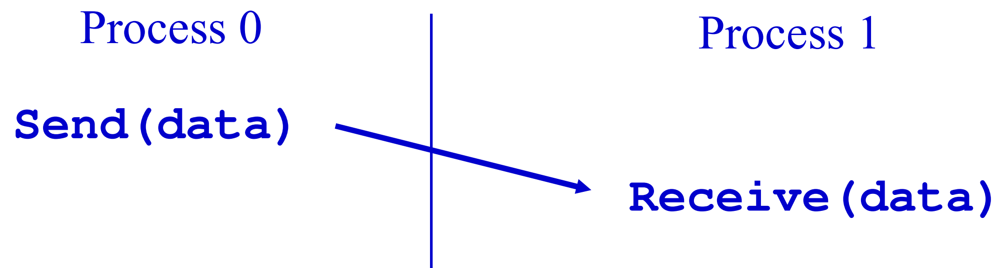
Hello (Fortran)

```
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
```

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

MPI Tags

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

MPI Basic (Blocking) Send

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

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

MPI Basic (Blocking) Receive

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

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

MPI is Simple

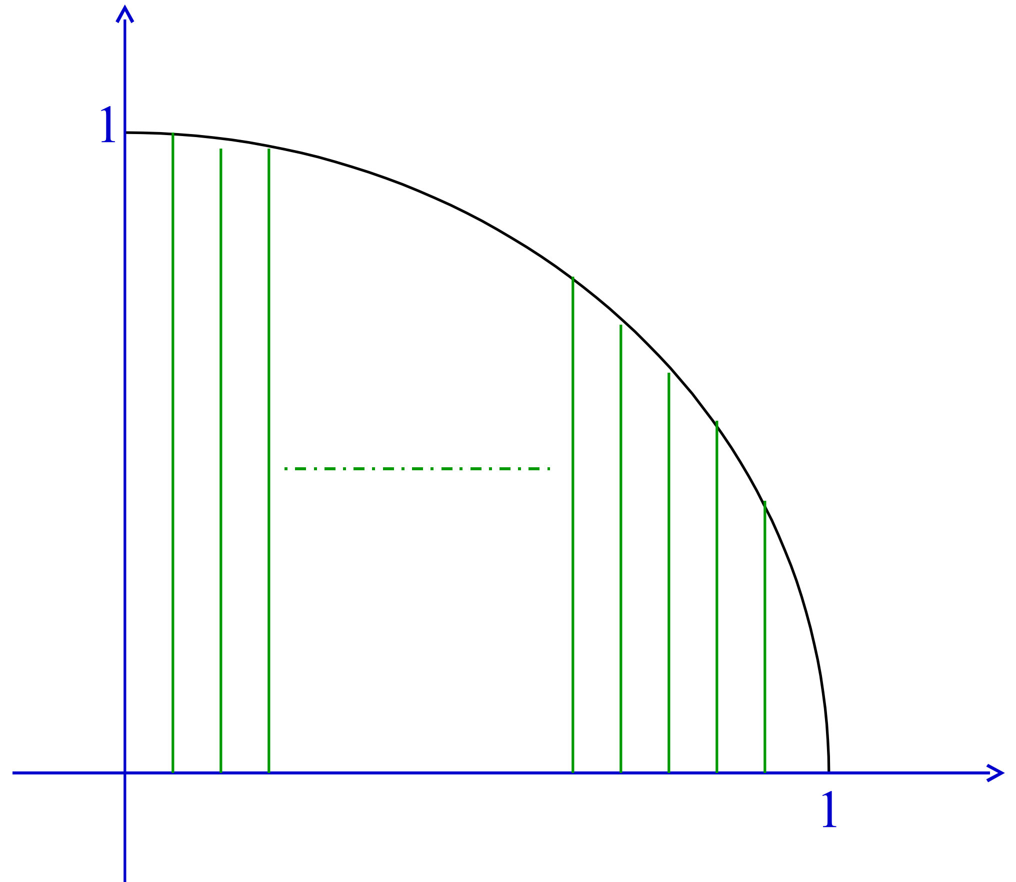
- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**

Collective Operations

- 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: Calculating Pi

$$\int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4}$$



Example: PI in C -1

```
#include "mpi.h"
#include <math.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;
}
```

Running MPI Programs

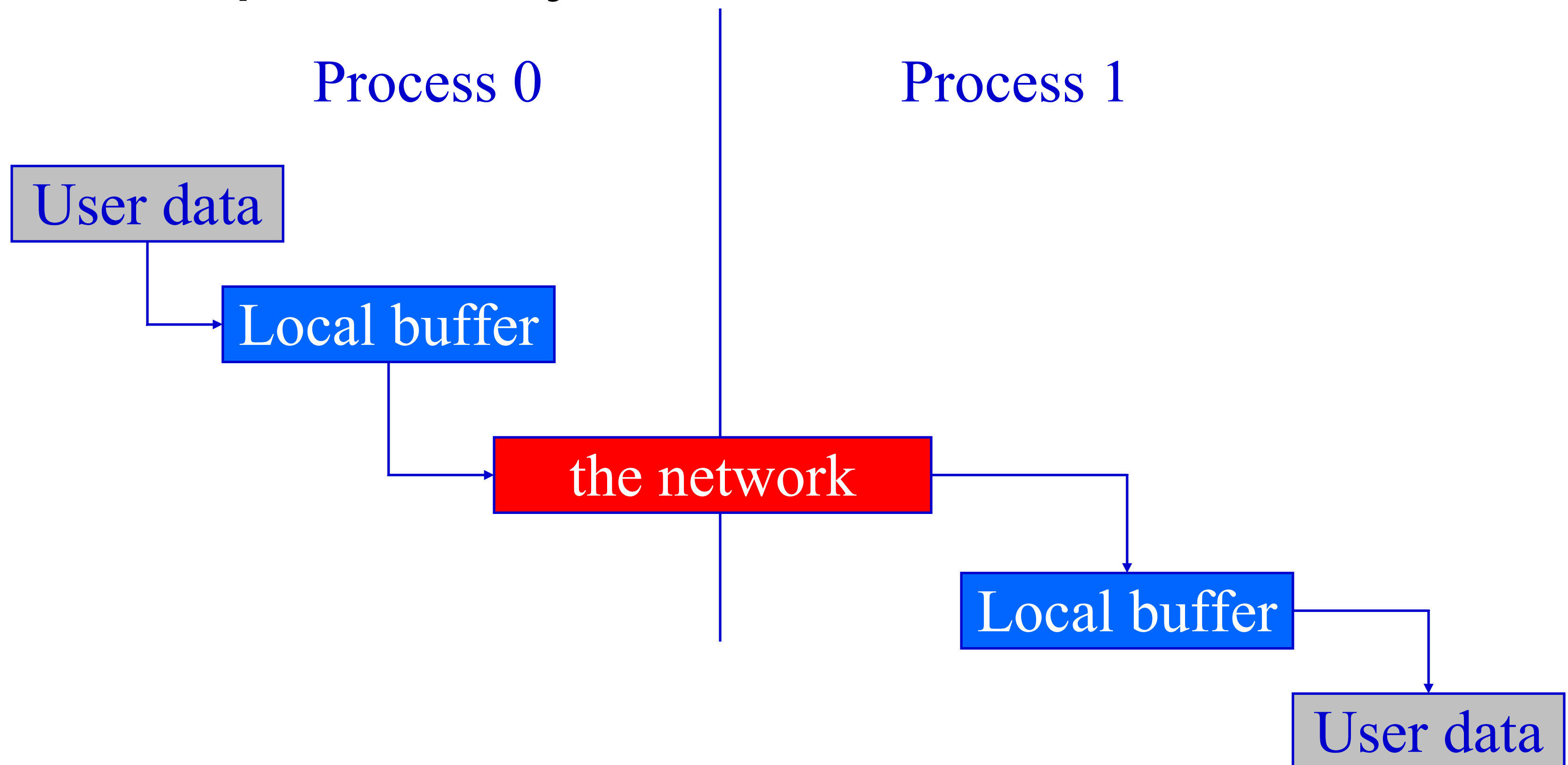
- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment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

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.

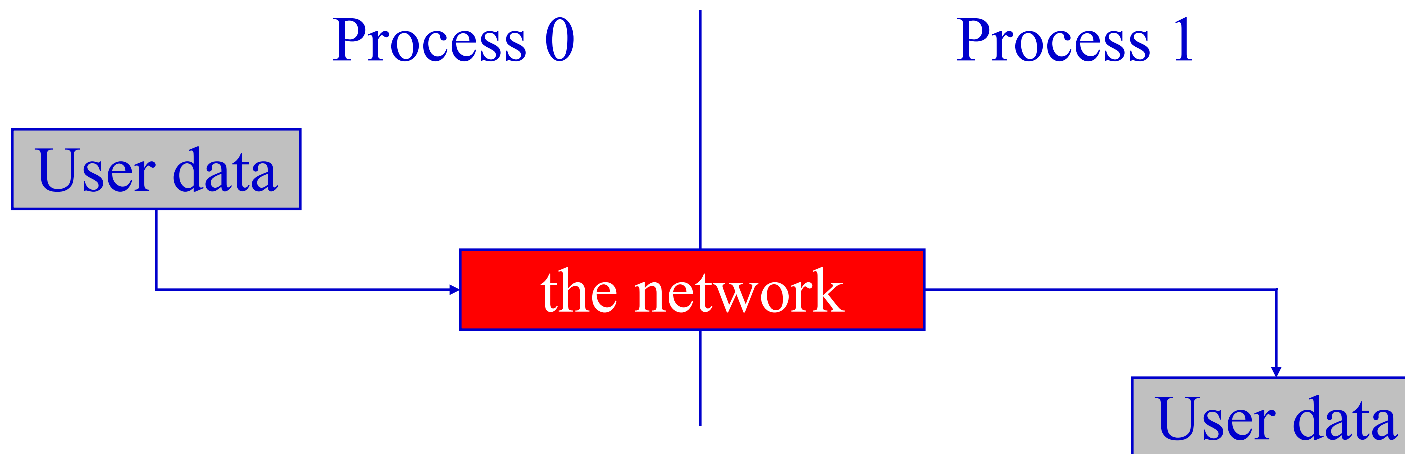
Buffers

- 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

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_Isend(start, count, datatype,
    dest, tag, comm, &request)
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request)
MPI_Wait(&request, &status)
```

- One can also test without waiting:

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

MPI's Non-blocking Operations

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

```
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)
```

- One can also test without waiting:

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

Multiple Completions (C)

- 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.

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.

Buffered Mode

- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.

```
int bufsize;
char *buf = malloc( bufsize );
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... )
...
MPI_Buffer_detach( &buf, &bufsize );
```

- **`MPI_Buffer_detach`** waits for completion.
- Performance depends on MPI implementation and size of message.

Test Your Understanding of Buffered Sends

- What is wrong with this code?

```
call MPI_Buffer_attach( buf, &
      bufsize+MPI_BSEND_OVERHEAD, ierr )
Do i=1,n

    ...
     call MPI_Bsend( bufsize bytes ... )
    ...
    Enough MPI_Recvs( )
enddo
call MPI_Buffer_detach( buf, bufsize, &
     ierr )
```

Buffering is limited

- Processor 0
  i=1
  MPI_Bsend
  MPI_Recv
  i=2
  MPI_Bsend

- i=2 Bsend fails because first Bsend has not been able to deliver the data

- Processor 1
  i=1
  MPI_Bsend
  … delay due to computing, process scheduling,...
  MPI_Recv

Correct Use of MPI_Bsend

- Fix: Attach and detach buffer in loop

```
Do i=1,n
     call MPI_Buffer_attach( buf, &
            bufsize+MPI_BSEND_OVERHEAD,ierr )
      ...
     call MPI_Bsend( bufsize bytes )
     ...
    Enough MPI_Recvs( )
    call MPI_Buffer_detach( buf, bufsize, ierr )
enddo
```

Buffer detach will wait until messages have been delivered

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, …)

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

Quick review of MPI Message passing

- Basic terms
  - nonblocking - Operation does not wait for completion
  - synchronous - Completion of send *requires* initiation (but not completion) of receive
  - ready - *Correct* send requires a matching receive
  - asynchronous - communication and computation take place simultaneously, ***not*** an MPI concept (implementations *may* use asynchronous methods)

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 %d\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.

Measuring Performance

- Using MPI_Wtime
  - timers are *not* continuous — MPI_Wtick
- MPI_Wtime is local unless the MPI_WTIME_IS_GLOBAL attribute is true
- MPI Profiling interface provides a way to easily instrument the MPI calls in an application
- Performance measurement tools for MPI

Sample Timing Harness (C)

- Average times, make several trials

```
tfinal = 100000.0
for (k=0; k<nloop; k++) {
    t1 = MPI_Wtime();
    for (i=0; i<maxloop; i++) {
        <operation to be timed>
    }
    time = MPI_Wtime() - t1;
    if (time < tfinal) tfinal = time;
}
time = time / maxloop;
```

Pitfalls in timing (C)

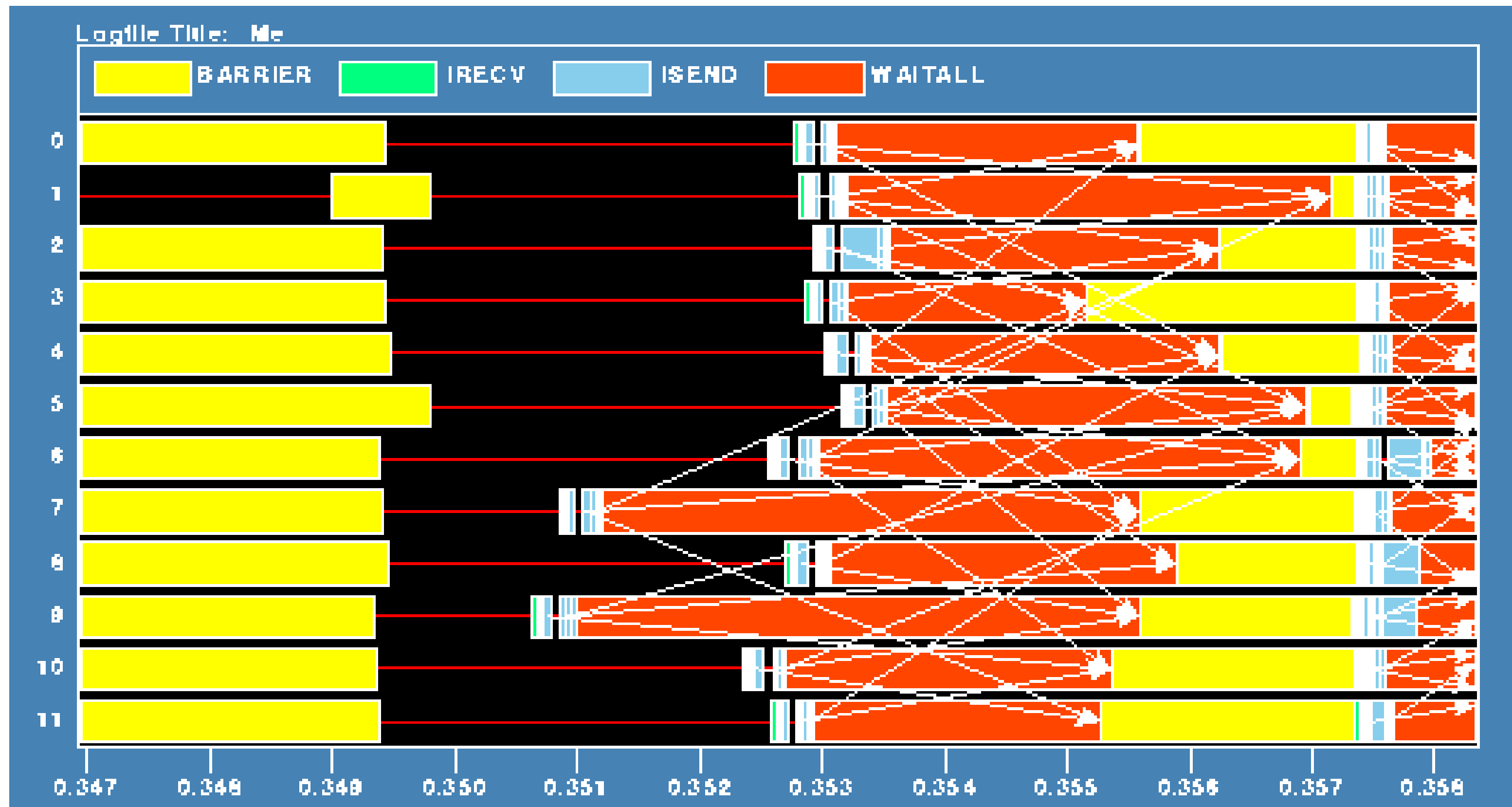
- Time too short:

```
t = MPI_Wtime();
MPI_Send(…);
time = MPI_Wtime() - t;
```

- Underestimates by MPI_Wtick, overestimates by cost of calling MPI_Wtime
- Code not paged in (always run at least twice)
- Minimums not what users see

Example of Paging Problem

- Black area is *identical* setup computation



Synchronization Delays

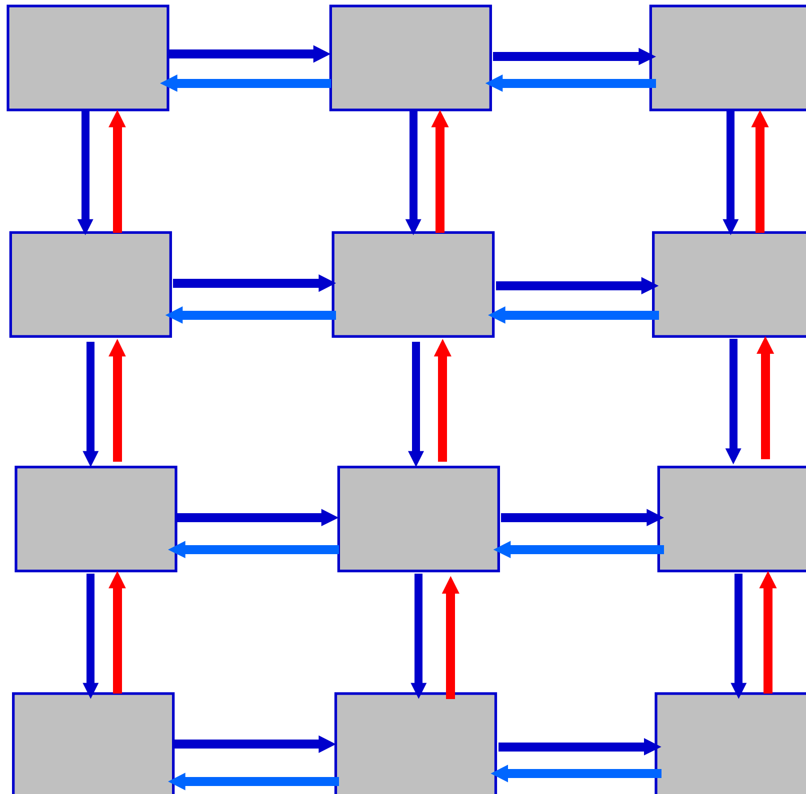
- Message passing is a cooperative method — if the partner doesn’t react quickly, a delay results
- There is a performance tradeoff caused by reacting quickly — it requires devoting resources to checking for things to do

Unexpected Hot Spots

- Even simple operations can give surprising performance behavior.
- Examples arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable stalls

Mesh Exchange

- Exchange data on a mesh



Sample Code

- Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL,&
  nbr(i), tag,comm, ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Recv(edge(1,i), len, MPI_REAL,&
  nbr(i), tag, comm, status, ierr)
  Enddo

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

```
if (has down nbr) then
    Call MPI_Send( … down … )
endif
if (has up nbr) then
    Call MPI_Recv( … up … )
endif
…
```

  sequentializes (all except the bottom process blocks)

Sequentialization

Start Start Start Start Start Start Start

Send Send Send Send Send Send Send Recv

Send Recv

Send Recv

Send Recv

Send Recv

Send Recv

Send Recv

Fix 1: Use Irecv

- ```
  Do i=1,n_neighbors
    Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
                   comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
    Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
                  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
  ```

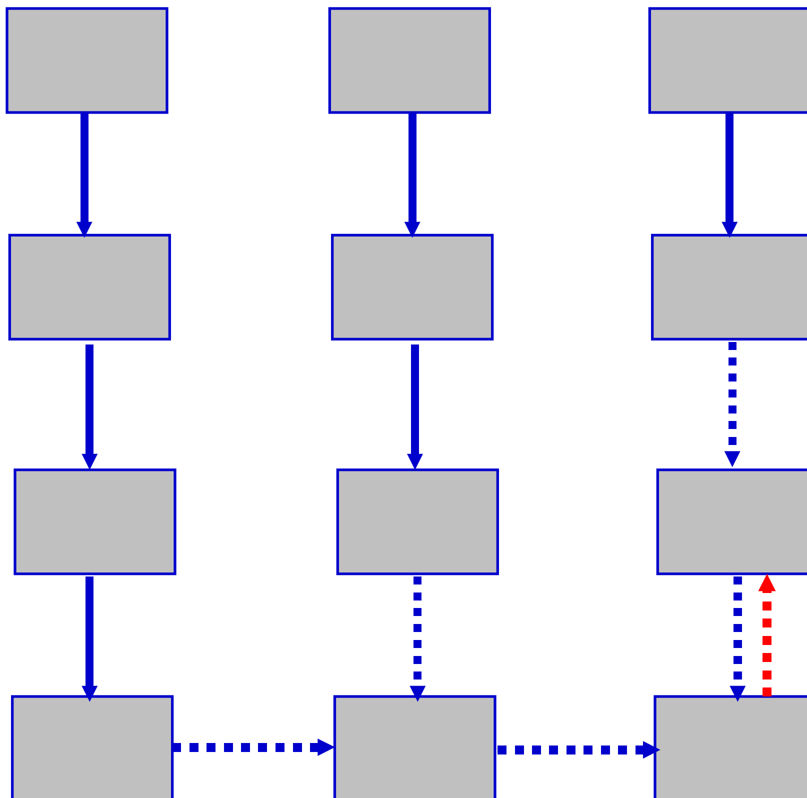
- Does not perform well in practice. Why?
  ```

Understanding the Behavior: Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

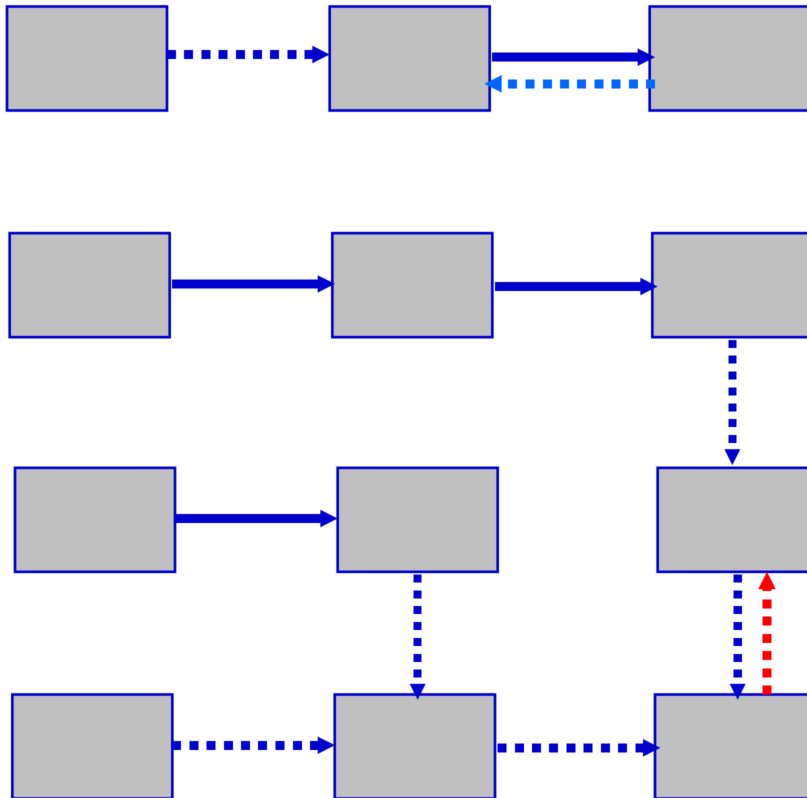
Mesh Exchange - Step 1

- Exchange data on a mesh



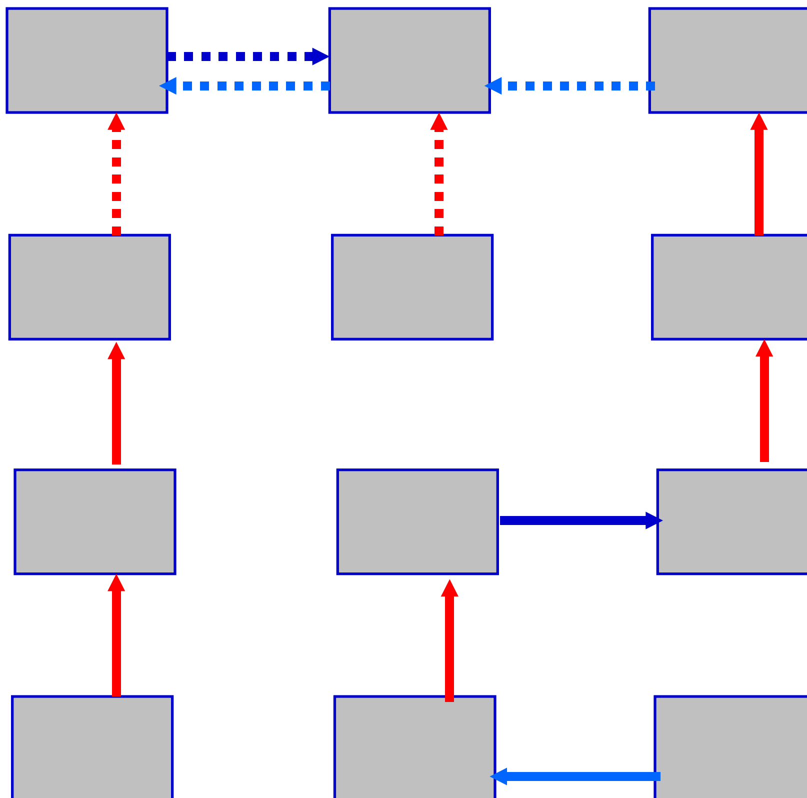
Mesh Exchange - Step 2

- Exchange data on a mesh



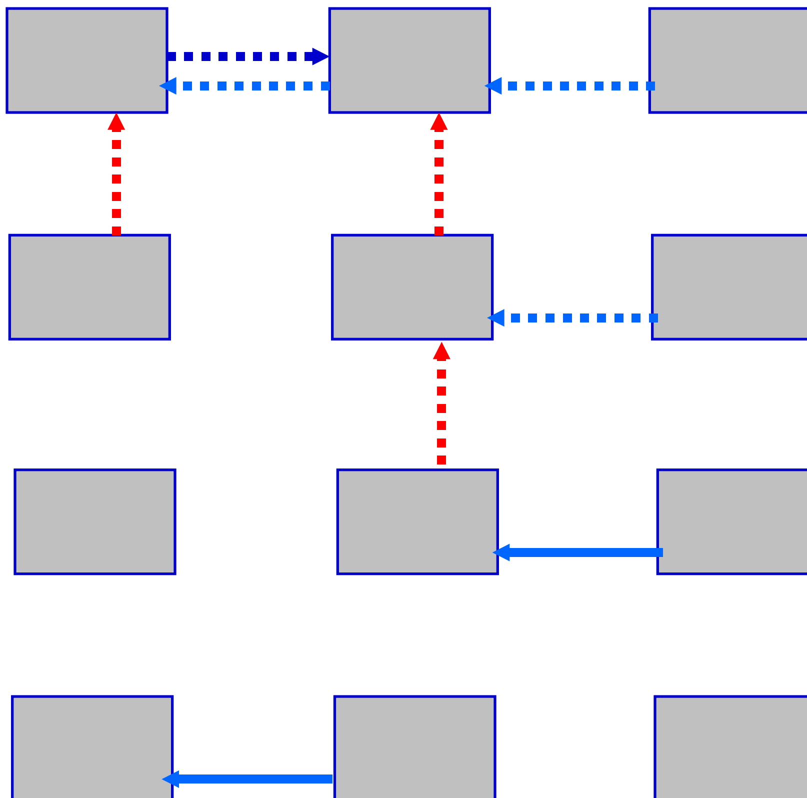
Mesh Exchange - Step 3

- Exchange data on a mesh



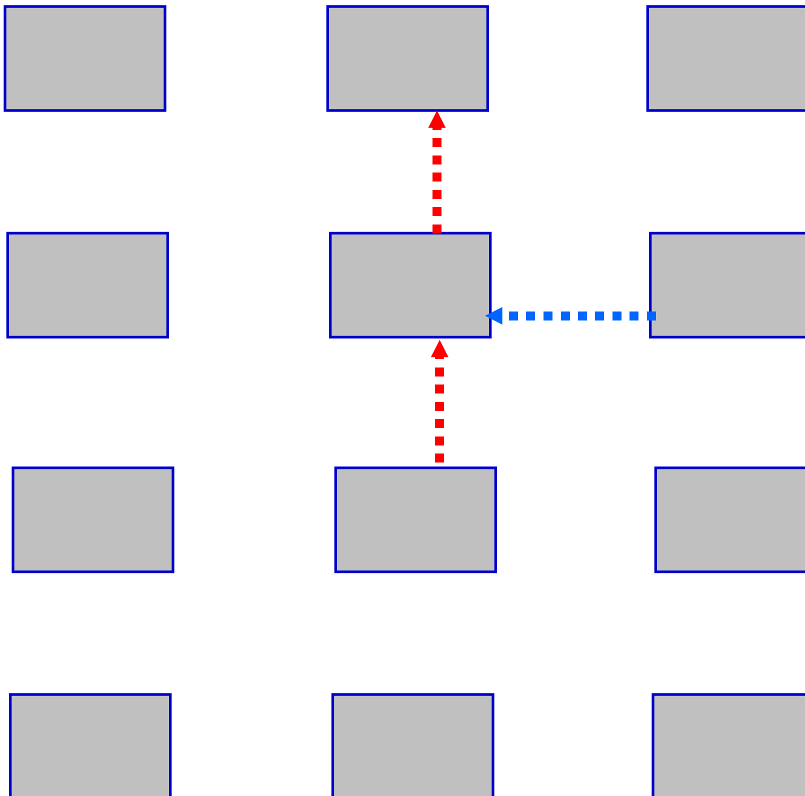
Mesh Exchange - Step 4

- Exchange data on a mesh



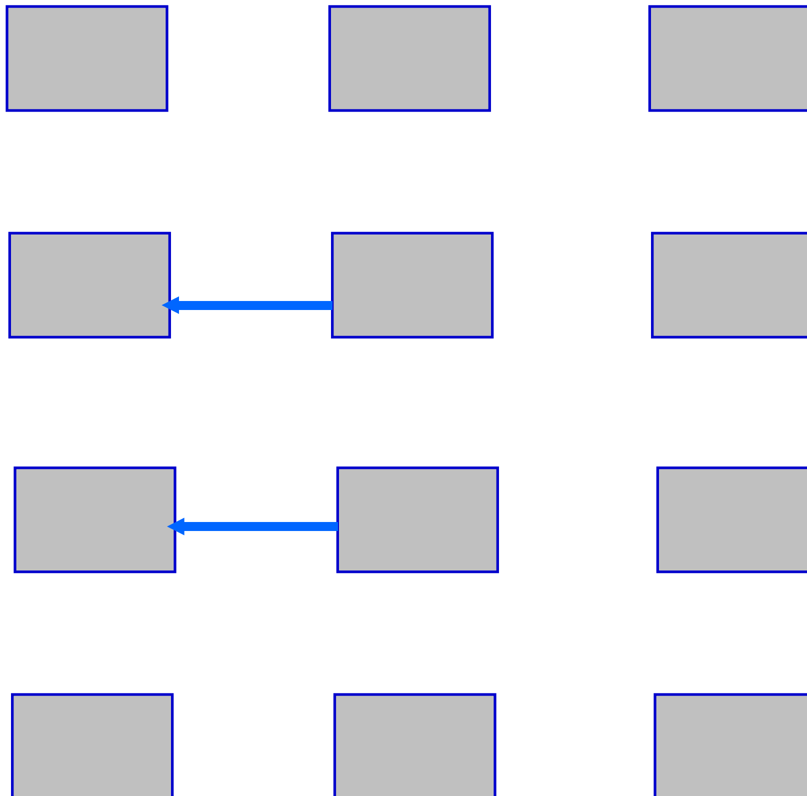
Mesh Exchange - Step 5

- Exchange data on a mesh

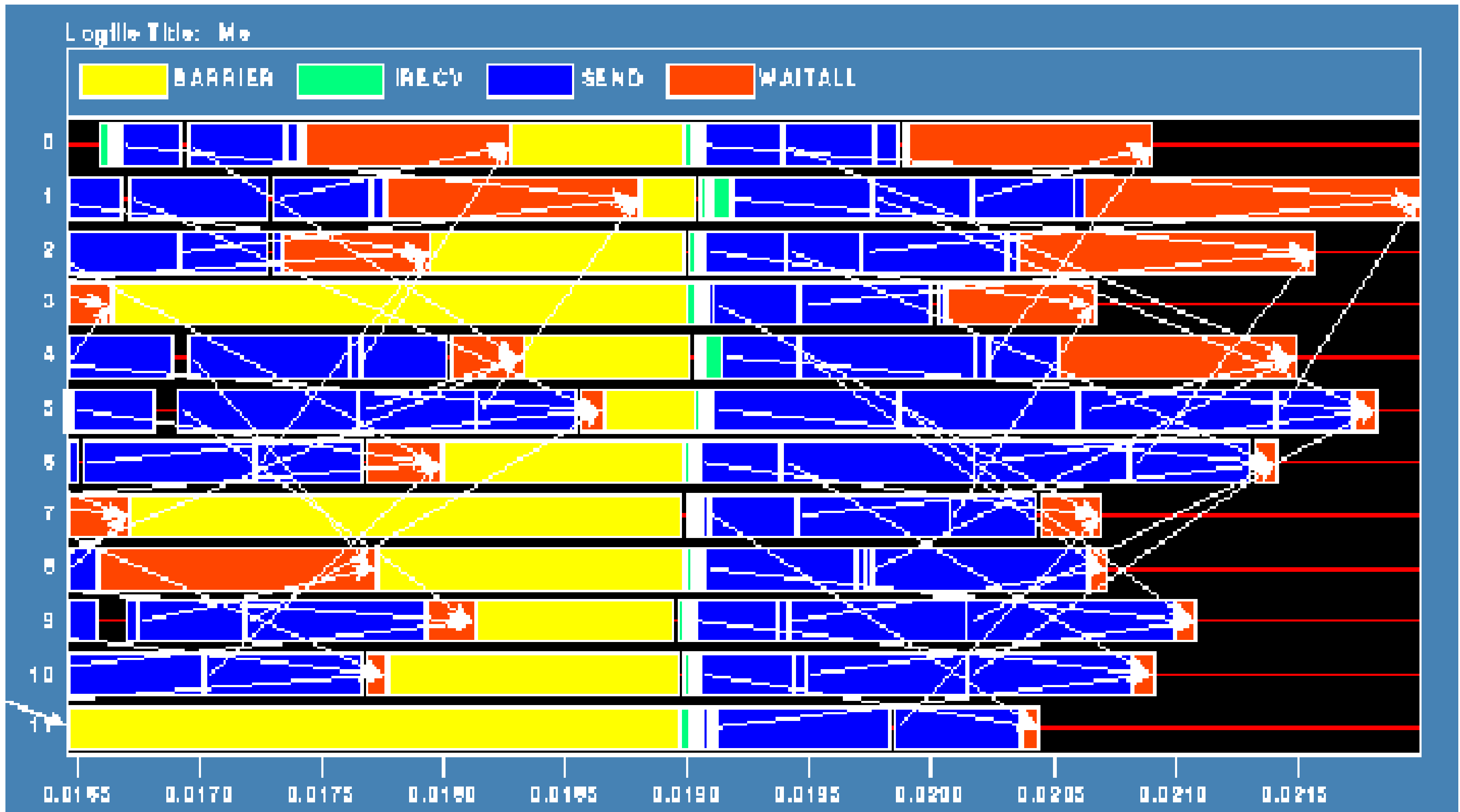


Mesh Exchange - Step 6

- Exchange data on a mesh

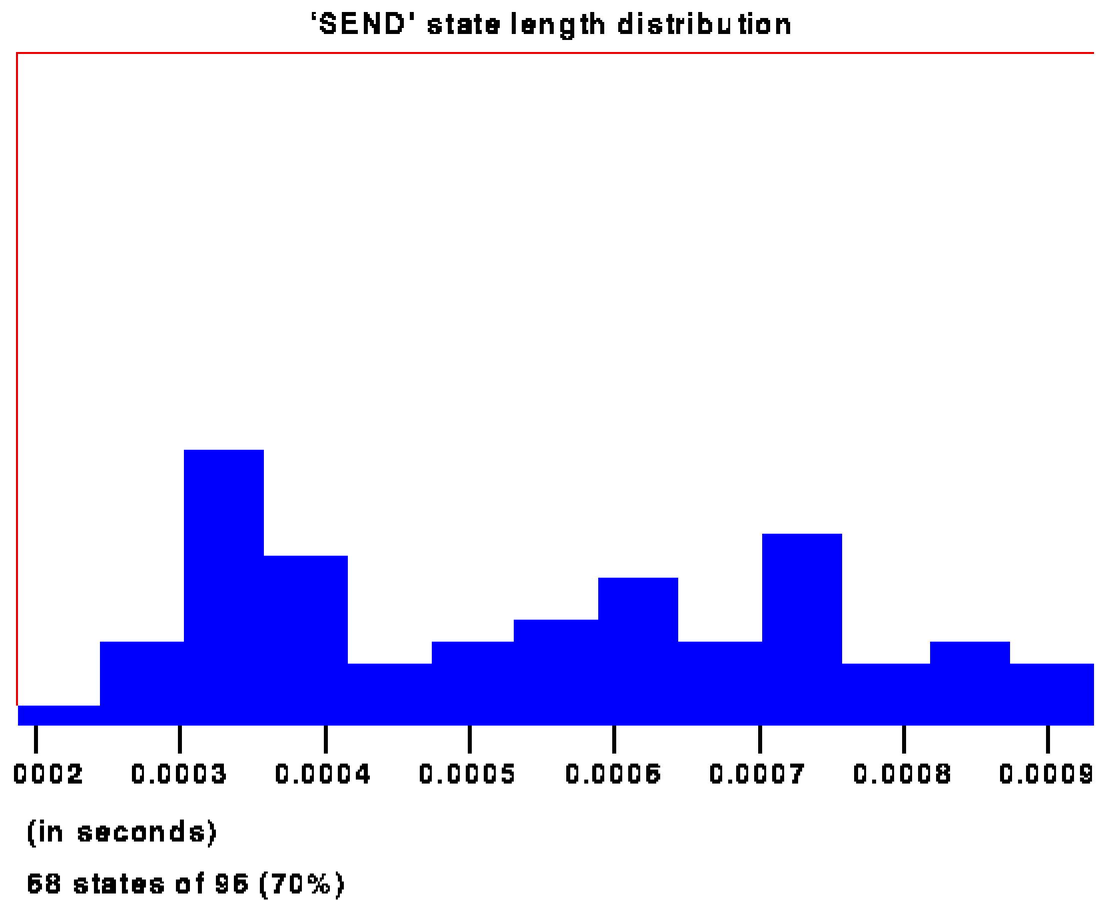


Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

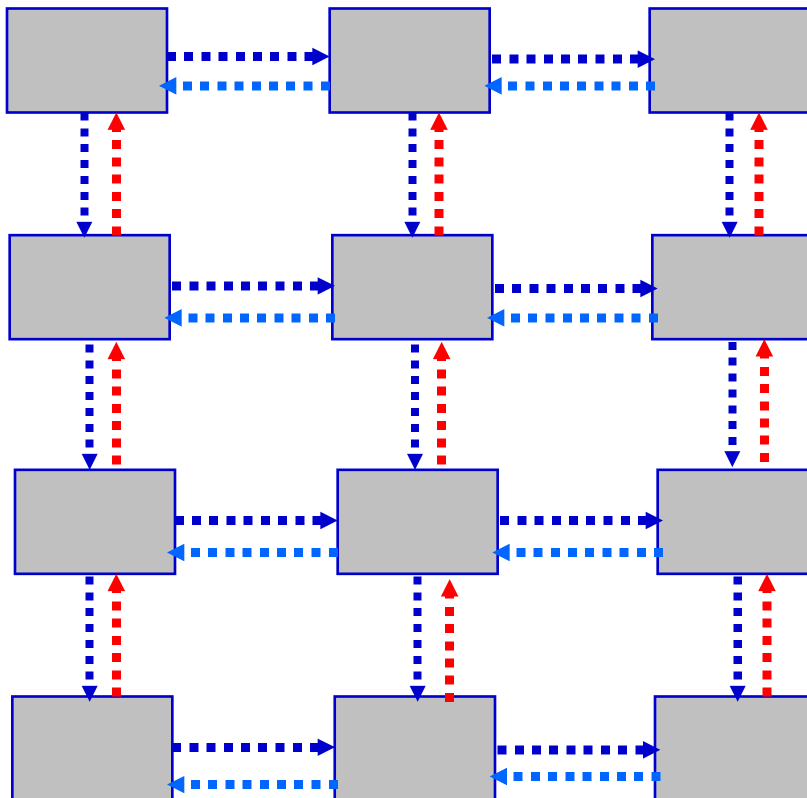
Fix 2: Use Isend and Irecv

- ```
  Do i=1,n_neighbors
    Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
                    comm, requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
    Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
                    comm, requests(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)
  ```

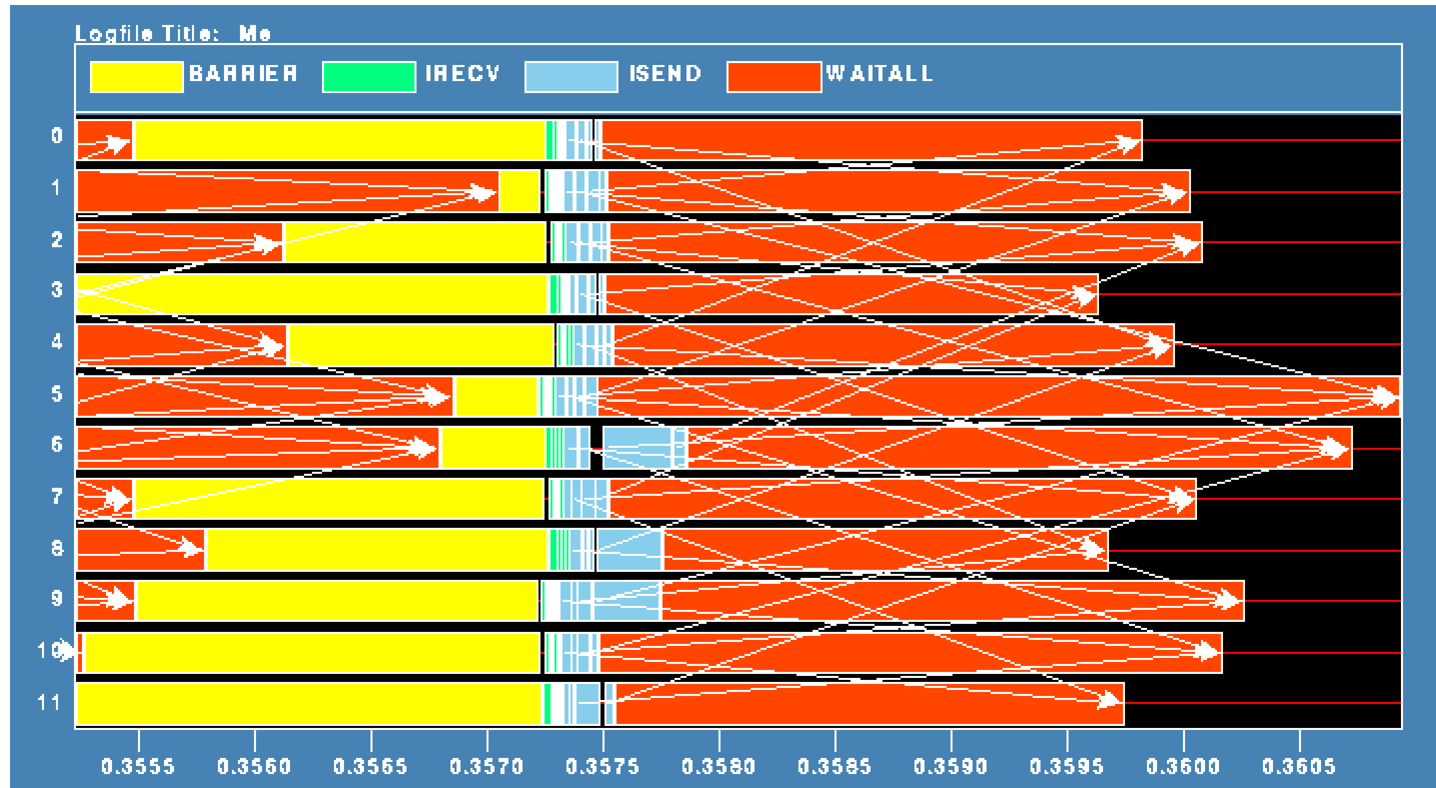
- (We'll see later how to do even better than this)
  ```

Mesh Exchange - Steps 1-4

- Four interleaved steps



Timeline from IBM SP



Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use nonblocking operations and MPI_Waitall to defer synchronization

MPI-2 Solution

- MPI-2 introduces one-sided operations
  - Put, Get, Accumulate
  - Separate data transfer from synchronization
- These are all nonblocking (blocking implies some synchronization)

One-sided Code

```
Do i=1,n_neighbors
   Call MPI_Get(inedge(1,i), len, MPI_REAL,   &
      nbr(i), edgedisp(i), len, MPI_REAL, win,&
      ierr)
Enddo
Call MPI_Win_fence( 0, win, ierr )
```

- MPI_Put may be preferable on some platforms
- Can avoid global synchronization (MPI_Win_fence) with MPI_Win_start/post/complete/wait
- Use MPI_Accumulate to move and add

Test Yourself: Deferred Synchronization

- Write code that has each processor send to all of the other processors. Use MPI_Irecv and MPI_Send. Compare:
  - All processors send to process 0, then process 1, etc., in that order
  - Each process sends to process (myrank+1), then (myrank+2), etc.
  - Compare with the MPI routine MPI_Alltoall
- If you have access to a shared-memory system, try the same thing using direct shared-memory copies (memcpy).

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)
  ```

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

Fairness in Message Passing

- What happens in this code:

```
if (rank .eq. 0) then
  do i=1,1000*(size-1)
    call MPI_Recv( a, n, MPI_INTEGER,&
        MPI_ANY_SOURCE, MPI_ANY_TAG, comm,&
        status, ierr )
    print *,' Received from',status(MPI_SOURCE)
  enddo
else
  do i=1,1000
    call MPI_Send( a, n, MPI_INTEGER, 0, i, &

                   comm, ierr )
  enddo
endif
```

- In what order are messages received?

Fairness

- MPI makes no guarantee, other than that all messages will be received *eventually*.
- The program could
  - Receive all from process 1, then all from process 2, etc.
  - That order would starve processes 2 and higher of work in a master/worker method
- How can we encourage or enforce fairness?

MPI Multiple Completion

- Provide one Irecv for each process:

```
do i=1,size-1
     call MPI_Irecv(…,i,… req(i), ierr)
enddo
```

- Process *all* completed receives (wait guarantees at least one):

```
call MPI_Waitsome( size-1, req, count,&
   array_of_indices, array_of_statuses, ierr )
do j=1,count
   ! Source of completed message is
    … array_of_statuses(MPI_SOURCE,j)
   ! Repost request
   call MPI_Irecv( …,req(array_of_indices(j)),)
enddo
```

MPI Datatypes

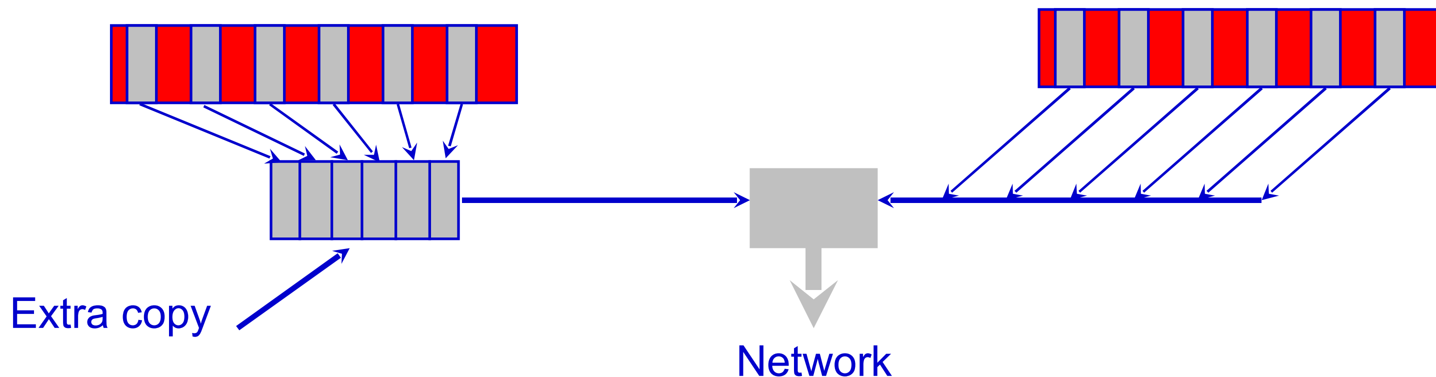
- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided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

Why Datatypes?

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

Non-contiguous Datatypes

- Provided to *allow* MPI implementations to avoid copy



  - Some MPI implementations can handle important special cases
  - Constant stride
  - Contiguous structures

Datatype Abstractions

- Standard Unix abstraction is "block of contiguous bytes" (e.g., readv, writev)
- MPI specifies datatypes recursively as
  - count of (type,offset)
    where offset may be relative or absolute
  - MPICH uses a simplified form of this for MPI_Type_vector (and so outperforms vendor MPIs)
- More general form would provide superior performance for most user-defined datatypes
  - MPI implementations can improve

Working With MPI Datatypes

- An MPI datatype defines a *type signature*:
  - sequence of pairs: (basic type,offset)
  - An integer at offset 0, followed by another integer at offset 8, followed by a double at offset 16 is
    - (integer,0), (integer,4), (double,16)
  - Offsets need not be increasing:
    - (integer,64),(double,0)
- An MPI datatype has an extent and a size
  - *size* is the number of bytes of the datatype
  - *extent* controls how a datatype is used with the *count* field in a send and similar MPI operations
  - extent is a misleading name

What does extent do?

- Consider
  `MPI_Send( buf, count, datatype, …)`
- What actually gets sent?
- MPI defines this as

```
do i=0,count-1
  MPI_Send(buf(1+i*extent(datatype)),1,
                datatype,…)
```

  (buf is a byte type like integer*1)
- *extent* is used to decide where to send from (or where to receive to in MPI_Recv) for count > 1
- Normally, this is right after the last byte used for (i-1)

Changing the extent

- MPI-1 provides two special types, MPI_LB and MPI_UB, for changing the extent of a datatype
  - This doesn't change the *size*, just how MPI decides what addresses in memory to use in offseting one datatype from another.
- Use MPI_Type_struct to create a new datatype from an old one with a different extent
  - Use MPI_Type_create_resized in MPI-2

Datatype Performance

Comparing MPI_Pack versus hand-coded packing

Test	Manual (MB/sec)	MPICH2 (%)	MPICH1 (%)	LAM (%)	Size (MB)	Extent (MB)
Contig	1,156.40	97.2	98.3	86.7	4	4
Struct Array	1,055.00	107.0	107.0	48.6	5.75	5.75
Vector	754.37	99.9	98.7	65.1	4	8
Struct Vector	746.04	100.0	4.9	19.0	4	8
Indexed	654.35	61.3	12.7	18.8	2	4
3D Face, XY	1,807.91	99.5	97.0	63.0	0.25	0.25
3D Face, XZ	1,244.52	99.5	97.3	79.8	0.25	63.75
3D Face, YZ	111.85	100.0	100.0	57.4	0.25	64

Sending Rows of a Matrix

- From Fortran, assume you want to send a row of the matrix
  A(n,m),
  that is, A(row,j), for j=1,…, m
- A(row,j) is not adjacent in memory to A(row,j+1)
- One solution: send each element separately:

```
Do j=1,m
   Call MPI_Send( A(row,j), 1, MPI_DOUBLE_PRECSION, …)
```

- Why not?

Fortran Array Layout

0	10	20							90
1									
2									
9	19								99

MPI Type vector

- Create a single datatype representing elements separated by a constant distance (*stride*) in memory
  - m items, separated by a stride of n:
  - call MPI_Type_vector(m, 1, n, &
            MPI_DOUBLE_PRECISION, newtype, ierr)
    call MPI_Type_commit(newtype, ierr)
  - Type_commit required before using a type in an MPI communication operation.
- Then send one instance of this type
  MPI_Send(a(row,1), 1, newtype, ….)

Test your understanding of Extent

- How do you send 2 rows of the matrix? Can you do this: MPI_Send(a(row,1),2,newtype,…)
- Hint: Extent(newtype) is distance from the first to last byte of the type
  - Last byte is a(row,m)
- Hint: What is the first location of A that is sent after the first row?

Sending with MPI_Vector

- Extent(newtype) is ((m-1)*n+1)*sizeof(double)
  - Last element sent is A(row,m)
- do i=0,1
  call MPI_Send(buf(1+i*extent(datatype)),1,&
  datatype,…)
  becomes
- call MPI_Send(A(row,1:m),…)   (i=0)
  call MPI_Send(A(row+1,m:2m-1),…) (i=1)
- The second step is *not*
  call MPI_Send(A(row+1,1:m),…)
- **Note:** Do not use A(row,1:m) in MPI programs; it is used here as a shorthand for A(row,k) for k=1,m

Send Two Rows

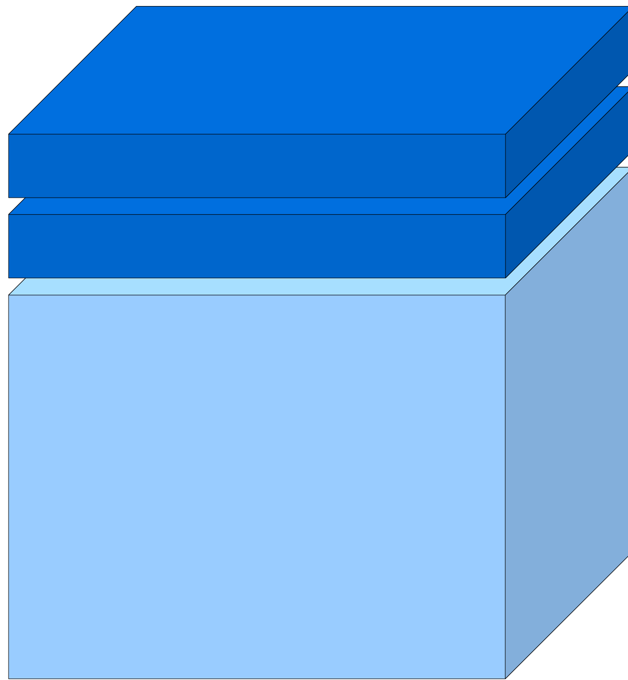
0	10	20							90
1									
2									
9	19								99

Solutions for Vectors

- MPI_Type_vector is for very specific uses
  - rarely makes sense to use count other than 1
- To send two rows, simply change the blockcount:

```
call MPI_Type_vector( m, 2, n, &
  MPI_DOUBLE_PRECISION, newtype, ierr )
```

- Stride is still relative to basic type



Striding Type

- Create a type with an extent of a column of the array:
  - ```
    types(1) = MPI_DOUBLE_PRECISION
    types(2) = MPI_UB
    displs(1) = 0
    displs(2) = n * 8   ! Bytes!
    blkcnt(1) = 1
    blkcnt(2) = 1
    call MPI_Type_struct( 2, blkcnt, displs, types,
                                   newtype, ierr )
    ```
- MPI_Send(A(i,2),m-1,newtype,…) sends the elements A(i,2:m)
    ```

Collective Operations in MPI

- Collective operations must be 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.

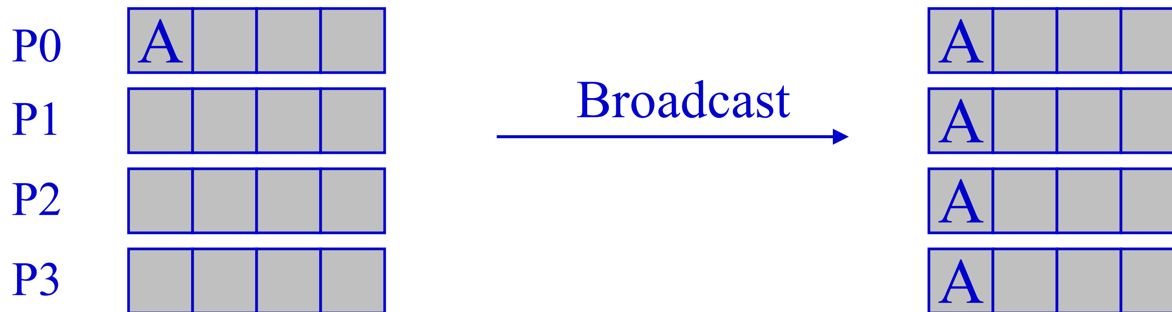
MPI Collective Communication

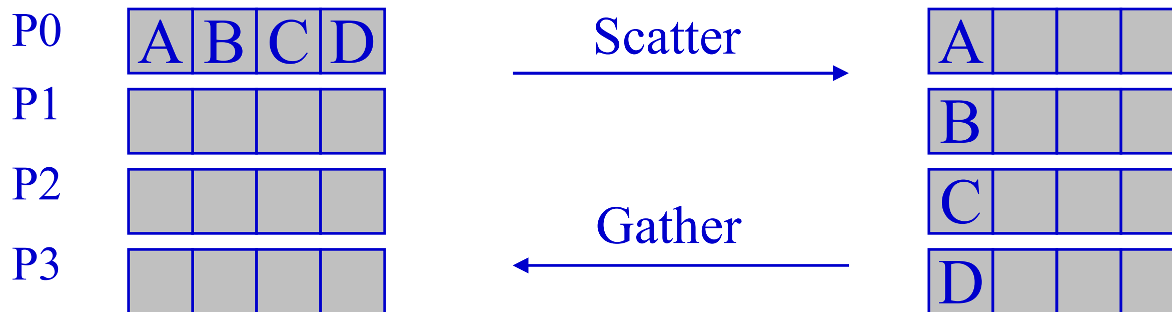
- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed “by hand” or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three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

Collective Data Movement



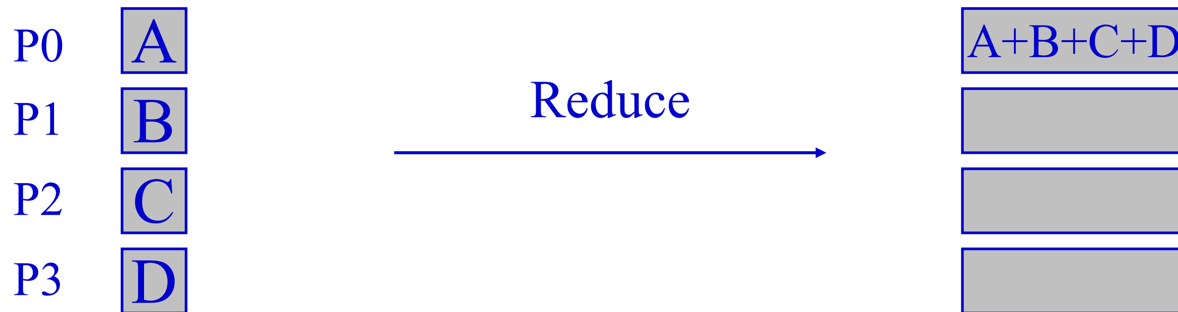


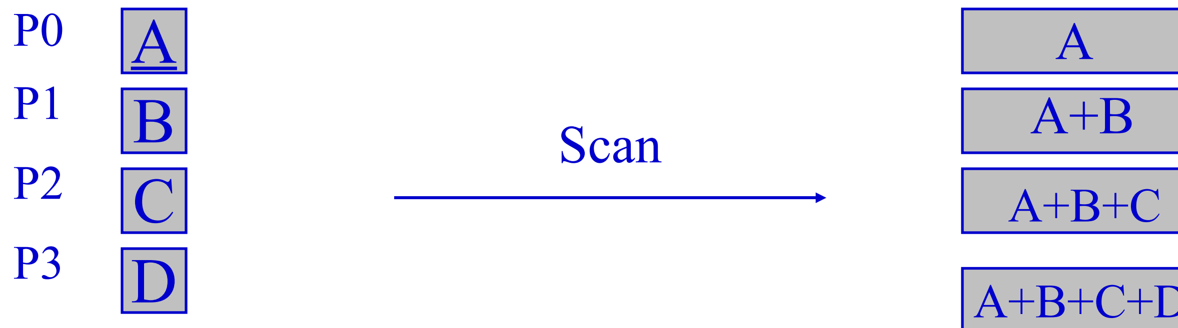
Comments on Broadcast

- 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`**, **`ReduceScatter`**, **`Scan`**, **`Scatter`**, **`Scatterv`**
- **`All`** versions deliver results to all participating processes.
- **`v`** versions allow the hunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`ReduceScatter`**, and **`Scan`** take both built-in and user-defined combiner functions.

When *not* to use Collective Operations

- Sequences of collective communication can be pipelined for better efficiency
- Example: Processor 0 reads data from a file and broadcasts it to all other processes.
  - Do i=1,m
    if (rank .eq. 0) read *, a
    call mpi_bcast(a, n, MPI_INTEGER, 0, comm, ierr)
    EndDo
  - Takes m n log p time.
- It can be done in (m+p) n time!

Pipeline the Messages

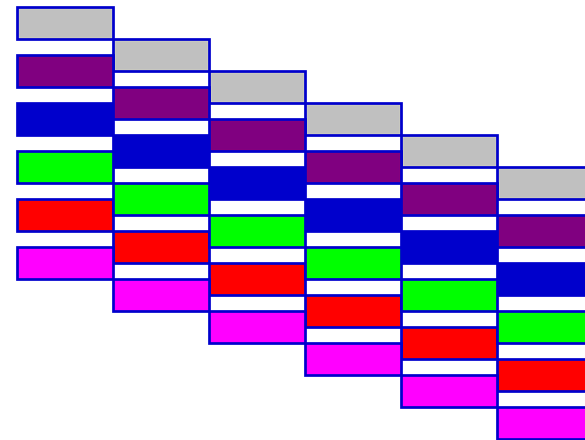
- Processor 0 reads data from a file and sends it to the next process. Other forward the data.

```
Do i=1,m
  if (rank .eq. 0) then
      read *, a
      call mpi_send(a, n, type, 1, 0, comm,ierr)
  else
      call mpi_recv(a,n,type,rank-1, 0,comm,status, ierr)
      call mpi_send(a,n,type,next, 0, comm,ierr)
  endif
EndDo
```

Concurrency Between Steps

- Broadcast:

- Pipeline



Each broadcast takes less time than pipeline version, but total time is longer

Another example of deferring synchronization

Notes on Pipelining Example

- Use `MPI_File_read_all` instead
  - Even more optimizations possible
    - Multiple disk reads
    - Pipeline the individual reads
    - Block transfers

- "Elegance" of collective routines can lead to synchronization
  - and hence a performance penalty

Persistent Operations

- Many applications use the same communications operations over and over
- Same parameters used many time
  - Do i=1,n
    ```
    call MPI_Isend(…)
    call MPI_Irecv(…)
    call MPI_Waitall(…)
    ```
- MPI provides *persistent* operations to make this more efficient
  - Reduce error checking of args (needed only once)
  - Implementation may be able to make special provision for repetitive operation (though none do to date)
  - All persistent operations are nonblocking

Persistent Operations and Networks

- Current trend in networks is “user mode” or “OS bypass”
  - Provides direct communication between designated user-buffers without OS intervention
- Requires registration of memory with OS; may be a limited resource
  - Examples are IB, LAPI
- MPI’s persistent operations are a good match to this capability

Using Persistent Operations

- Replace

  MPI_Isend(buf, count, datatype, tag, dest, comm, &request)

  with

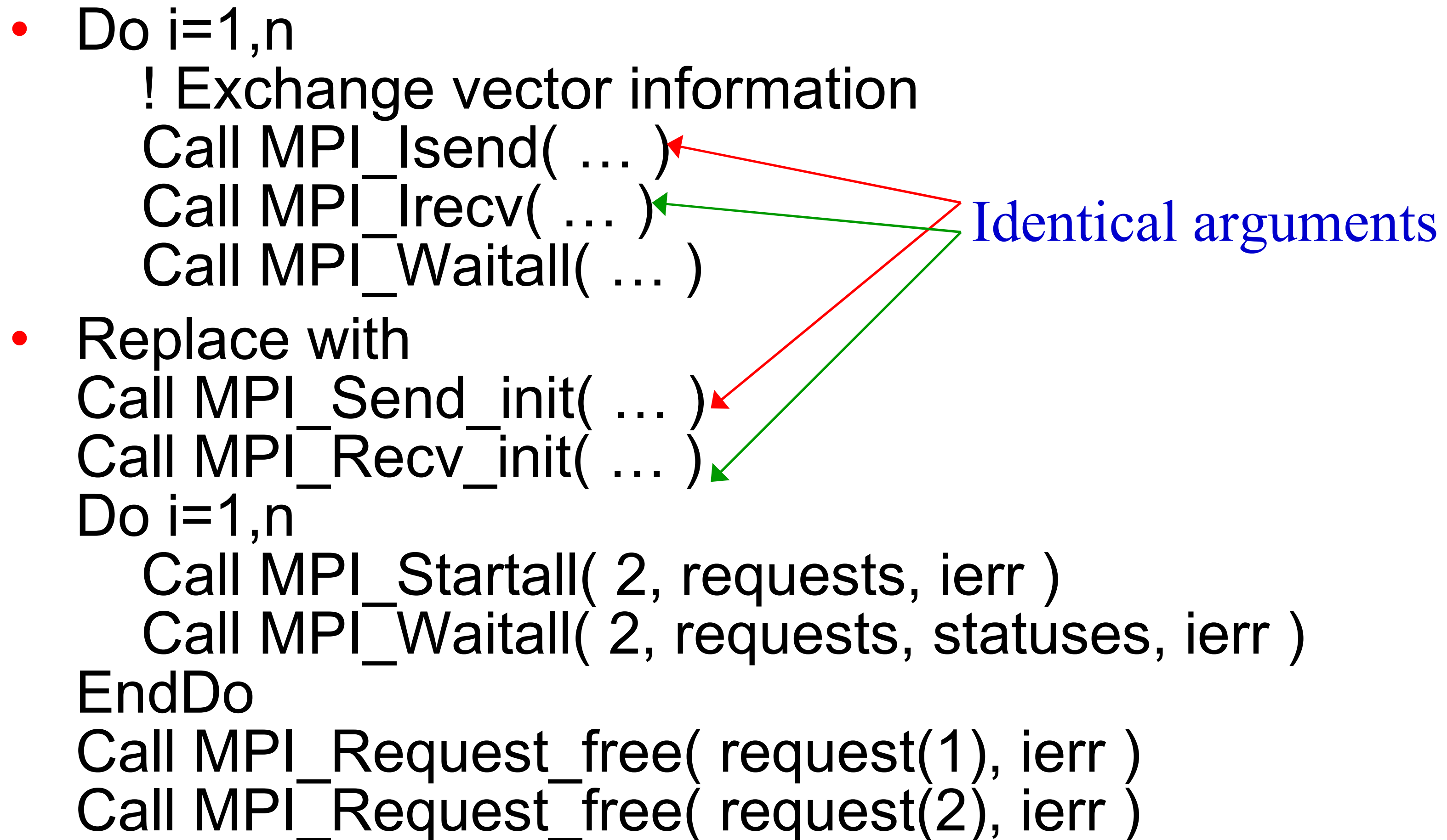
  MPI_Send_init(buf, count, datatype, tag, dest, comm, &request)
  MPI_Start(request)

- MPI_Irecv with MPI_Recv_init, MPI_Irsend with MPI_Rsend_init, etc.
- Wait/test requests just like other nonblocking requests
- Free requests when done with MPI_Request_free

Example: Sparse Matrix-Vector Product

- Many iterative methods require matrix-vector products
- Same operation (with same arguments) performed many times (vector updated in place)
- Divide sparse matrix into groups of rows by process: e.g., rows 1-10 on process 0, 11-20 on process 1. Use same division for vector.
- To perform matrix-vector product, get elements of vector on different processes with Irecv/Isend/Waitall

Changing MPI Nonblocking to MPI Persistent

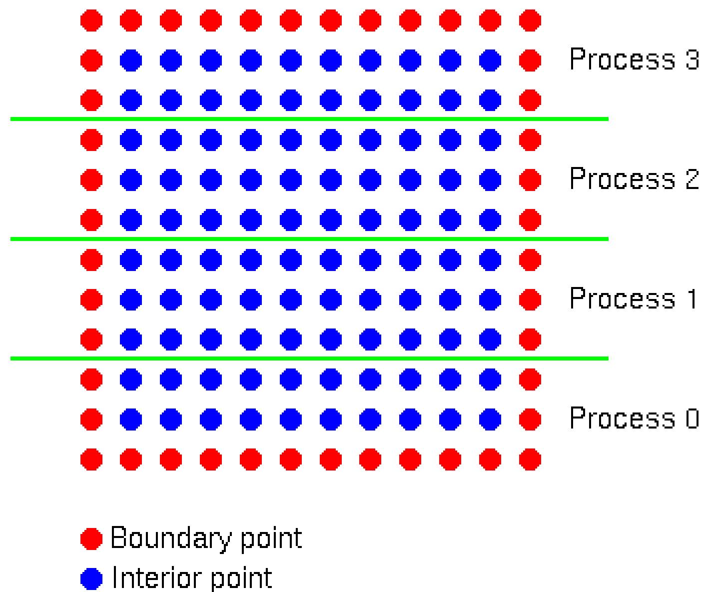


Experiments with MPI Implementations

- 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 (C Ordering)

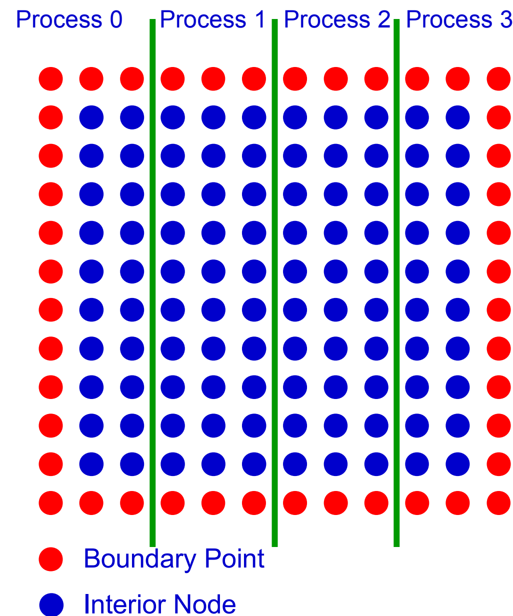
- Simple parallel data structure



- Processes exchange rows with neighbors

Jacobi Iteration (Fortran Ordering)

- Simple parallel data structure



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

Background to Tests

- Goals
  - Identify better performing idioms for the same communication operation
  - Understand these by understanding the underlying MPI process
  - Provide a starting point for evaluating additional options (there are many ways to write even simple codes)

Some Send/Receive Approaches

- Based on operation hypothesis. Most of these are for polling mode. Each of the following is a *hypothesis* that the experiments test
  - Better to start receives first
  - Ensure recvs posted before sends
  - Ordered (no overlap)
  - Nonblocking operations, overlap effective
  - Use of Ssend, Rsend versions
  - Persistent operations

Scheduling Communications

- Is it better to use MPI_Waitall or to schedule/ order the requests?
  - Does the implementation complete a Waitall in any order or does it prefer requests as ordered in the array of requests?
- In principle, it should always be best to let MPI schedule the operations. In practice, it may be better to order either the short or long messages first, depending on how data is transferred.

Some Example Results

- Summarize some different approaches

Send and Recv (C)

- Simplest use of send and recv

```
{
   MPI_Status status;
   MPI_Comm ring_comm = mesh->ring_comm;

   /* Send up, then receive from below */
   MPI_Send( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
            ring_comm );
   MPI_Recv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &status );
   /* Send down, then receive from above */
   MPI_Send( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );
   MPI_Recv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
            ring_comm, &status );
}
```

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 (C)

- Irecv, Isend, Waitall - ok performance

```
MPI_Status statuses[4];
MPI_Comm ring_comm;
MPI_Request r[4];

    /* Send up, then receive from below */
    MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &r[1] );
    MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
               ring_comm, &r[3] );
    MPI_Isend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
               ring_comm, &r[0] );
    /* Send down, then receive from above */
    MPI_Isend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm,
               &r[2] );
    MPI_Waitall( 4, r, statuses );
}
```

Ensure recvs posted before sends (C)

- Irecv, Sendrecv/Barrier, Rsend, Waitall

```
void ExchangeInit( mesh )
Mesh *mesh;
{
    MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm,
               &mesh->rq[0] );
    MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
              ring_comm, &mesh->rq[1] );
}

void Exchange( mesh )
Mesh *mesh;
{
    MPI_Status statuses[2];

    /* Send up and down, then receive */
    MPI_Rsend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
               ring_comm );
    MPI_Rsend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );

    MPI_Waitall( 2, mesh->rq, statuses );
}

void ExchangeEnd( mesh )
Mesh *mesh;
{
    MPI_Cancel( &mesh->rq[0] );
    MPI_Cancel( &mesh->rq[1] );
}
```

Use of Ssend versions (C)

- Ssend allows send to wait until receive ready
  - At least one implementation (T3D) gives better performance for Ssend than for Send

```
void Exchange( mesh )
Mesh *mesh;
{
   MPI_Status  status;

   /* Send up, then receive from below */
   MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &rq );
   MPI_Ssend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
              ring_comm );
   MPI_Wait( &rq, &status );
   /* Send down, then receive from above */
   MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
              ring_comm, &rq );
   MPI_Ssend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );
   MPI_Wait( &rq, &status );
}
```

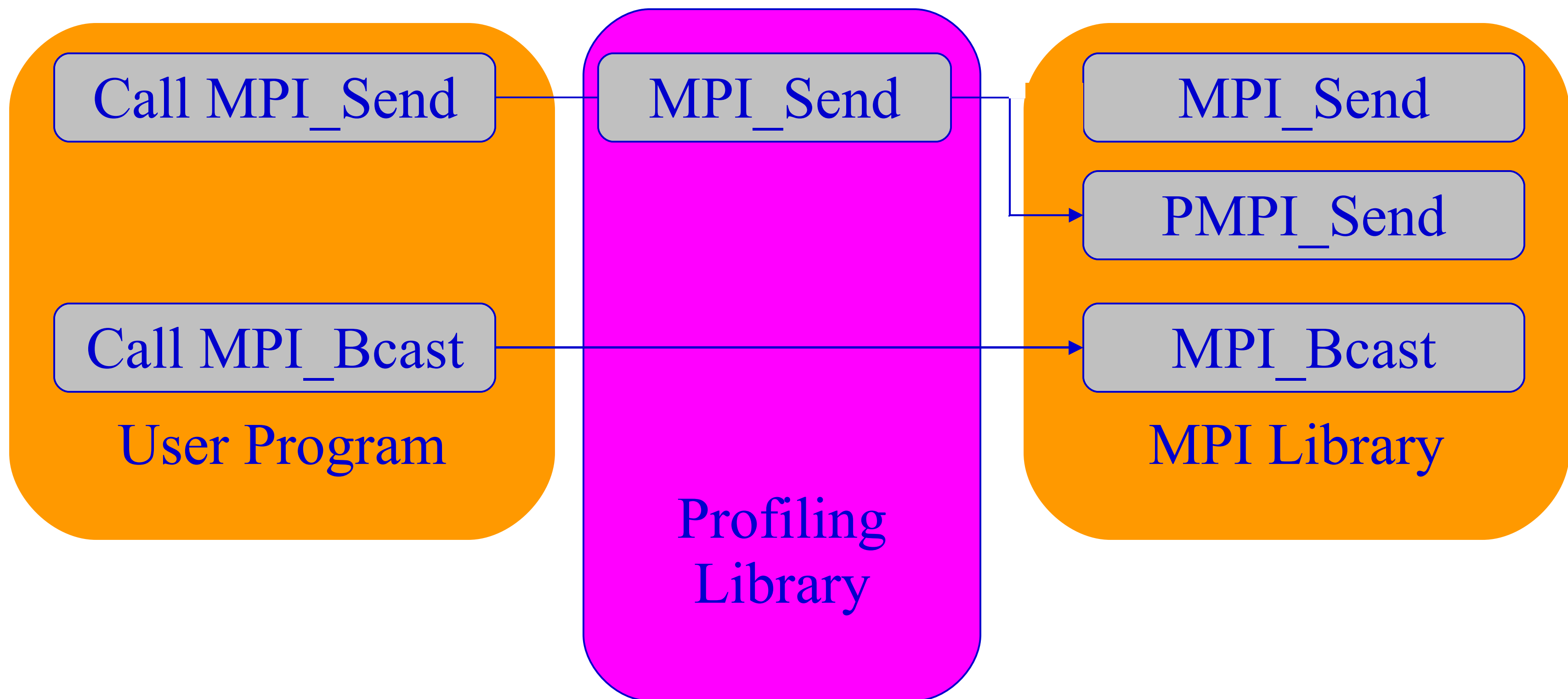
Summary of Results

- For short messages (eager protocol), better to post receives before sends
- For long messages, better to start sends before receives
  - Most implementations use rendezvous protocols for long messages (Cray, IBM, SGI, MPICH)
- Synchronous sends better on T3D
  - otherwise system buffers
- MPI_Rsend can offer some performance gain on IBM SP
  - as long as receives can be guaranteed *without* extra messages

MPI's Profiling Interface: PMPI

- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- Some libraries already make use of PMPI

Profiling Interface



Using the Profiling Interface From C

```
static int nsend = 0;

int MPI_Send( void *start, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm )
{
    nsend++;
    return PMPI_send(start, count, datatype,
                       dest, tag, comm);
}
```

Using the Profiling Interface from Fortran

```
Block data
common /mycounters/ nsend
data nsend/0/
end

subroutine MPI_Send( start, count, datatype, dest,&
                     tag, comm, ierr )
integer start(*), count, datatype, dest, tag, comm
common /mycounters/ nsend
save   /mycounters/
nsend = nsend + 1
call PMPI_send(start, count, datatype, &
                   dest, tag, comm, ierr )
end
```

Test Yourself: Find Unsafe Uses of MPI_Send

- Assume that you have a debugger that will tell you where a program is stopped (most will). How can you find unsafe uses of MPI_Send (calls that assume that data will be buffered) by running the program *without* making assumptions about the amount of buffering
  - Hint: Use MPI_Ssend

Finding Unsafe uses of MPI_Send

```
subroutine MPI_Send( start, count, datatype, dest,
                     tag, comm, ierr )
integer start(*), count, datatype, dest, tag, comm
call PMPI_Ssend(start, count, datatype,
                   dest, tag, comm, ierr )
end
```

- MPI_Ssend will not complete until the matching receive starts
- MPI_Send can be implemented as MPI_Ssend
- At some value of *count*, MPI_Send will act like MPI_Ssend (or fail)

Finding Unsafe Uses of MPI_Send II

- Have the application generate a message about unsafe uses of MPI_Send
  - Hint: use MPI_Issend
  - C users can use __FILE__ and __LINE__
    - sometimes possible in Fortran (.F files)

Reporting on Unsafe MPI_Send

```
subroutine MPI_Send( start, count, datatype, dest, tag, comm,&
                     ierr )
integer start(*), count, datatype, dest, tag, comm
include 'mpif.h'
integer request, status(MPI_STATUS_SIZE)
double precision tend, delay
parameter (delay=10.0d0)
logical flag

call PMPI_Issend(start, count, datatype, dest, tag, comm, &
                 request, ierr )
flag = .false.
tend   = MPI_Wtime()+ delay
Do while (.not. flag .and. t1 .gt. MPI_Wtime())
    call PMPI_Test( request, flag, status, ierr )
Enddo
if (.not. flag) then
    print *, 'MPI_Send appears to be hanging'
    call MPI_Abort( MPI_COMM_WORLD, 1, ierr )
endif
end
```

Logging and Visualization Tools

- Jumpshot, and MPE tools
  http://www.mcs.anl.gov/research/projects/mpich2
- Intel Trace Analyzer and Collector
  http://www.intel.com/software/products/cluster
- Paradyn
  http://www.cs.wisc.edu/paradyn/
- Many other vendor tools exist

Contexts (hidden in communicators)

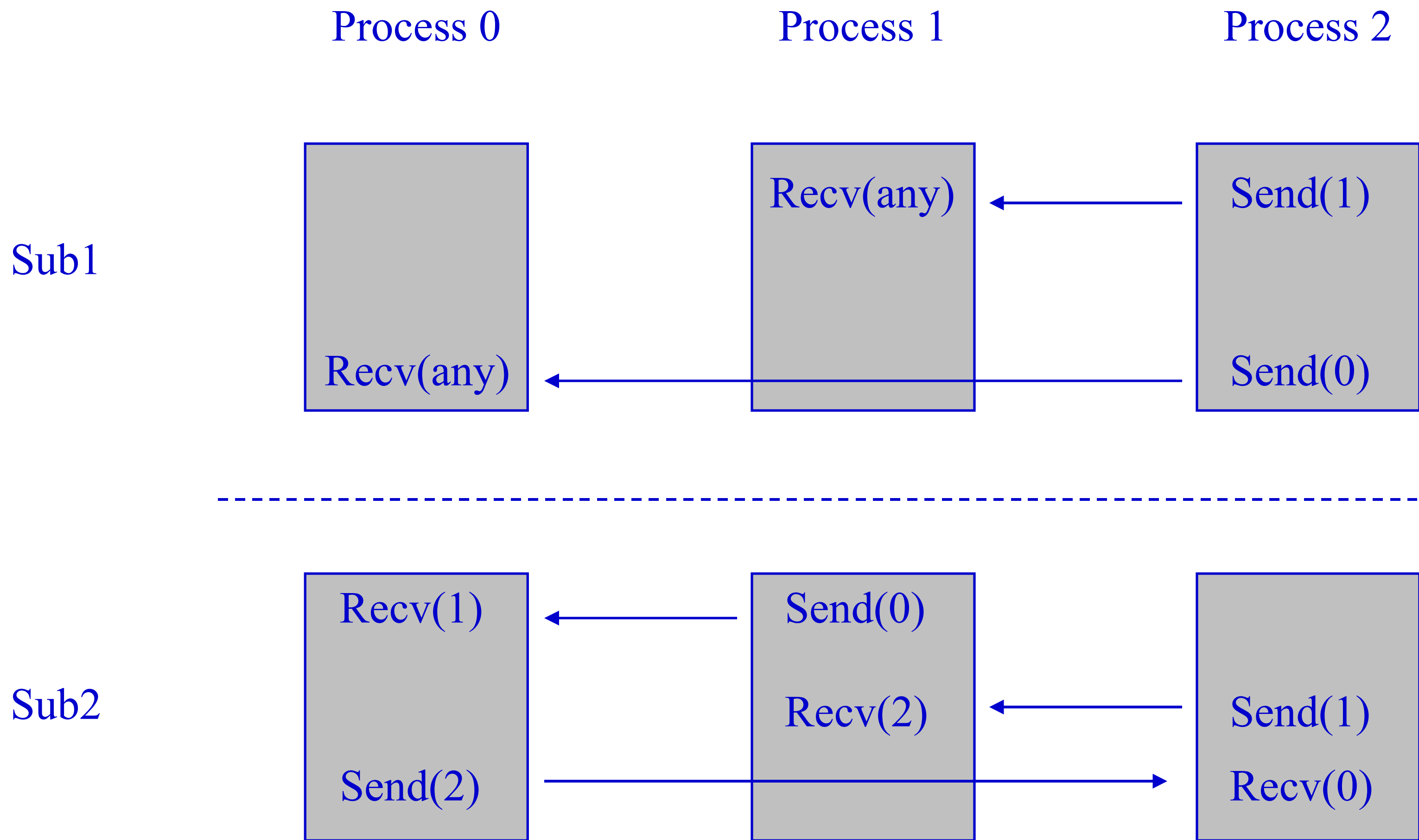
- Parallel libraries require isolation of messages from one another and from the user that cannot be adequately handled by tags.
- The context hidden in a communicator provides this isolation
- The following examples are due to Marc Snir.
  - Sub1 and Sub2 are from different libraries

```
Sub1();
Sub2();
```

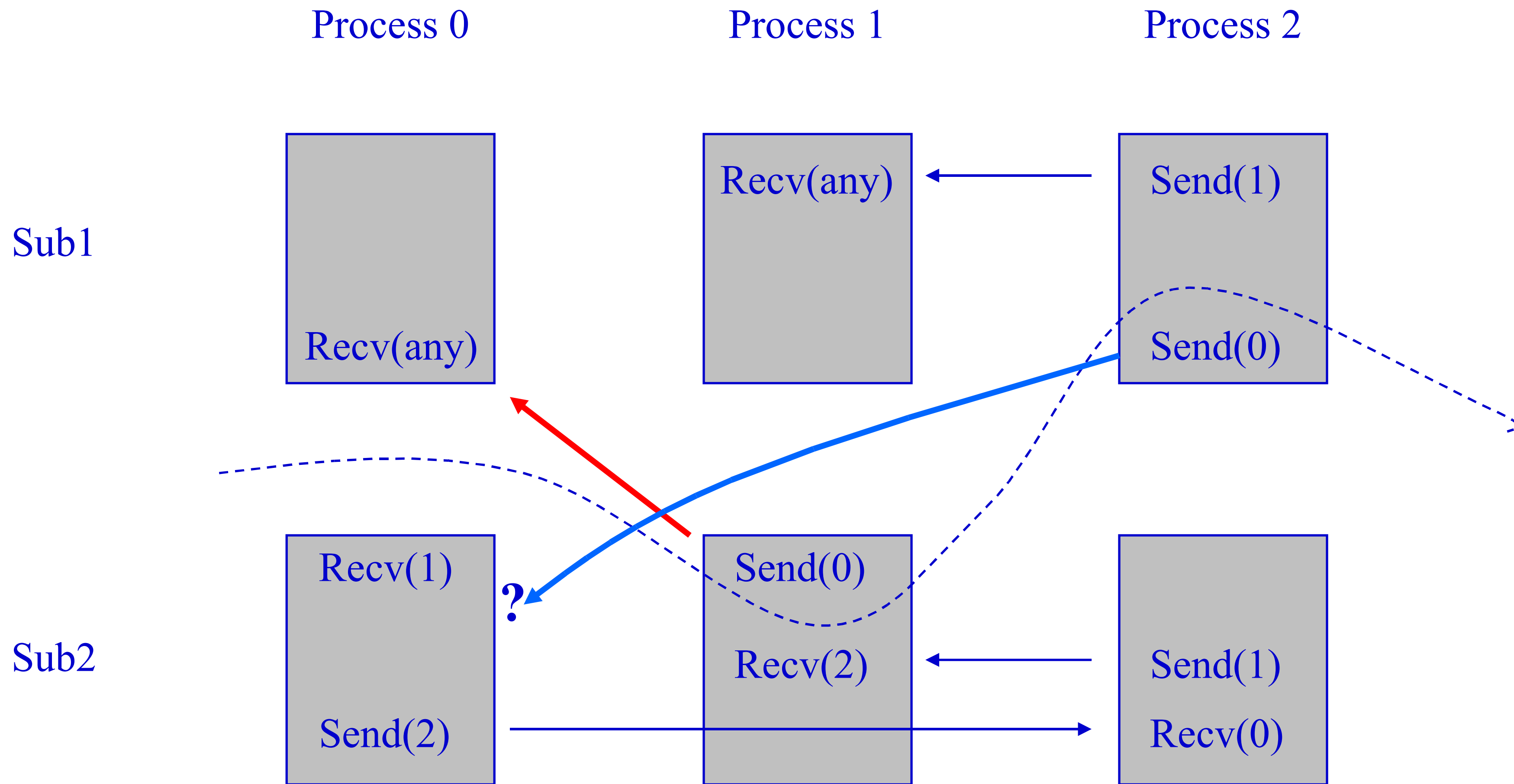
  - Sub1a and Sub1b are from the same library

```
Sub1a();
Sub2();
Sub1b();
```

Correct Execution of Library Calls

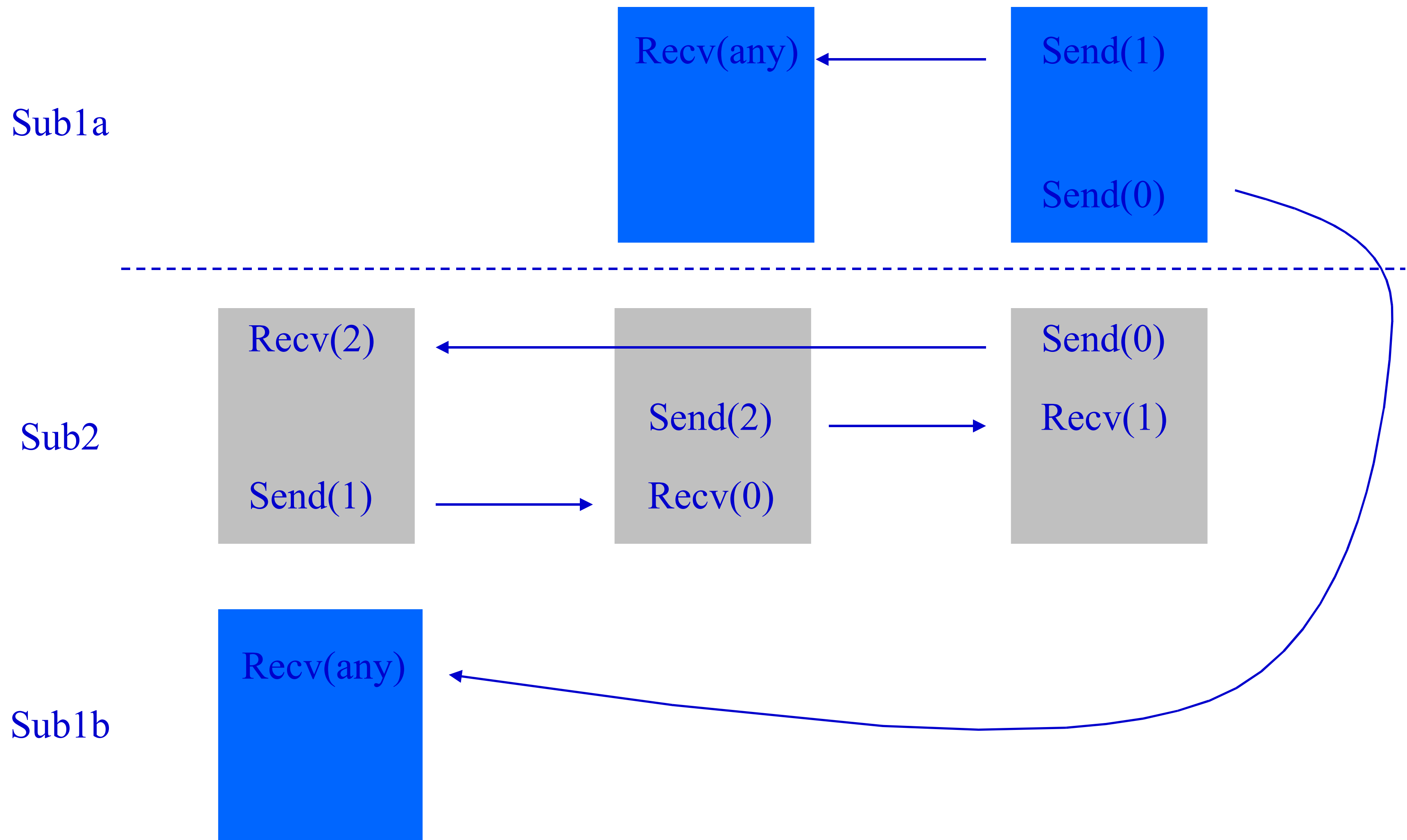


Incorrect Execution of Library Calls



Program hangs (Recv(1) never satisfied)

Correct Execution of Library Calls with Pending Communication



Incorrect Execution of Library Calls with Pending Communication

Sub1a

Recv(any)

Send(1)

Send(0)

Sub2

Recv(2)

Send(1)

Send(2)

Recv(0)

Send(0)

Recv(1)

Sub1b

Recv(any)

Program Runs—but with wrong data!

MPI-2

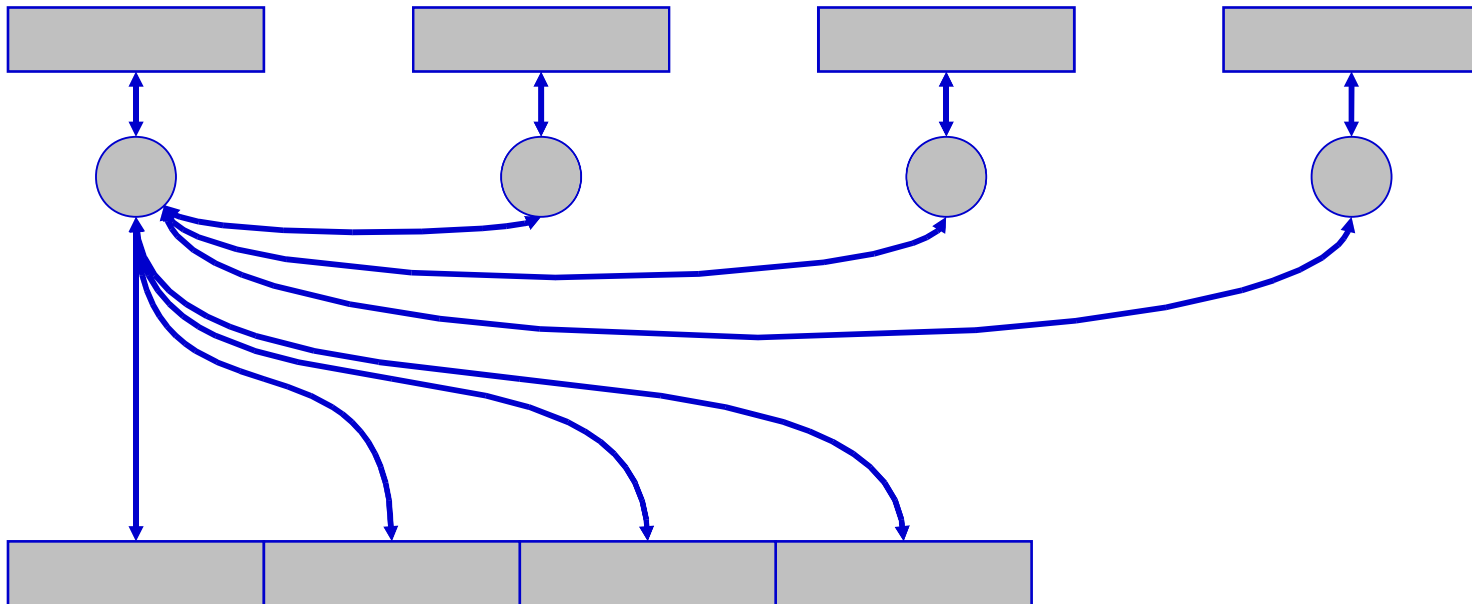
- Same process of definition by MPI Forum
- MPI-2 is an extension of MPI
  - Extends the message-passing *model*.
    - Parallel I/O
    - Remote memory operations (one-sided)
    - Dynamic process management
  - Adds other functionality
    - C++ and Fortran 90 (2003) bindings
    - External interfaces
    - Language interoperability
    - MPI interaction with threads
- Implementations appeared more slowly
  - Japanese Supercomputer vendors had first complete implementations.
  - MPICH now nearly complete
  - Other vendors have various parts, esp. I/O. Many have RMA. Dynamic is more difficult for integrated systems

MPI-2 Implementation Status

- MPICH2 (nearly) complete
- LAM has major parts of dynamic, I/O, and one-sided
- IBM has everything but dynamic chapter
- Cray X-1 has MPI-1, one-sided
- SGI has one-sided
- HP, Sun ?

Parallel I/O

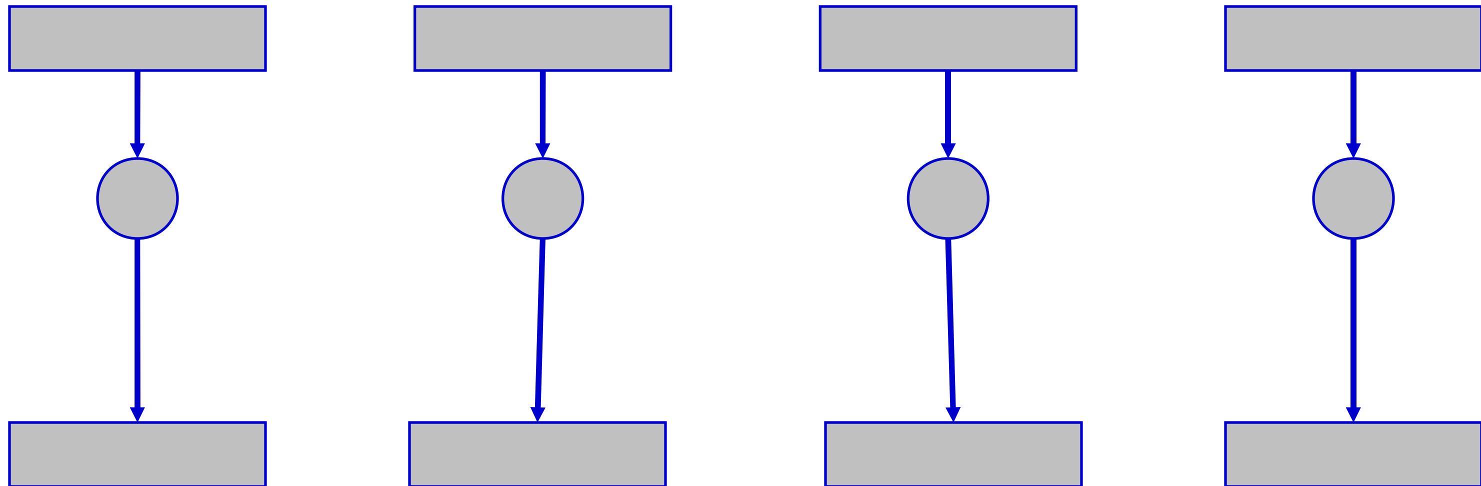
Non-Parallel I/O



- Non-parallel
- Performance worse than sequential
- Legacy from before application was parallelized

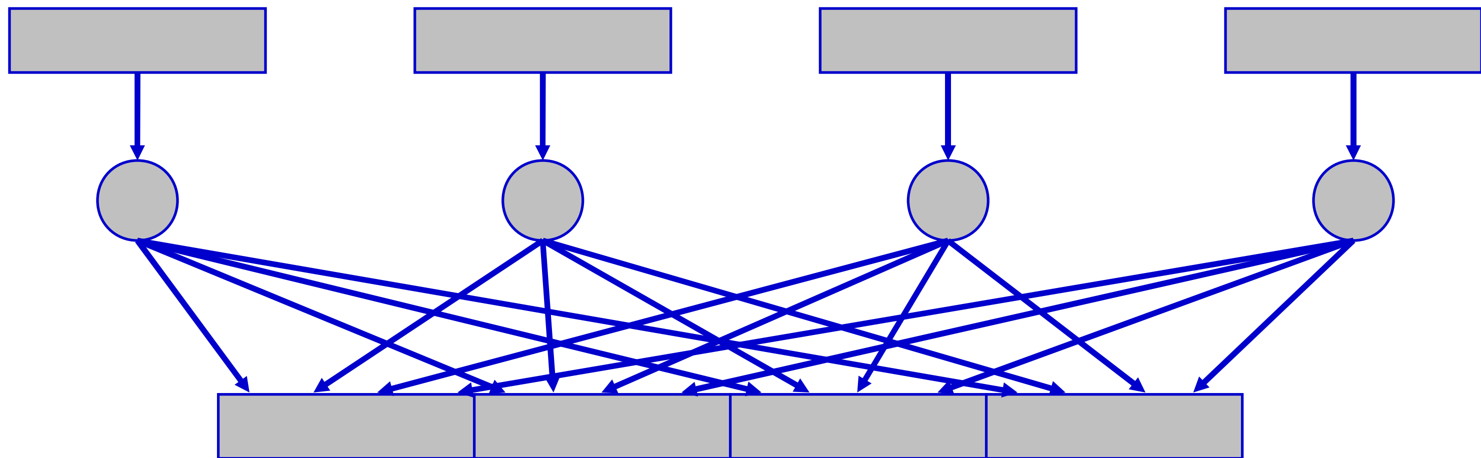
Independent Parallel I/O

- Each process writes to a separate file



- Pro: parallelism
- Con: lots of small files to manage
- Legacy from before MPI

Cooperative Parallel I/O



- Parallelism
- Can only be expressed in MPI
- Natural once you get used to it

Why MPI is a Good 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

MPI Versions of Unix I/O

- Unix

```
FILE myfile;
myfile = fopen(...)

fread(...)
fwrite(...)

fclose(...)
```

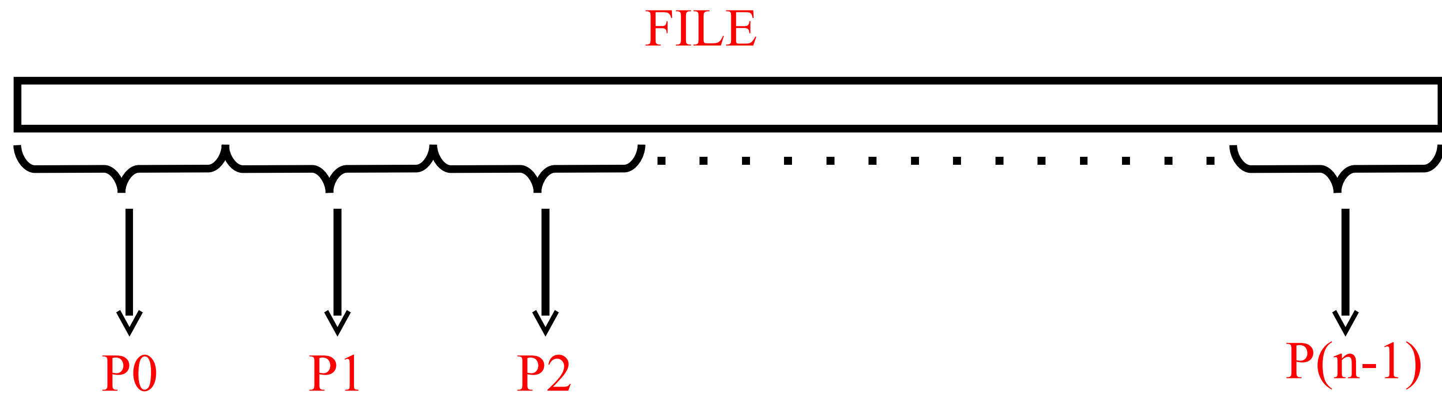
- MPI

```
MPI_File myfile;
MPI_File_open(...)
  takes info, comm args

MPI_File_read/write(...)
  take (addr, count,
  datatype)

MPI_File_close(...)
```

Using MPI for Simple I/O



Each process needs to read a chunk of data from a common file

Using Individual File Pointers (C)

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

Using Individual File Pointers (Fortran)

```
integer fh, status(MPI_STATUS_SIZE)

call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr )
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr )

bufsize = FILESIZE/nprocs
nints = bufsize/sizeof(int)

call MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
           MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
call MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET,
                   ierr )
call MPI_File_read(fh, buf, nints, MPI_INTEGER, &
                   status, ierr )
call MPI_File_close( fh, ierr )
```

Using Explicit Offsets (C)

```
#include "mpi.h"
MPI_Status status;
MPI_File fh;
MPI_Offset offset;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*sizeof(int));
offset = rank * nints * sizeof(int);
MPI_File_read_at(fh, offset, buf, nints, MPI_INT,
                 &status);
MPI_Get_count(&status, MPI_INT, &count );
printf( "process %d read %d ints\n", rank, count );

MPI_File_close(&fh);
```

Using Explicit Offsets (Fortran)

```
  include 'mpif.h'

  integer status(MPI_STATUS_SIZE)
  integer (kind=MPI_OFFSET_KIND) offset
C in F77, see implementation notes (might be integer*8)

  call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
            MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
  nints = FILESIZE / (nprocs*INTSIZE)
  offset = rank * nints * INTSIZE
  call MPI_FILE_READ_AT(fh, offset, buf, nints,
                      MPI_INTEGER, status, ierr)
  call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
  print *, 'process ', rank, 'read ', count, 'integers'

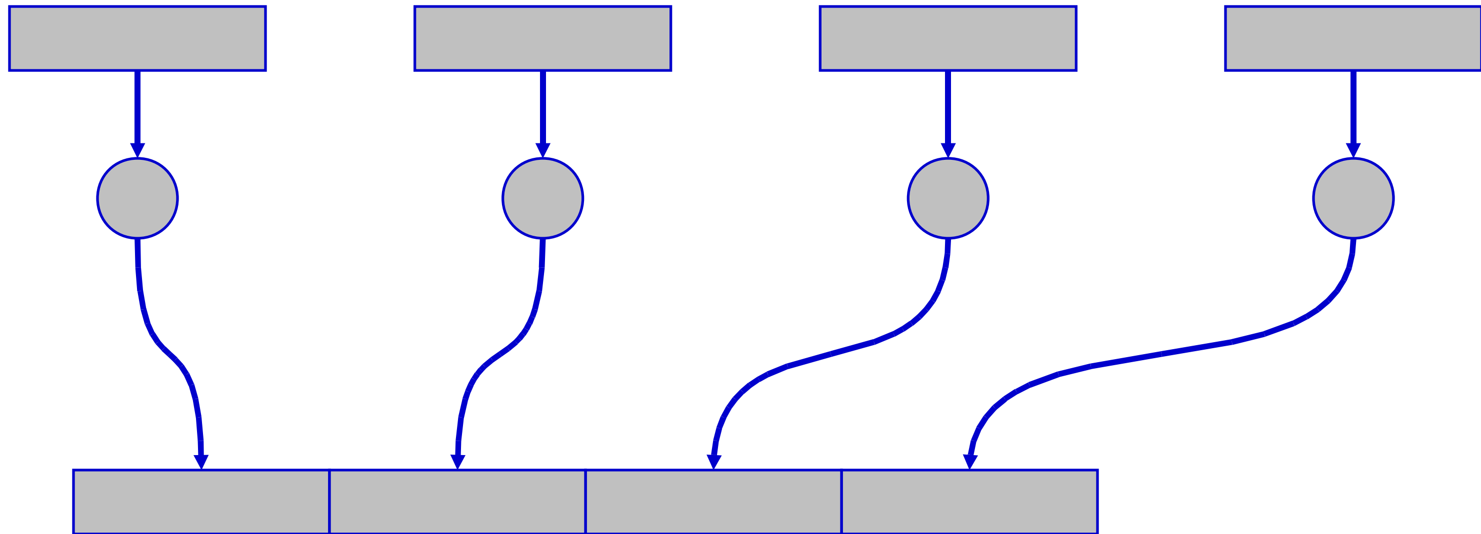
  call MPI_FILE_CLOSE(fh, ierr)
```

Writing to a File

- Use **`MPI_File_write`** or **`MPI_File_write_at`**
- Use **`MPI_MODE_WRONLY`** or **`MPI_MODE_RDWR`** as the flags to **`MPI_File_open`**
- If the file doesn't exist previously, the flag **`MPI_MODE_CREATE`** must also be passed to **`MPI_File_open`**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

Using File Views

- Processes write to shared file



- **`MPI_File_set_view`** assigns regions of the file to separate processes

File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

File View Example (C)

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                  MPI_INT, MPI_INT, "native",
                  MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

File View Example (Fortran)

```
PROGRAM main

use mpi

integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
do i = 0, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo


* in F77, see implementation notes (might be integer*8)
```

File View Example (Fortran) contd.

```
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
                   MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                   MPI_INFO_NULL, thefile, ierr)
call MPI_TYPE_SIZE(MPI_INTEGER, intsize)
disp = myrank * BUFSIZE * intsize
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
                       MPI_INTEGER, 'native', &
                       MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, &
                       MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)

END PROGRAM main
```

Ways to Access a Shared File

- `MPI_File_seek`
- `MPI_File_read`
- `MPI_File_write`

like Unix I/O

- `MPI_File_read_at`
- `MPI_File_write_at`

combine seek and I/O for thread safety

- `MPI_File_seek_shared`
- `MPI_File_read_shared`
- `MPI_File_write_shared`

use shared file pointer

Ways to Access a Shared File Collectively

- **`MPI_File_read_all`**
- **`MPI_File_write_all`**

- **`MPI_File_read_at_all`**
- **`MPI_File_write_at_all`**

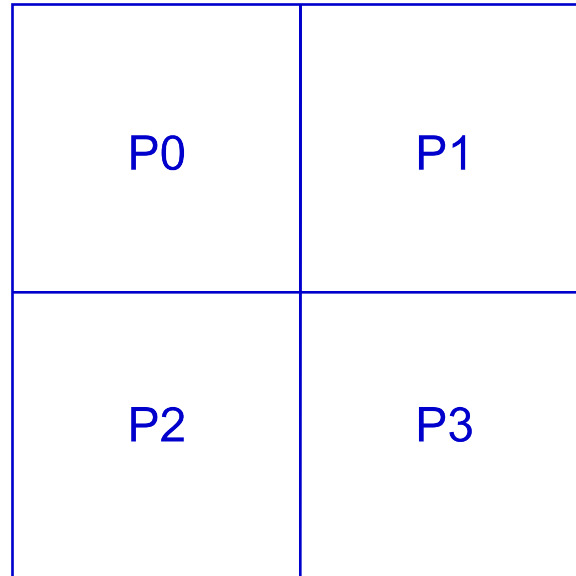
- **`MPI_File_read_shared_ordered`**
- **`MPI_File_write_shared_ordered`**

Noncontiguous Accesses

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory and file within a single function call by using derived datatypes
- Allows the MPI implementation to optimize the access
- Collective I/O combined with noncontiguous accesses yields the highest performance.

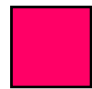
Example: Distributed Array Access

2D array distributed among four processes



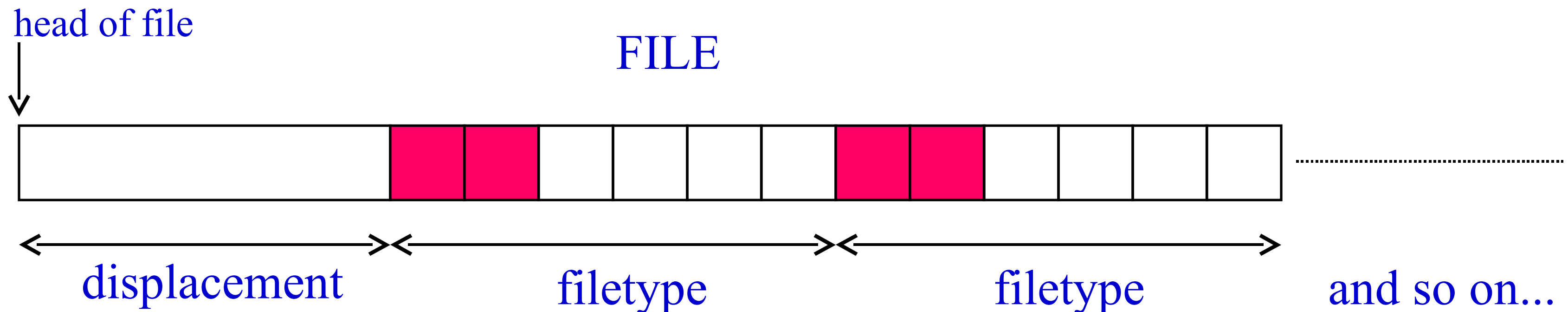
File containing the global array in row-major order

A Simple Noncontiguous File View Example



etype = MPI_INT

filetype = two MPI_INTs followed by a gap of four MPI_INTs



Noncontiguous File View Code

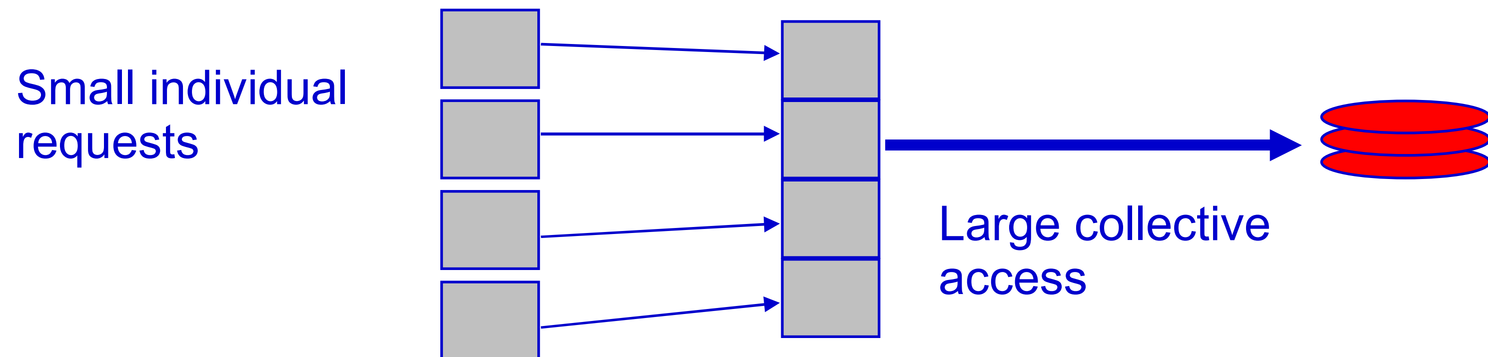
```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Collective I/O in MPI

- A critical optimization in parallel I/O
- Allows communication of “big picture” to file system
- Framework for two-phase I/O, in which communication precedes I/O (can use MPI machinery)
- Basic idea: build large blocks, so that reads/writes in I/O system will be large



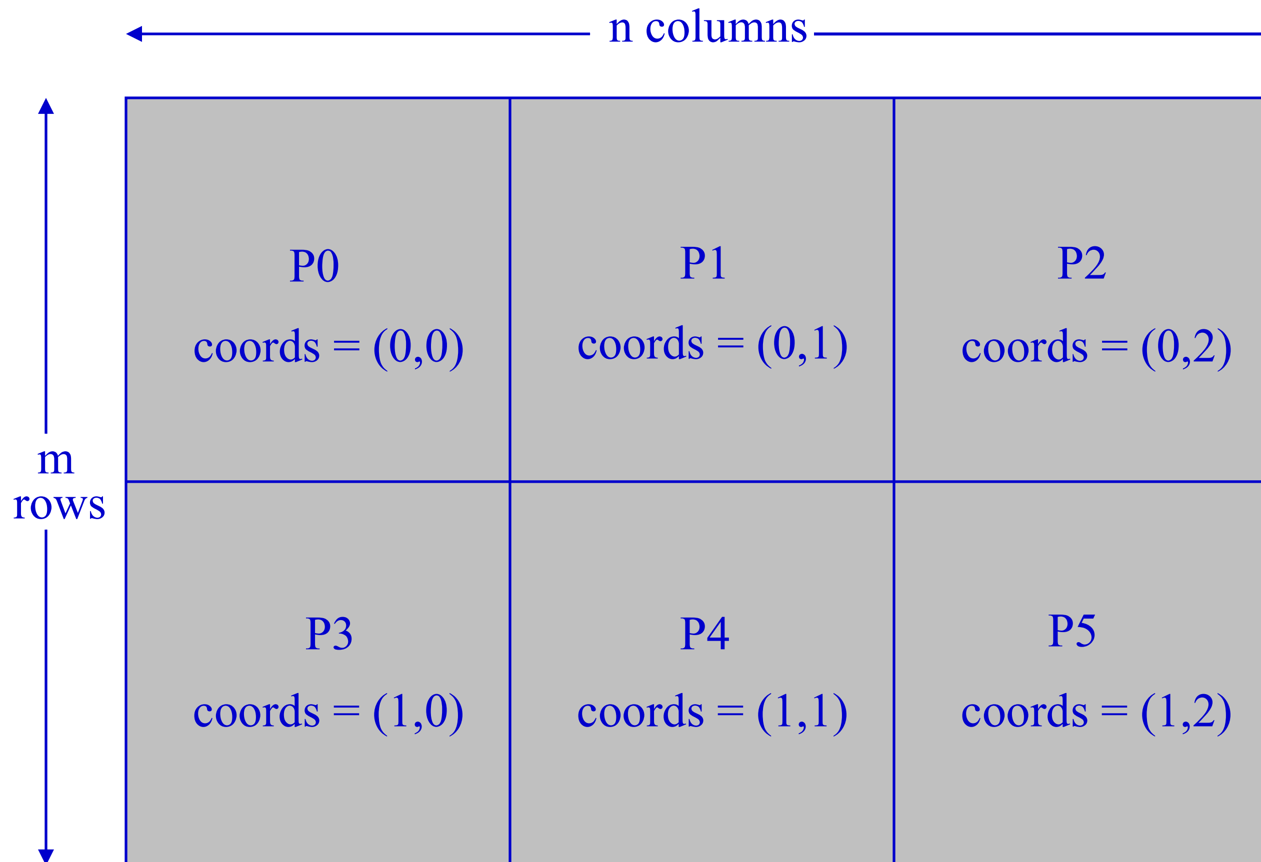
Collective I/O Functions

- **`MPI_File_read_all`**, **`MPI_File_read_at_all`**, etc
- **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
- Each process specifies only its own access information -- the argument list is the same as for the non-collective functions

Collective I/O

- By calling the collective I/O functions, the user allows an implementation to optimize the request based on the combined request of all processes
- The implementation can merge the requests of different processes and service the merged request efficiently
- Particularly effective when the accesses of different processes are noncontiguous and interleaved

Accessing Arrays Stored in Files



nproc(1) = 2, nproc(2) = 3

Using the "Distributed Array" (Darray) Datatype

```
int gsizes[2], distribs[2], dargs[2], psizes[2];

gsizes[0] = m;    /* no. of rows in global array */
gsizes[1] = n;    /* no. of columns in global array*/

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;

psizes[0] = 2; /* no. of processes in vertical dimension
                  of process grid */
psizes[1] = 3; /* no. of processes in horizontal dimension
                  of process grid */
```

Darray Continued

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
               psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                  MPI_INFO_NULL);

local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,
               MPI_FLOAT, &status);

MPI_File_close(&fh);
```

A Word of Warning about Darray

- The darray datatype assumes a very specific definition of data distribution -- the exact definition as in HPF
- For example, if the array size is not divisible by the number of processes, darray calculates the block size using a *ceiling* division (20 / 6 = 4)
- darray assumes a row-major ordering of processes in the logical grid, as assumed by cartesian process topologies in MPI-1
- If your application uses a different definition for data distribution or logical grid ordering, you cannot use darray. Use subarray instead.

Using the Subarray Datatype

```
gsizes[0] = m;  /* no. of rows in global array */
gsizes[1] = n;  /* no. of columns in global array*/

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */

periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD,2,psizes,periods,0,&comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```

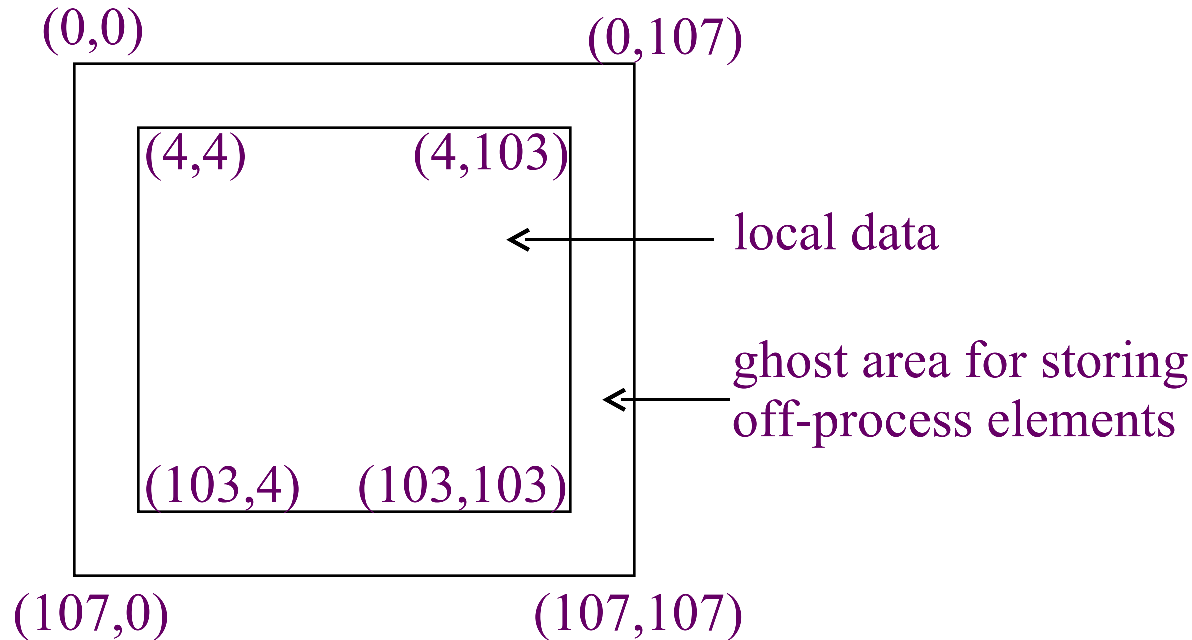
Subarray Datatype contd.

```
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                      MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
              MPI_INFO_NULL);
local_array_size = lsizes[0] * lsizes[1];
MPI_File_write_all(fh, local_array, local_array_size,
               MPI_FLOAT, &status);
```

Local Array with Ghost Area in Memory



- Use a subarray datatype to describe the noncontiguous layout in memory
- Pass this datatype as argument to `MPI_File_write_all`

Local Array with Ghost Area

```
memsizes[0] = lsizes[0] + 8;
   /* no. of rows in allocated array */
memsizes[1] = lsizes[1] + 8;
   /* no. of columns in allocated array */
start_indices[0] = start_indices[1] = 4;
   /* indices of the first element of the local array
      in the allocated array */

MPI_Type_create_subarray(2, memsizes, lsizes,
         start_indices, MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);

/* create filetype and set file view exactly as in the
   subarray example */

MPI_File_write_all(fh, local_array, 1, memtype, &status);
```

Passing Hints to the Implementation

```
MPI_Info info;

MPI_Info_create(&info);

/* no. of I/O devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);

MPI_Info_free(&info);
```

Examples of Hints (used in ROMIO)

- `striping_unit`
- `striping_factor`
- `cb_buffer_size`
- `cb_nodes`

MPI-2 predefined hints

- `ind_rd_buffer_size`
- `ind_wr_buffer_size`

New Algorithm Parameters

- `start_iodevice`
- `pfs_svr_buf`
- `direct_read`
- `direct_write`

Platform-specific hints

File Interoperability

- Users can optionally create files with a portable binary data representation
- “datarep” parameter to `MPI_File_set_view`
- `native` - default, same as in memory, not portable
- `internal` - impl. defined representation providing an impl. defined level of portability
- `external32` - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across machines and MPI implementations

General Guidelines for Achieving High I/O Performance

- Buy sufficient I/O hardware for the machine
- Use fast file systems, not NFS-mounted home directories
- Do not perform I/O from one process only
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call

Achieving High I/O Performance with MPI

- Any application as a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- I/O access patterns in MPI-IO can be classified into four *levels*: level 0 – level 3
- We demonstrate how the user’s choice of *level* affects performance

Example: Distributed Array Access

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

Access Pattern in the file

P0 | P1 | P2 | P3 | P0 | P1 | P2

P4 | P5 | P6 | P7 | P4 | P5 | P6

P8 | P9 | P10 | P11 | P8 | P9 | P10

P12 | P13 | P14 | P15 | P12 | P13 | P14

Level-0 Access (C)

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-1 Access (C)

- Similar to level 0, but each process uses collective I/O functions

```
MPI_File_open(MPI_COMM_WORLD, file, …, &fh);
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, …);
   MPI_File_read_all(fh, &(A[i][0]), …);
}
MPI_File_close(&fh);
```

Level-2 Access (C)

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

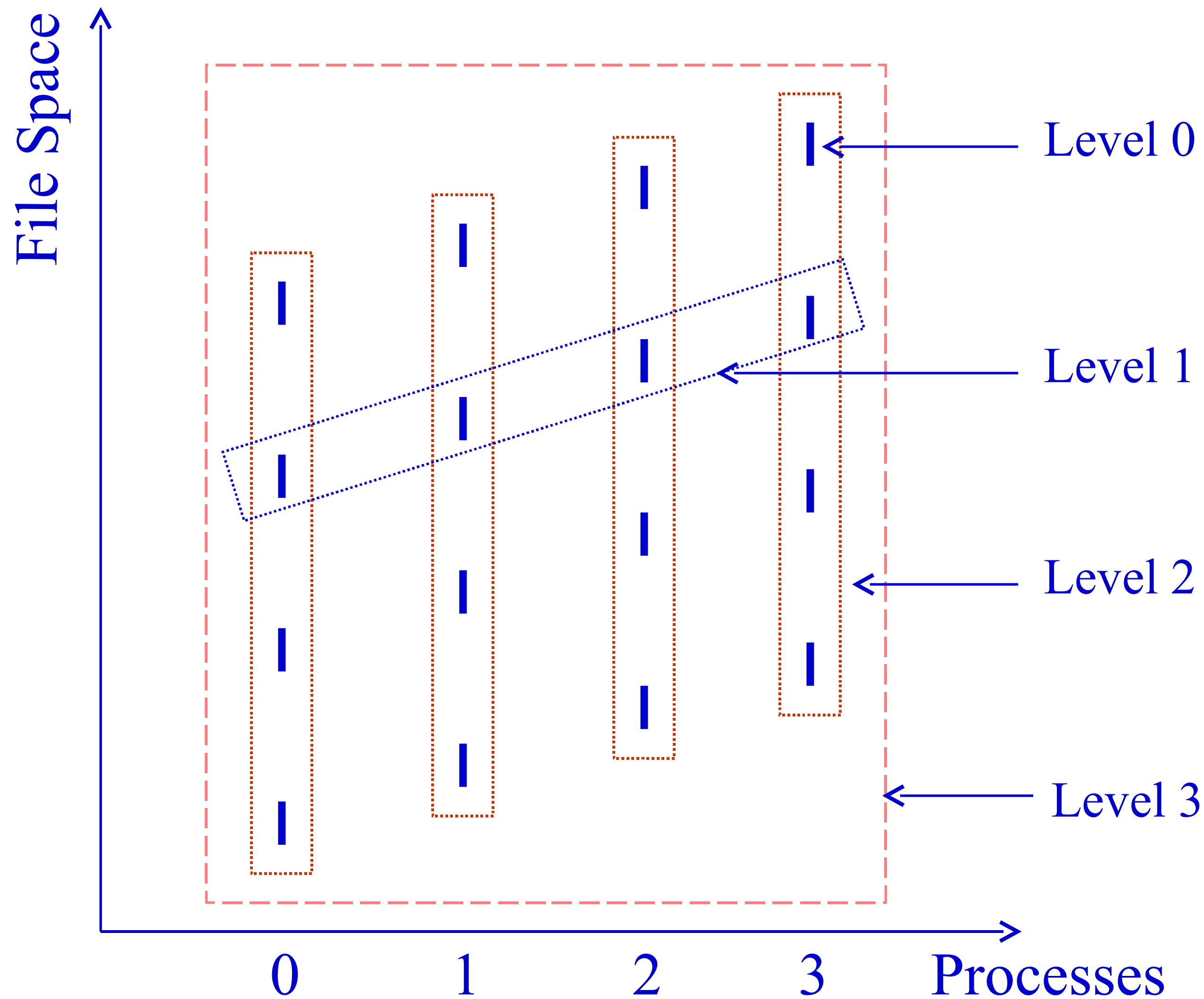
```
MPI_Type_create_subarray(…, &subarray, …);
MPI_Type_commit(&subarray);
MPI_File_open(…, file, …, &fh);
MPI_File_set_view(fh, …, subarray, …);
MPI_File_read(fh, A, …);
MPI_File_close(&fh);
```

Level-3 Access (C)

- Similar to level 2, except that each process uses collective I/O functions

```
MPI_Type_create_subarray(…, &subarray, …);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file, …, &fh);
MPI_File_set_view(fh, …, subarray, …);
MPI_File_read_all(fh, A, …);
MPI_File_close(&fh);
```

The Four Levels of Access



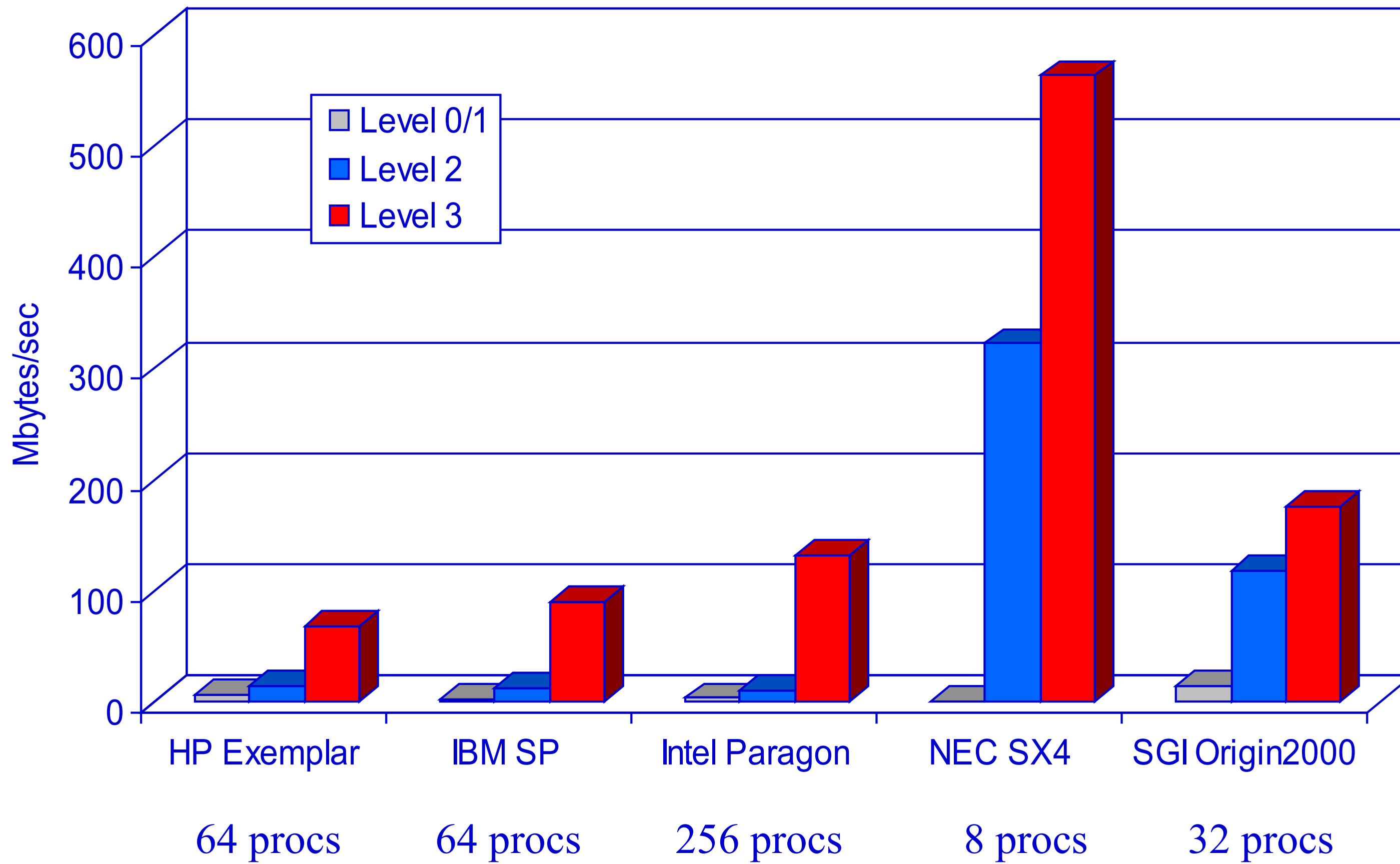
Optimizations

- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really needed
  - Collective I/O: Merge requests of different processes into larger requests
  - Improved prefetching and caching

Performance Results

- Distributed array access
- Unstructured code from Sandia
- On five different parallel machines:
  - HP Exemplar
  - IBM SP
  - Intel Paragon
  - NEC SX-4
  - SGI Origin2000

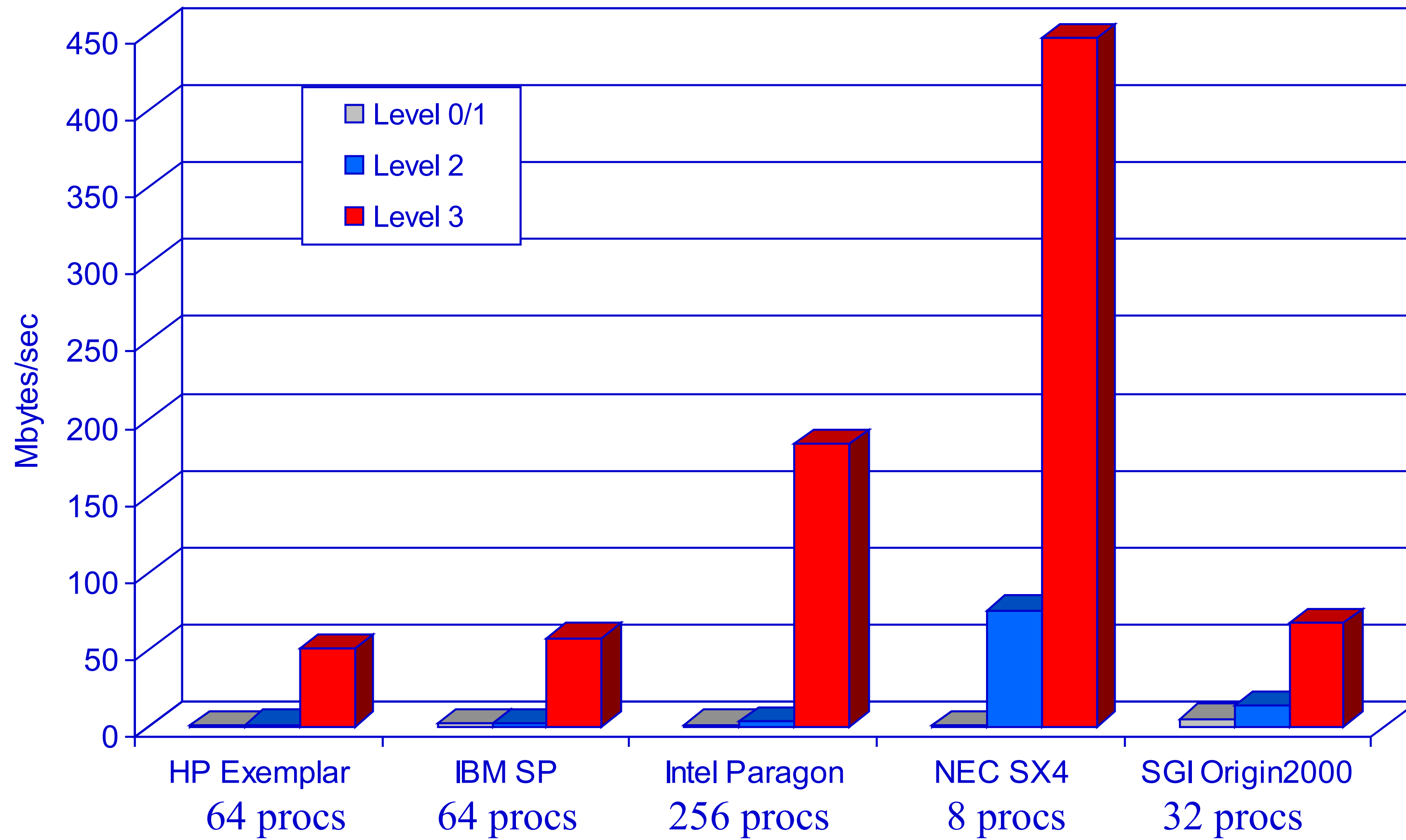
Distributed Array Access: Read Bandwidth



64 procs 64 procs 256 procs 8 procs 32 procs

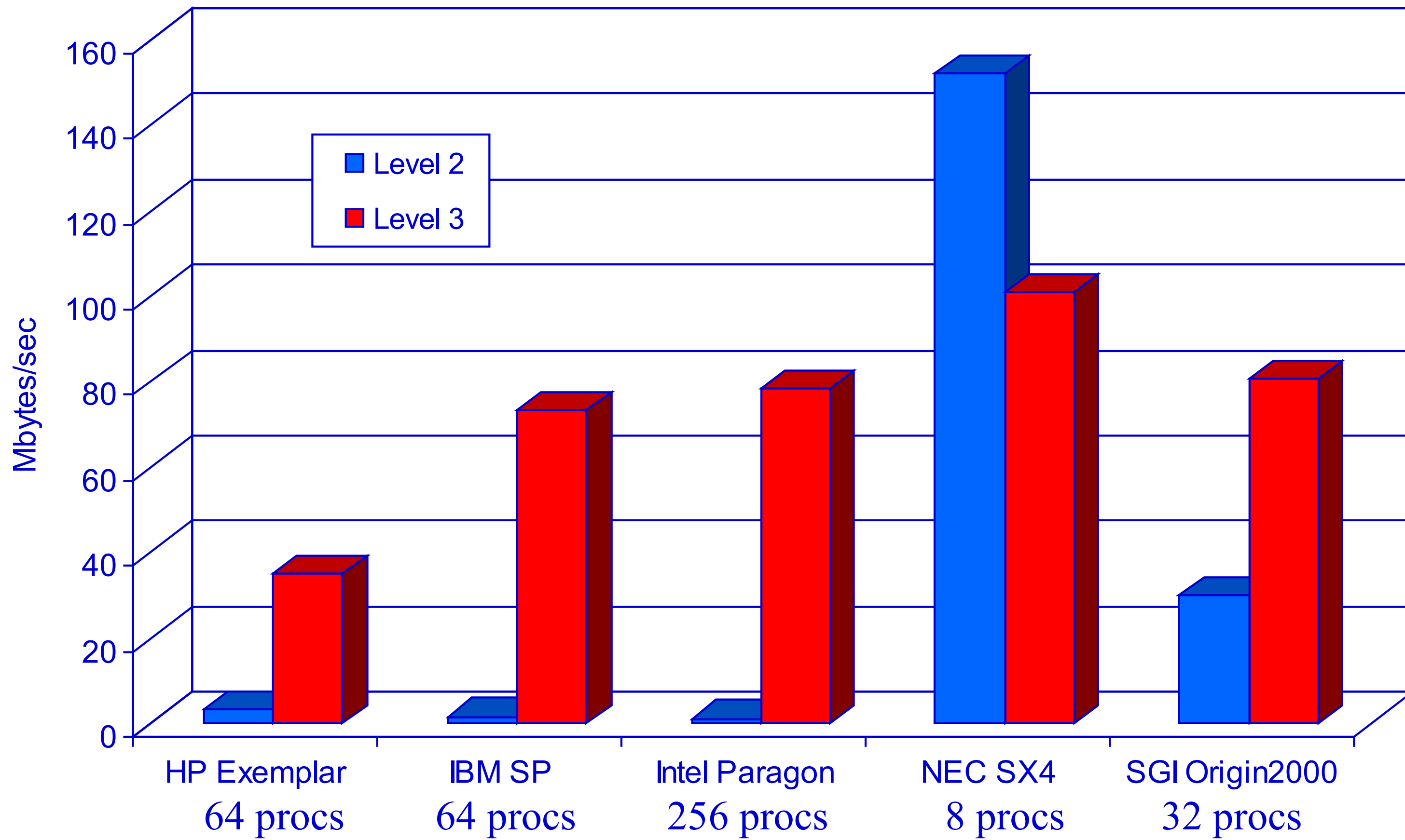
Array size: 512 x 512 x 512

Distributed Array Access: Write Bandwidth

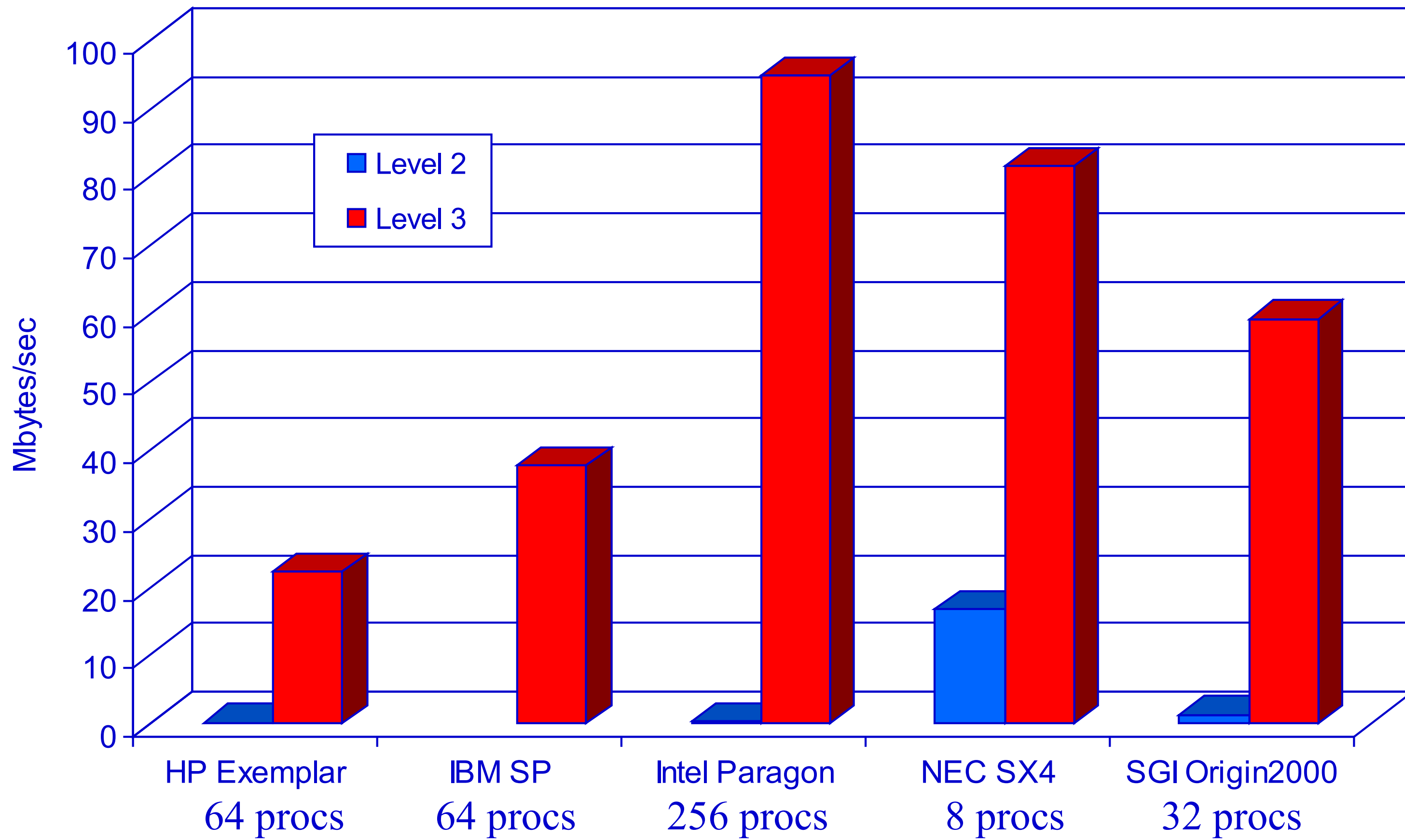


Array size: 512 x 512 x 512

Unstructured Code: Read Bandwidth



Unstructured Code: Write Bandwidth

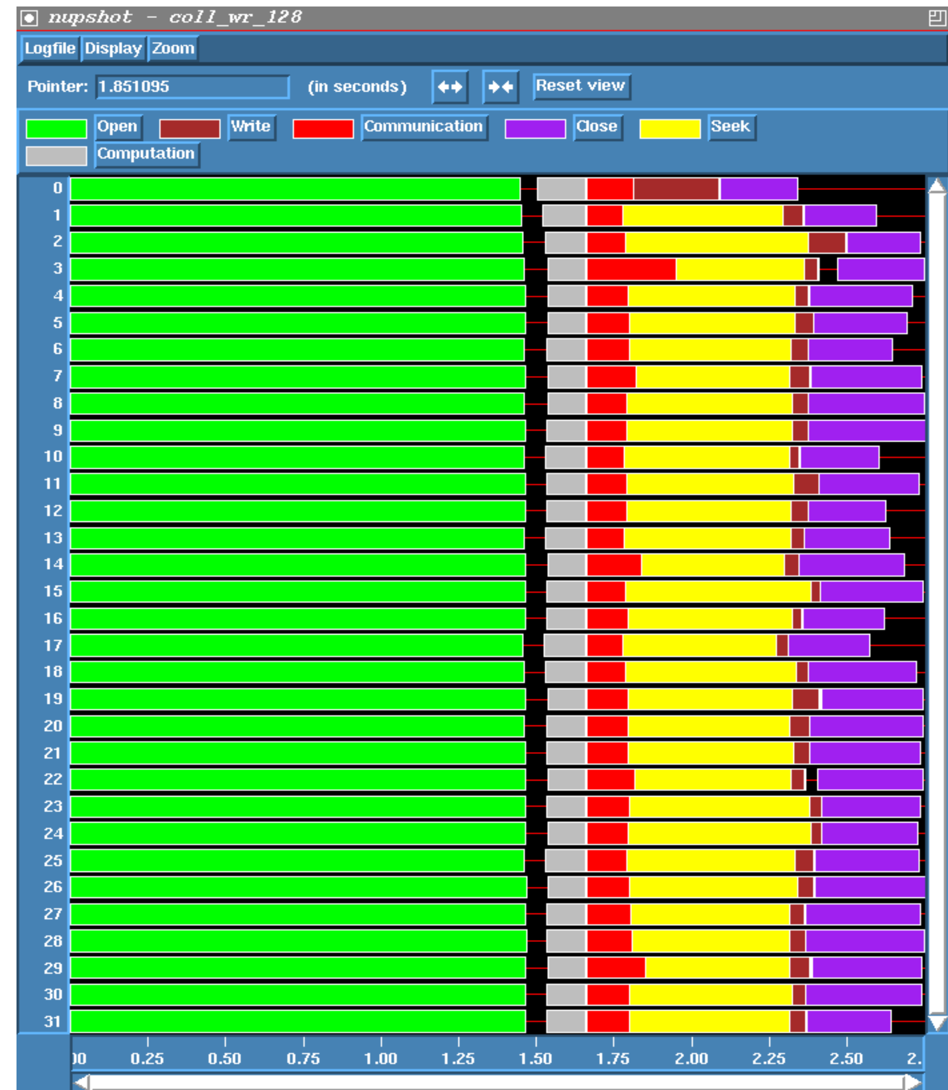


Independent Writes

- On Paragon
- Lots of seeks and small writes
- Time shown = 130 seconds

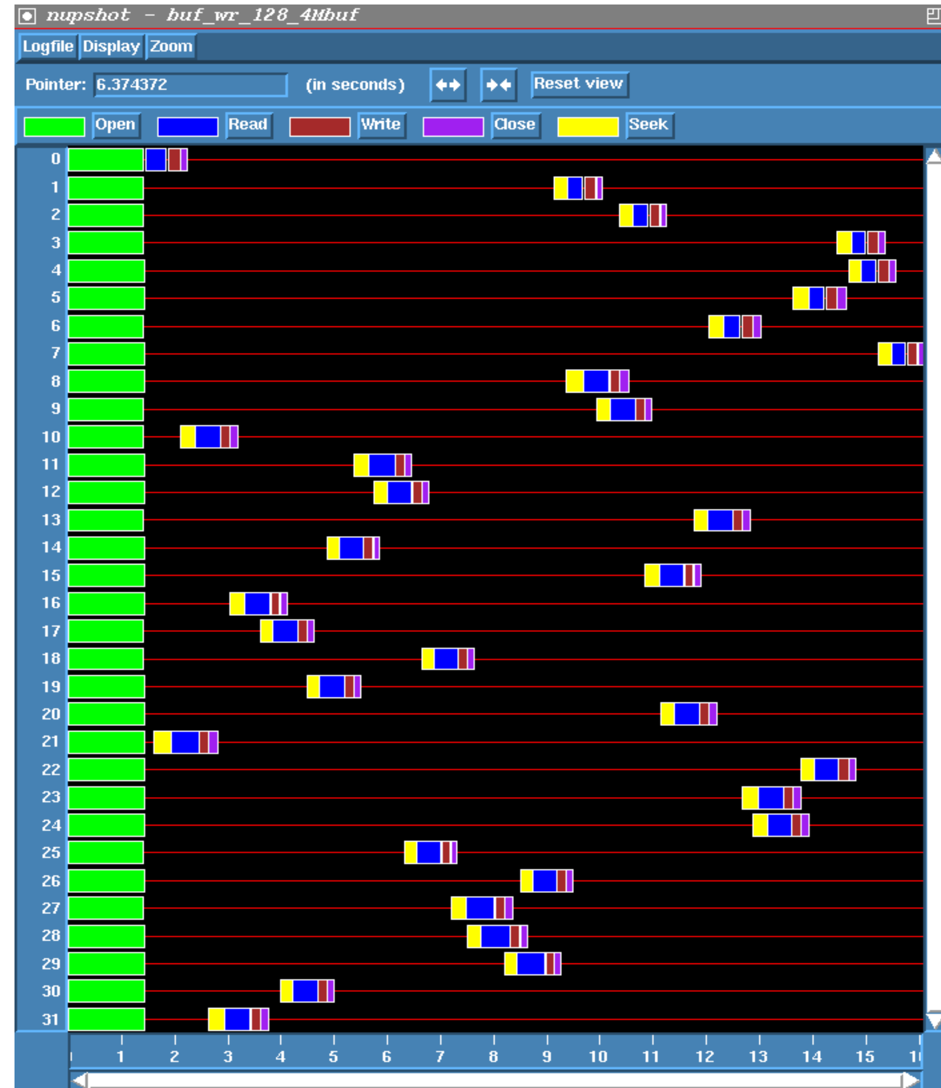
Collective Write

- On Paragon
- Computation and communication precede seek and write
- Time shown = 2.75 seconds



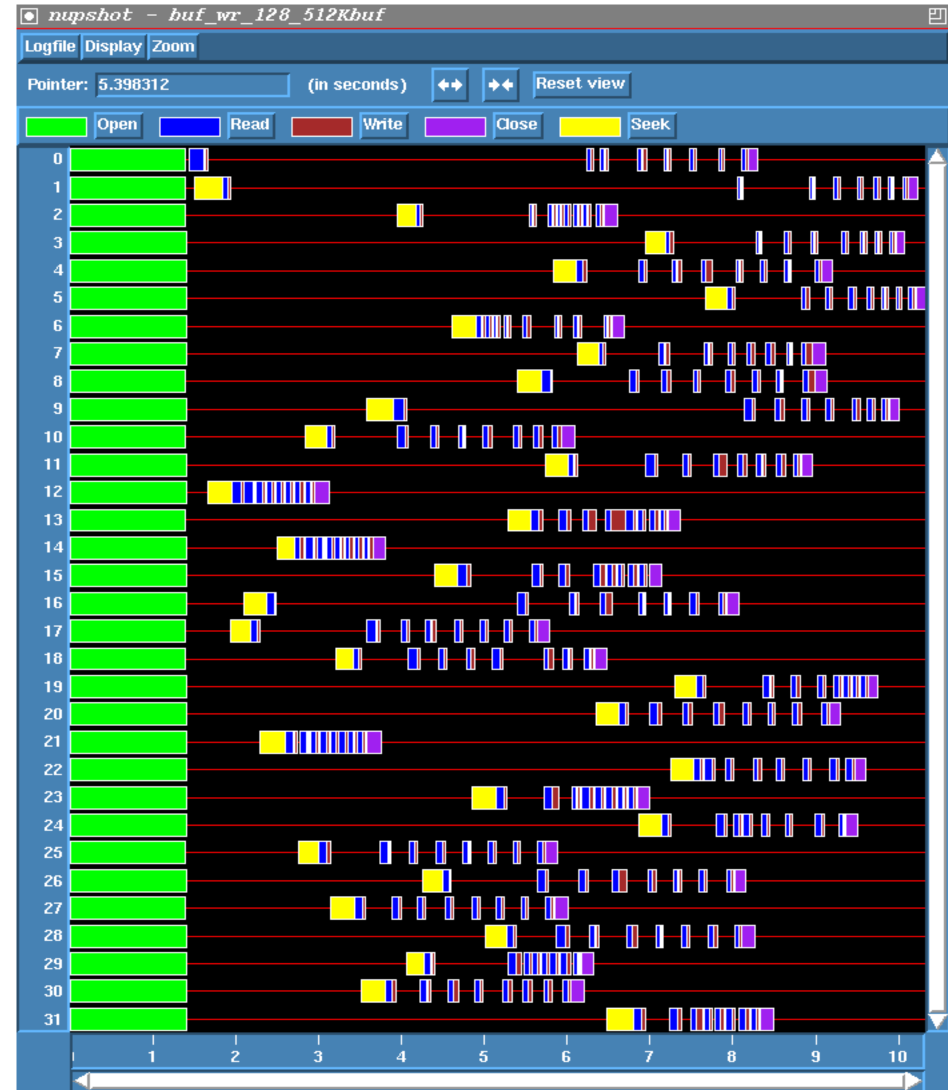
Independent Writes with Data Sieving

- On Paragon
- Access data in large “blocks” and extract needed data
- Requires lock, read, modify, write, unlock for writes
- 4 MB blocks
- Time = 16 sec.



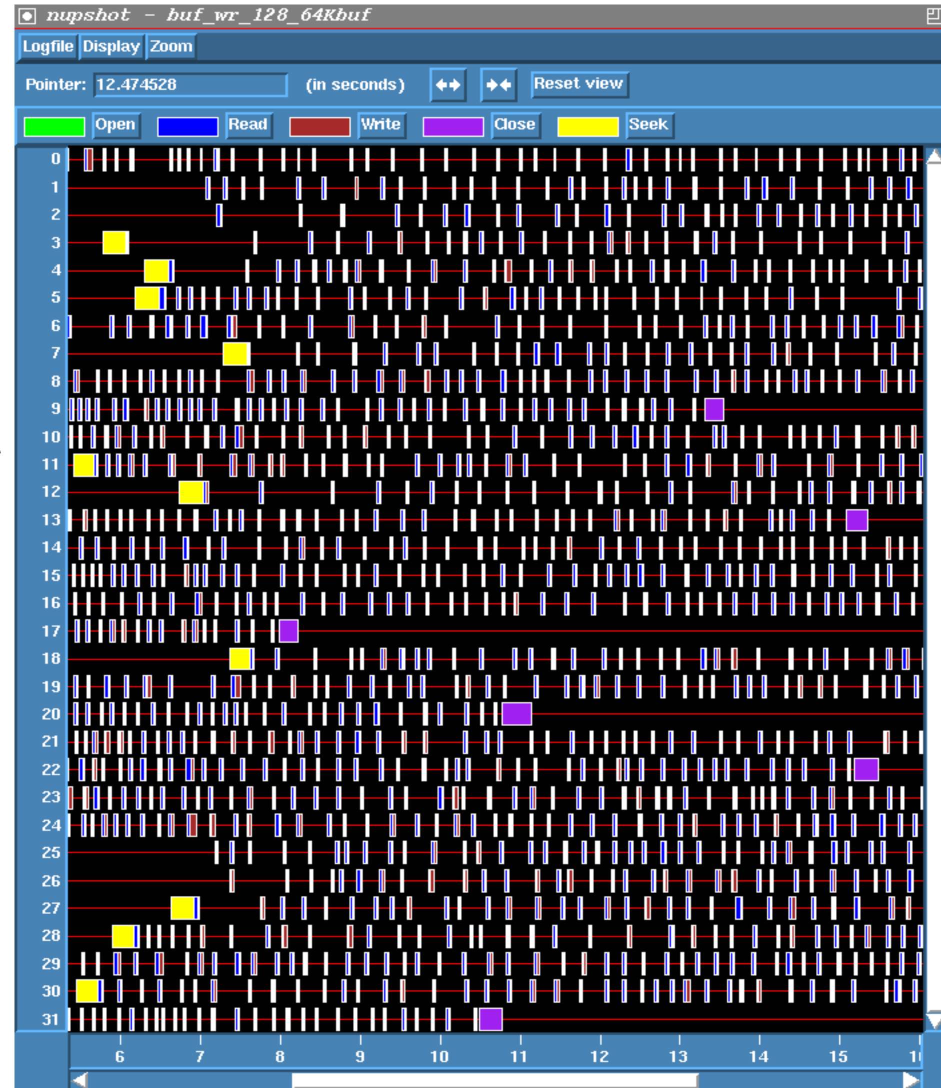
Changing the Block Size

- Smaller blocks mean less contention, therefore more parallelism
- 512 KB blocks
- Time = 10.2 seconds



Data Sieving with Small Blocks

- If the block size is too small, however, the increased parallelism doesn't make up for the many small writes
- 64 KB blocks
- Time = 21.5 seconds



Common Errors in Using MPI I/O

- Not defining file offsets as `MPI_Offset` in C and `integer (kind=MPI_OFFSET_KIND)` in Fortran (or perhaps `integer*8` in Fortran 77)
- In Fortran, passing the offset or displacement directly as a constant (e.g., 0) in the absence of function prototypes (F90 mpi module)
- Using darray datatype for a block distribution other than the one defined in darray (e.g., floor division)
- filetype defined using offsets that are not monotonically nondecreasing, e.g., 0, 3, 8, 4, 6. (happens in irregular applications)

Summary

- MPI I/O has many features that can help users achieve high performance
- The most important of these features are the ability to specify noncontiguous accesses, the collective I/O functions, and the ability to pass hints to the implementation
- Users must use the above features!
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

MPI and Threads

- MPI describes parallelism between *processes*
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and pthreads are common
  - OpenMP provides convenient features for loop-level parallelism

Threads and MPI in MPI-2

- MPI-2 specifies four levels of thread safety
  - MPI_THREAD_SINGLE : only one thread
  - MPI_THREAD_FUNNELED : only one thread that makes MPI calls
  - MPI_THREAD_SERIALIZED : only one thread at a time makes MPI calls
  - MPI_THREAD_MULTIPLE : any thread can make MPI calls at any time
- MPI_Init_thread(..., required, &provided) can be used instead of MPI_Init

Where is MPI in 2008?

- MPI is synonymous with message passing
- Provides substrate for other programming models
- Provides portability for libraries (ScaLAPack, PETSc, many others)
- Many implementations at all levels
  - From the IBM BlueGene to machines in homes
  - All vendors
  - Freely available (MPICH, OpenMPI, others)

What is Next for MPI?

- No radical changes or additions on the immediate horizon
- MPI Forum – standards body for MPI (mpi-forum.org)
  - MPI-2.1 released a few months ago
    - Fixed errors and ambiguities in the standard
  - MPI-2.2 is being worked on
    - More fixes, minor additions to standard
  - MPI 3.0 also being worked on
    - Major changes to standard, possibly changes to existing API to address scalability
- MPI will be around for a long time
  - Well established application base
  - Standard allows for high-performance, scalable implementations

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

MPICH Facts

- MPICH2
  - High-performance
  - Open-source
  - Widely portable
- 1.6 million lines of code in 5,000 files
- Been alive for 15 years w/ releases about every 6 months
  - Three source code management systems
- MPICH-based implementations
  - IBM for BG/L and BG/P
  - Cray for XT3/4
  - Intel
  - Microsoft
  - SiCortex
  - Myricom
  - Ohio State

Some Research Areas

- MPI-2 RMA interface
  - Can we get high performance?
- Fault Tolerance and MPI
  - Are intercommunicators enough?
- MPI on 100s of thousands of processors
  - Improve scalability of internal data structures
    - Performance trade-offs
  - Hybrid programming model: threads + MPI

Getting MPICH for your cluster

- http://www.mcs.anl.gov/research/projects/mpich2
  - Or just google MPICH2

Top MPI Errors: I

- Fortran: missing ierr argument
- Fortran: missing MPI_STATUS_SIZE on status
- MPI_Bcast not called collectively (e.g., sender bcasts, receivers use MPI_Recv)
- Failure to wait on MPI_Request
- Reusing buffers on nonblocking operations
- Using the mpirun from the wrong MPI implementation
- Assuming that command-line args are available on all proceses

Top MPI Errors: II

- Using a single process for all file I/O
- Using MPI_Pack/Unpack instead of Datatypes
- Unsafe use of blocking sends/receives
- Using MPI_COMM_WORLD instead of comm in libraries
- Not understanding implementation performance settings
- Failing to install and use the MPI implementation according to its documentation

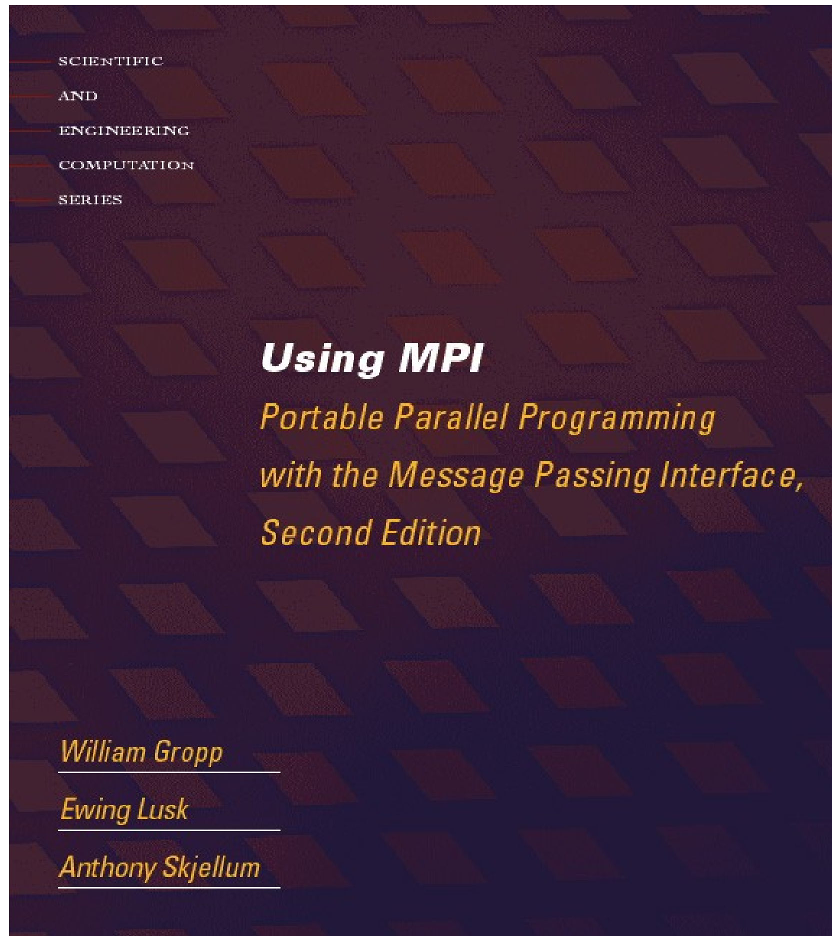
Conclusions

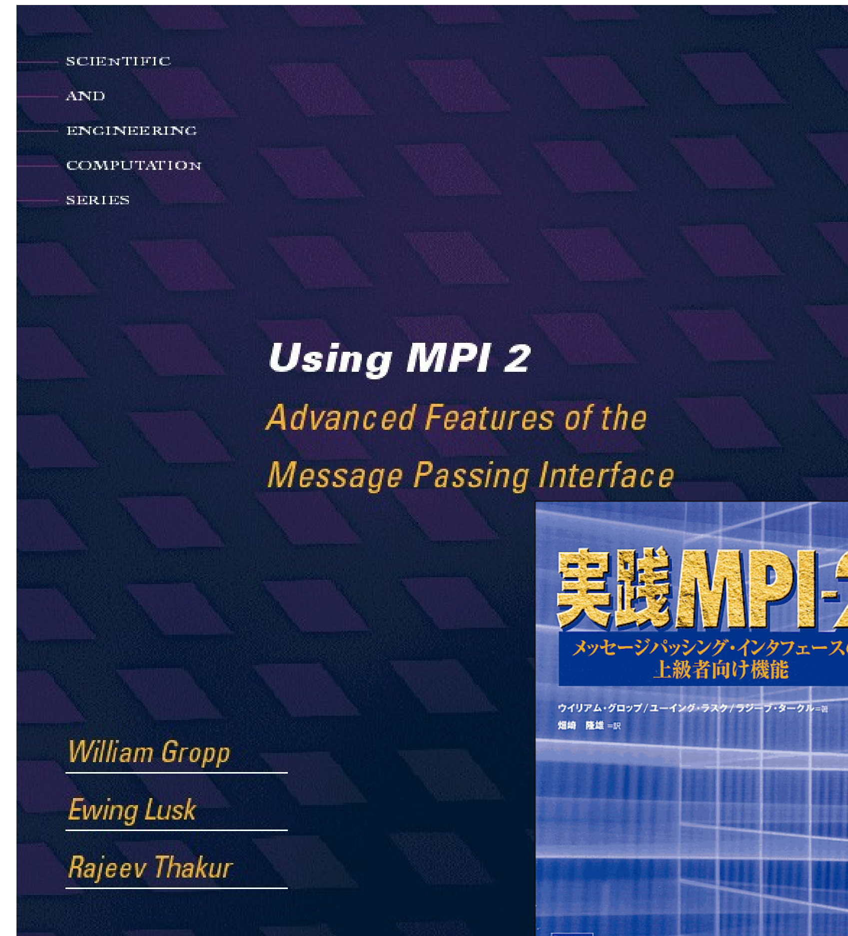
- MPI provides a well-developed, efficient and portable model for programming parallel computers
- Just as important, MPI provides features that enable and encourage the construction of software *components*.
- Parallel applications can often be quickly built using these components
- Even when no available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

Conclusions contd.

- MPI is a proven, effective, portable parallel programming model
  - 1.4PF application on the IBM RoadRunner with 122,400 cores
- MPI has succeeded because
  - features are orthogonal (complexity is the product of the number of *features*, not routines)
  - programmer can control memory motion (critical in high-performance computing)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design

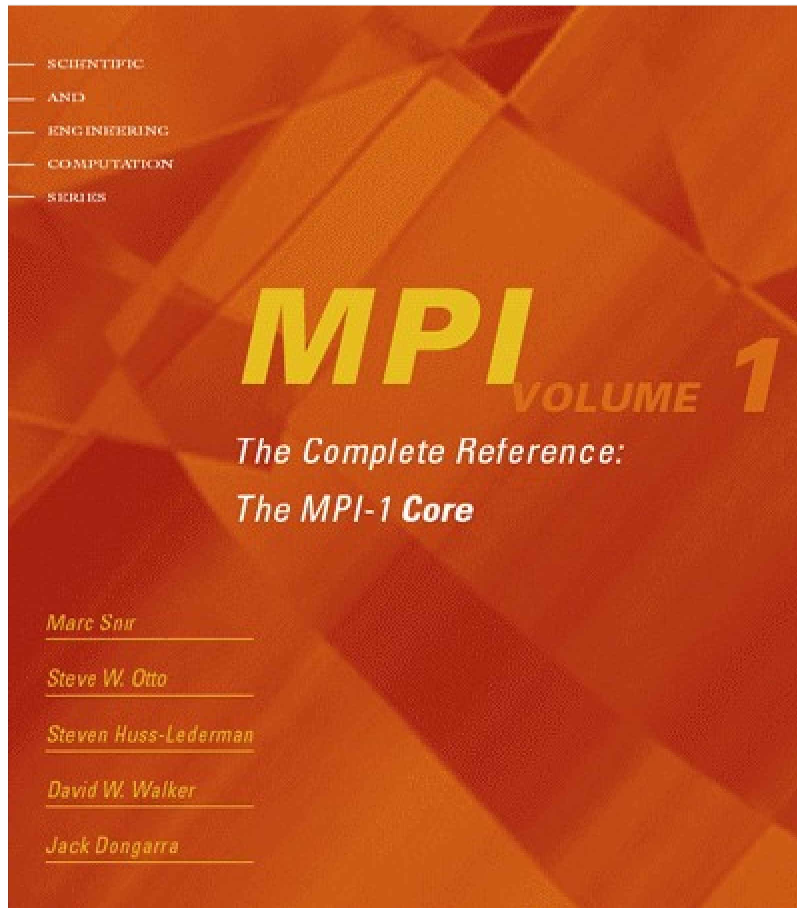
Tutorial Material on MPI, MPI-2

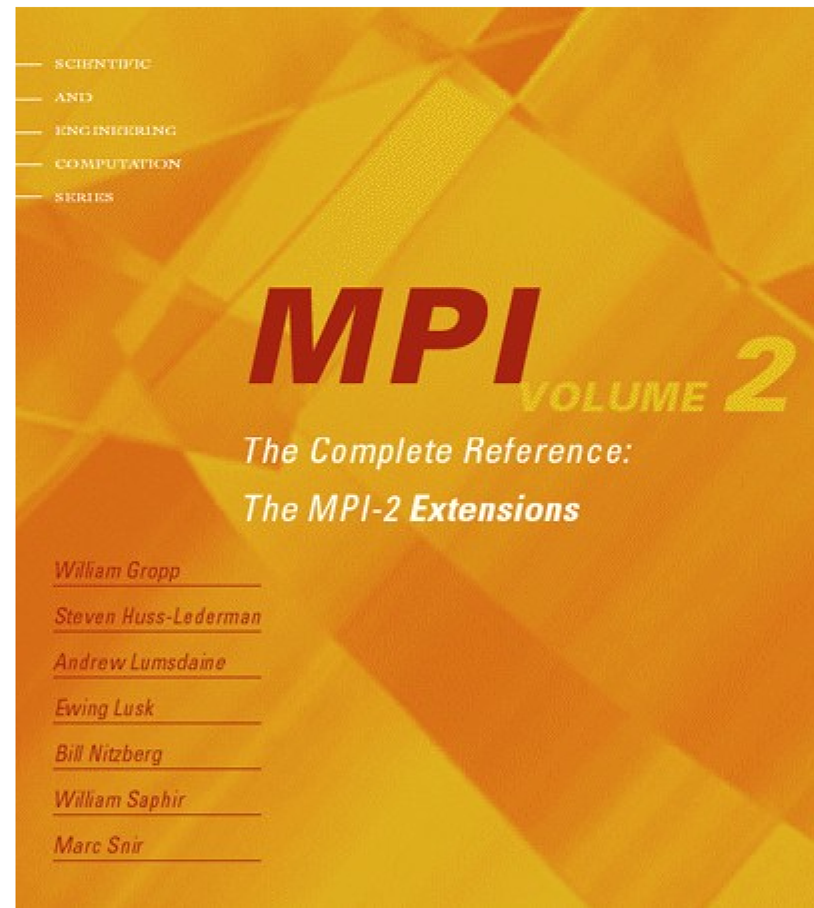




http://www.mcs.anl.gov/mpi/{usingmpi,usingmpi2}

The MPI Standard (1 & 2)





The End