Introduction to the Message Passing Interface

William Gropp
Mathematics and Computer Science
Argonne National Laboratory

Thanks

- Rusty Lusk and Rajeev Thakur for some of this material

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 Gordan Bell Prize finalists)
- 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

Example: PDE Solution

- Problem: Solve an elliptic PDE on non-rectangular domain using unstructured grids
- Approach: Preconditioned Krylov Method
- Best preconditioners are
  - Hierarchical (Multigrid, Overlapping Schwarz domain decomposition)
  - Incomplete Factorization
- Krylov method also requires
  - Sparse matrix-vector product
  - Dot products

Consequences

- Hierarchical methods
  - Apply same methods on subdomains
    - Memory locality forces subdomains onto subsets of processors
- Incomplete Factorization
  - Data dependencies in (sparse) triangular solves can serialize execution
- Sparse matrix-vector products
  - Low data reuse requires careful use of memory
- Dot products
  - In hierarchical methods, must be applied to subsets of processors

Programming Issues

- Modular programming in parallel systems introduces new issues
  - Whole-program methods inapplicable
    - simple data distributions
  - Common collective operations (e.g., barrier, reduction)
    - must be restricted to subsets
- Numerical Libraries require a parallel environment in which to operate

Large-Scale Parallel Implicit Pseudotransient Newton-Krylov-Schwarz Solution of a 3D Unstructured Steady Incompressible Euler Problem

Procs	**Its**	**Exec Time**	**Speed Up**	**Algor. Effic.**	**Communications** Inner Prod.	Halo Exchange	Sustained Mflops/s per proc	**Sustained total Gflop/s**
128	164	6,048s	1.00	—	6%	3%	66.7	8.5
256	166	3,242s	1.87	99%	8%	3%	64.8	16.6
512	171	1,811s	3.34	96%	12%	4%	62.6	32.1

- FUN3D-PETSc ONERA M6 Wing Test Case, 2nd-order Roe Scheme, 1-layer Halo; Tetrahedral grid of 2,761,744 vertices (11,047,096 unknowns) on Cray T3E-900 (450 MHz) at NERSC
- 83% overall efficiency on 512 nodes, relative to 128 nodes (smallest on which problem fits)
- 32.1 Gflop/s on 512 nodes for a **sparse unstructured** computation—12% of **dense** LINPACK peak (79744 ×79744 matrix)
- Principle nonscaling features are inner products and norms—due to a load imbalance
- Principle “slow” routines are orthogonalization in Krylov method and Jacobian preconditioner formation

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

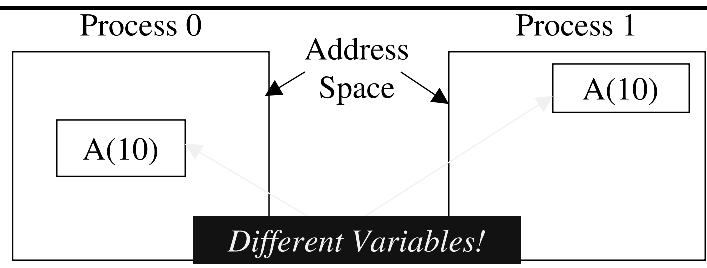
- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program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

More on "Single Name Space"



- Integer A(10)
  …
  print *, A

- Integer A(10)
  do i=1,10
    A(i) = i
  enddo
  ...

More on 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?

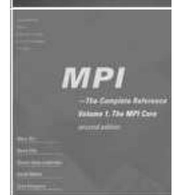
- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware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

Where Did MPI Come From?

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

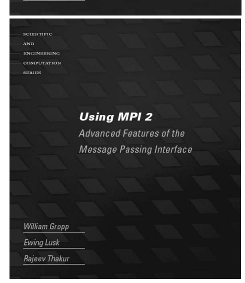
Books on MPI

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
- *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.



More Books on MPI

- *Using MPI, 2nd edition*, by Gropp, Lusk, and Skjellum, MIT Press, 1999
- *Using MPI-2*, by Gropp, Lusk, and Thakur, MIT Press, 1999



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

Simple Program

- Fortran Program:

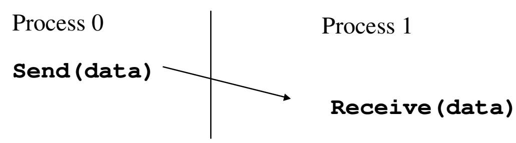
```
program main
include 'mpif.h'
integer ierr, rank, size, i
real A(10)
call MPI_Init( ierr )
call MPI_Comm_size( MPI_COMM_WORLD, size, ierr )
call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
do i=1,10
    A(i) = i * rank
enddo
print *, 'My rank ', rank, ' of ', size
print *, 'Here are my values for A:'
print *, A
call MPI_Finalize( ierr )
end
```

Notes on Simple Program

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpif.h and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `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 undefined (may be interleaved by character, line, or blocks of characters)

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 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 **`des`** , which is the rank of the target process in the communicator specified by **`com`** .
- 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.

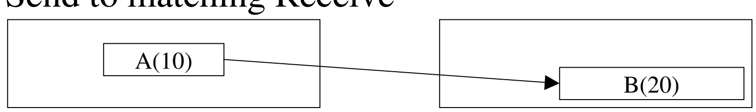
MPI Basic (Blocking) Receive

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 **`coun`** occurrences of **`datatyp`** is OK, but receiving more is an error.

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

Retrieving Further Information

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

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

- In Fortran:

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

Tags and Contexts

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

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.

Where Should MPI be Used?

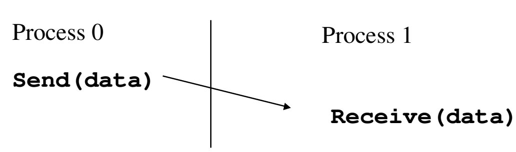
- Massively parallel systems
  - MPI's programming model designed to be *scalable* to 1000's of tasks
  - Special *collective operations* provide access to system-specific scalable communication
- High-performance computing
  - Programmer has extensive control of data motion
  - Designed to encourage and support the development of modular, parallel software

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.

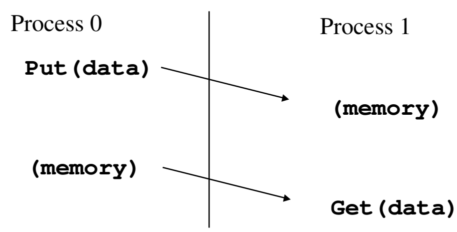
Cooperative Operations for Communication

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



One-Sided Operations for Communication

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



MPI Sources

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - *MPI: The Complete Reference 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.
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Notes on C and Fortran

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

What MPI Functions are in Use?

- For single application:
  - MPI_Irecv
  - MPI_Isend
  - MPI_Wait
  - MPI_Finalize
  - MPI_Init
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Allreduce

Libraries use More MPI Routines

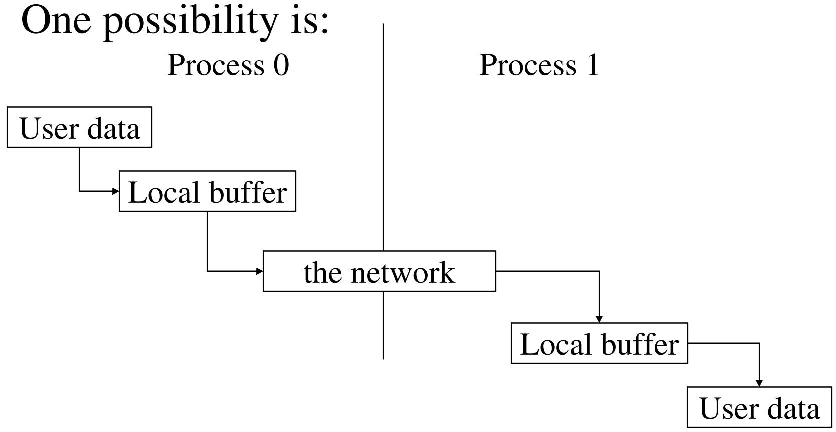
- MPI_Abort
- MPI_Allgather
- MPI_Allgatherv
- MPI_Allreduce
- MPI_Attr_delete
- MPI_Attr_get
- MPI_Attr_put
- MPI_Barrier
- MPI_Bcas
- MPI_Cancel
- MPI_Comm_compare
- MPI_Comm_create
- MPI_Comm_dup
- MPI_Comm_free
- MPI_Comm_group
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Errhandler_create
- MPI_Errhandler_set
- MPI_Gather
- MPI_Gatherv
- MPI_Get_count
- MPI_Group_free
- MPI_Group_incl
- MPI_Init
- MPI_Initialized
- MPI_Irecv
- MPI_Isend
- MPI_Keyval_create
- MPI_Keyval_free
- MPI_Recv
- MPI_Recv_init
- MPI_Reduce
- MPI_Request_free
- MPI_Rsend_init
- MPI_Scan
- MPI_Scatterv
- MPI_Send
- MPI_Send_init
- MPI_Ssend_init
- MPI_Start
- MPI_Startall
- MPI_Type_size
- MPI_Wait
- MPI_Waitall
- MPI_Waitany

PETSc (PDE solvers)

Break

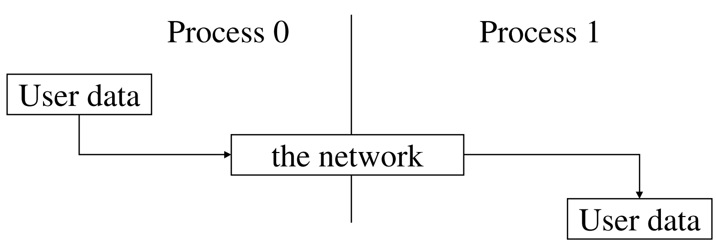
Buffers

- 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_Rec` does not complete until the buffer is full (available for use).
  - `MPI_Sen` 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

- 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

- 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.

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

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.

```
integer bufsize, buf(10000)

call MPI_Buffer_attach( buf, bufsize, ierr )
...
call MPI_Bsend( ... same as MPI_Send ... )
...
call MPI_Buffer_detach( buf, bufsize, ierr )
```

- **`MPI_Buffer_detac`** 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

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

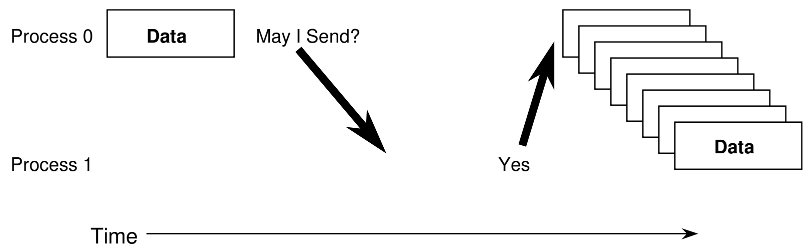
MPI_Sendrecv

- 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)`**

What is message passing?

- Data transfer plus synchronization



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

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)

Message protocols

- Message consists of "envelope" and data
  - Envelope contains tag, communicator, length, source information, plus impl. private data
- Short
  - Message data (message for short) sent with envelope
- Eager
  - Message sent assuming destination can store
- Rendezvou
  - Message not sent until destination oks

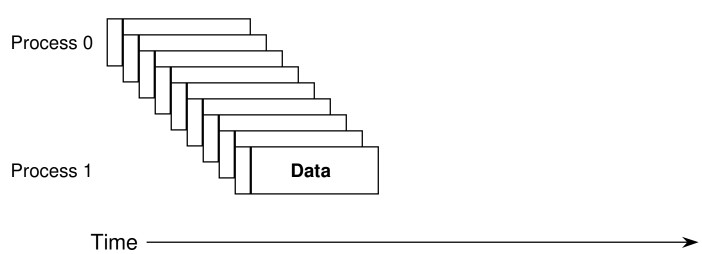
MPI on Distributed Shared Memory Systems

- Message passing is a good way to use distributed shared memory (DSM) machines because it provides a way to express memory locality.
- Put
  - Sender puts to destination memory (user or MPI buffer). Like Eager.
- Get
  - Receiver gets data from sender or MPI buffer. Like Rendezvous.
- Short, long, rendezvous versions of these

Message Protocol Details

- Eager not Rsend, rendezvous not Ssend resp., but related
- User versus system buffer space
- Packetization
- Collective operations
- Datatypes, particularly non-contiguous
  - Handling of important special cases
    - Constant stride
    - Contiguous structures

Eager Protocol



- Data delivered to process 1
  - No matching receive may exist; process 1 must then buffer and copy.

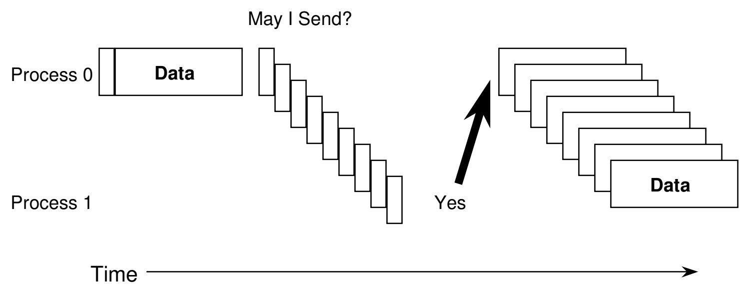
Eager Features

- Reduces synchronization delays
- Simplifies programming (just MPI_Send)
- Requires significant buffering
- May require active involvement of CPU to drain network at receiver's end
- May introduce additional copy (buffer to final destination)

How Scaleable is Eager Delivery?

- Buffering must be reserved for arbitrary senders
- User-model mismatch (often expect buffering allocated entirely to "used" connections).
- Common approach in implementations is to provide same buffering for all members of MPI_COMM_WORLD; this is optimizing for non-scaleable computations
- Scalable implementations that exploit message patterns are possible (but not widely implemented)

Rendezvous Protocol



- Envelope delivered first
- Data delivered when user-buffer available
  - Only buffering of envelopes required

Rendezvous Features

- Robust and safe
  - (except for limit on the number of envelopes…)
- May remove copy (user to user direct)
- More complex programming (waits/tests)
- May introduce synchronization delays (waiting for receiver to ok send)

Short Protocol

- Data is part of the envelope
- Otherwise like eager protocol
- May be performance optimization in interconnection system for short messages, particularly for networks that send fixed-length packets (or cache lines)

User and System Buffering

- Where is data stored (or staged) while being sent?
  - User's memory
    - Allocated on the fly
    - Preallocated
  - System memory
    - May be limited
    - Special memory may be faster

Implementing MPI_Isend

- Simplest implementation is to always use rendezvous protocol:
  - MPI_Isend delivers a request-to-send control message to receiver
  - Receiving process responds with an ok-to-send
    - May or may not have matching MPI receive; only needs buffer space to store incoming message
  - Sending process transfers data
- Wait for MPI_Isend request
  - wait for ok-to-send message from receiver
  - wait for data transfer to be complete on sending side

Alternatives for MPI_Isend

- Use a short protocol for small messages
  - No need to exchange control messages
  - Need guaranteed (but small) buffer space on destination for short message envelope
  - Wait becomes a no-op
- Use eager protocol for modest sized messages
  - Still need guaranteed buffer space for both message envelope and eager data on destination
  - Avoids exchange of control messages

Implementing MPI_Send

- Can't use eager always because this could overwhelm the receiving process

```
if (rank != 0) MPI_Send( 100 MB of data )
else receive 100 MB from each process
```

- Would like to exploit the blocking nature (can wait for receive)
- Would like to be fast
- Select protocol based on message size (and perhaps available buffer space at destination)
  - Short and/or eager for small messages
  - Rendezvous for longer messages

Implementing MPI_Rsend

- Just use MPI_Send; no advantage for users
- Use eager always (or short if small)
  - even for long messages

Why Bsend?

- Buffer space not infinite
- Careful management of buffering required for *correctness* and *performance*
- MPI provides user control over buffering modes
- Allows implementations freedom to pick specific protocols
  - Users give up some control for more convenient general-purpose send (MPI_Send)
- Implementations *could* make Bsend fast, at least when buffering is available at the destination

Lunch

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. See attribute **`MPI_WTIME_IS_GLOBAL`**.

Timing MPI Programs

- 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 — 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

```
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;
    }
```

- Use MPI_Wtick to discover clock resolution
- Use getrusage to get other effects (e.g., context switches, paging)

Sample Timing Harness

- Average times, make several trials

```
tfinal=1000.0
do k=1, nloop
    t1 = MPI_Wtime()
     do i=1,maxloop
          <operation to be timed>
    enddo
    time = (MPI_Wtime() - t1)/maxloop
    if (time .lt. tfinal) tfinal = time
enddo
```

- Use MPI_Wtick to discover clock resolution
- Use getrusage to get other effects (e.g., context switches, paging)

Pitfalls in timing (C)

- Time too short:

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

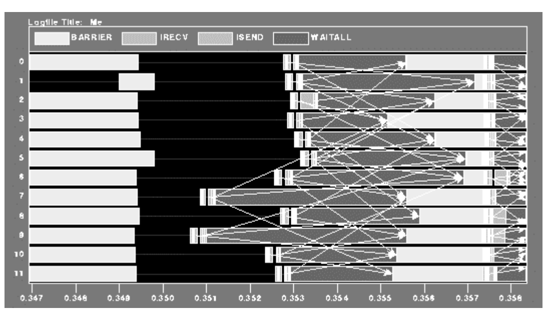
- Underestimates by MPI_Wtick, over by cost of calling MPI_Wtime
- "Correcting" MPI_Wtime by subtracting average of MPI_Wtime calls overestimates MPI_Wtime
- Code not paged in (always run at least twice)
- Minimums not what users see
- Tests with 2 processors may not be representative
  - T3D has processors in pairs, pingpong give 130 MB/sec for 2 but 75 MB/sec for 4 (for MPI_Ssend)

Pitfalls in timing (Fortran)

- Time too short:
```
t = MPI_Wtime()
call MPI_Send(…)
time = MPI_Wtime() - t
```
- Underestimates by MPI_Wtick, over by cost of calling MPI_Wtime
- "Correcting" MPI_Wtime by subtracting average of MPI_Wtime calls overestimates MPI_Wtime
- Code not paged in (always run at least twice)
- Minimums not what users see
- Tests with 2 processors may not be representative
  - T3D has processors in pairs, pingpong give 130 MB/sec for 2 but 75 MB/sec for 4 (for MPI_Ssend)

Example of Paging Problem

- Black area is *identical* setup computation

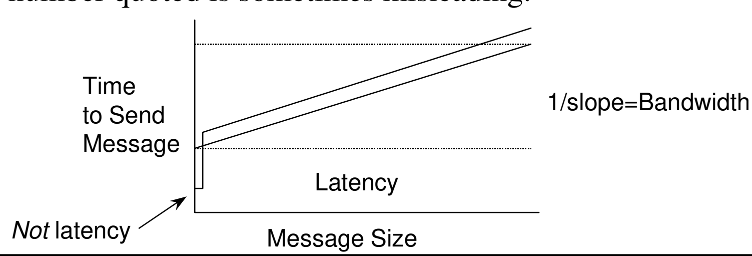


Latency and Bandwidth

- 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.

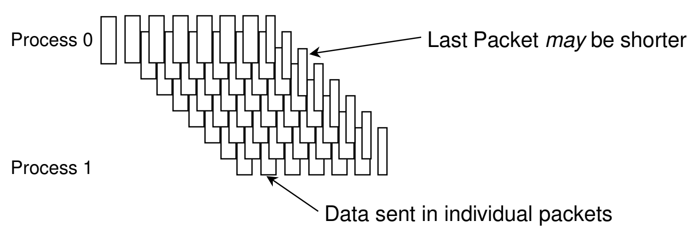


Exercise: Timing MPI Operations

- Estimate the latency and bandwidth for some MPI operation (e.g., Send/Recv, Bcast, Ssend/Irecv-Wait)
  - Make sure all processes are ready before starting the test
  - How repeatable are your measurements?
  - How does the performance compare to the performance of other operations (e.g., memcpy, floating multiply)?

Packetization

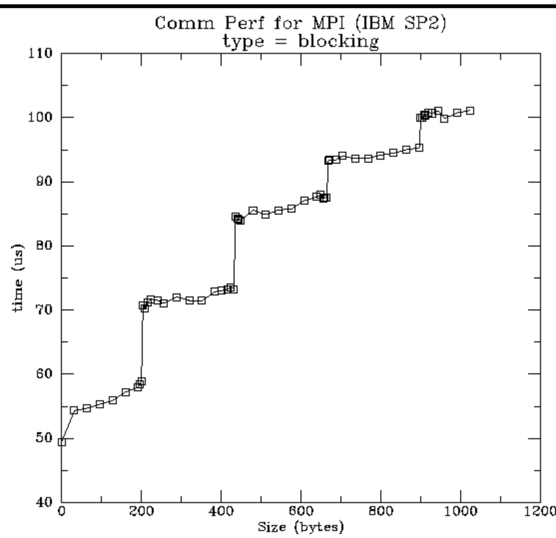
- Some networks send data in discrete chunks called *packets*



Introduces a ceil(n/packet_size) term

Staircase appearance of performance graph

Example of Packetization



Packets contain 232 bytes of data. (first is 200 bytes, so MPI header is probably 32 bytes).

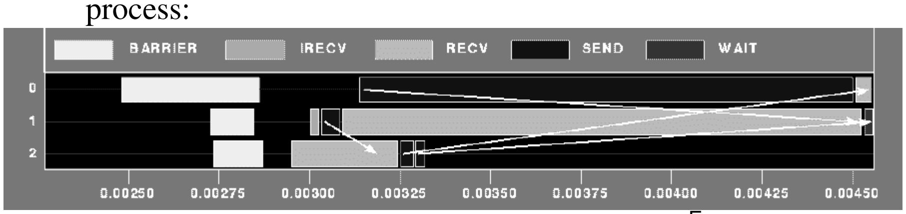
Data from mpptest, available at ftp://ftp.mcs.anl.gov/pub/mpi/misc/perftest.tar.gz

Synchronization Delays

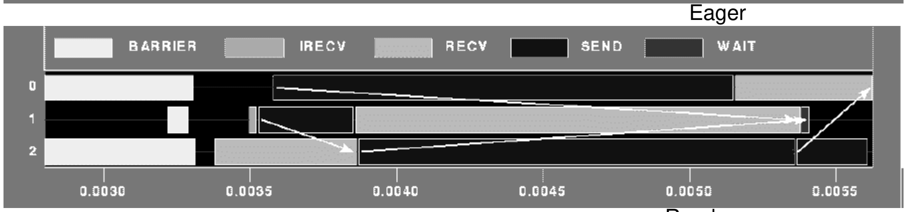
- Message passing is a cooperative method — if the partner doesn't react quickly, a delay results
- There is a performance tradeoff caused by reacting quickly — it requires devoting resources to checking for things to do

Observing Synchronization Delays

- 3 processors sending data, with one sending a short message and another sending a long message to the same process:



Eager



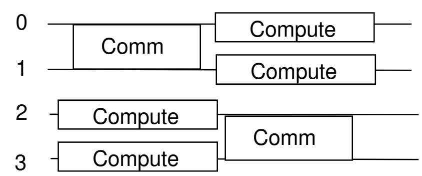
Rendezvous

Contention

- Point-to-point analysis ignores fact that communications links (usually) shared
- Easiest model is to equally share bandwidth (if K can shared at one time, give each 1/K of the bandwidth).
- “Topology doesn’t matter anymore” is *not* true, but there is less you can do about it (just like cache memory)
- MPI has processor topology routines ...

Scheduling for Contention

- Many programs alternate between communication and computation phases
- Contention can reduce effective bandwidth
- Consider restructuring program so that some nodes communicate while others compute:



Effect of contention

- IBM SP2 has a multistage switch. This test shows the point-to-point bandwidth with half the nodes sending and half receiving

Processors	Bandwidth (MB/sec)
2	34
4	34
8	34
16	31
32	25
64	22

IBM SP with TB3 also demonstrates contention effects (as will any multistage system)

MPI Implementation Parameters

- Each MPI implementation makes different choices about protocols, allocation of memory, etc.
- Some of these parameters can be adjusted for each run, some even during a run
- Check vendor documentation. implementation and system, and change
  - Recent changes include thread options on IBM

Buffering-Related Options

- **SGI Options:**
- MPI_BUFFER_MAX
- MPI_BUFFER_TOTAL
- MPI_BUFS_PER_HOST
- MPI_BUFS_PER_PROC
- MPI_MSGS_PER_HOST
- MPI_MSGS_PER_PROC

- Closely related to implementation details
- MPI_THRESHOLD in previous versions

- **IBM Options:**
- MP_BUFFER_MEM
- MP_EAGER_LIMIT
- MP_USE_FLOW_CONTROL

Other Important Options

- **SGI Options:**
- MPI_CHECK_ARGS
- MPI_xxxx_MAX
- MPI_DSM_PPM
- MPI_DSM_OFF
- MPI_DSM_MUSTRUN

- **IBM Options:**
- MP_INTRDELAY
- MP_CSS_INTERRUPT
- MP_PRIORITY
- MP_SINGLE_THREAD
- MP_RETRY
- MP_RETRYCOUNT
- MP_PULSE
- MP_EUILIB

Free MPI Implementations

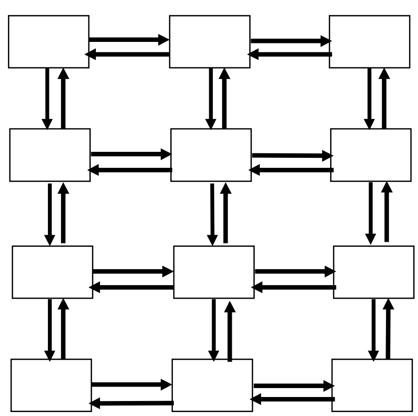
- LAM
  - MPI_GER: Number of envelopes
- MPICH
  - P4_SOCKBUFSIZE: Size of socket buffers
  - P4_GLOBMEMSIZE: Size of shared memory arena

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	Send	Recv
Send	Send	Send	Send	Send	Send		
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
Send	Recv						

Fix 1: Use Irecv

- Do i=1,n_neighbors

```
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)
```

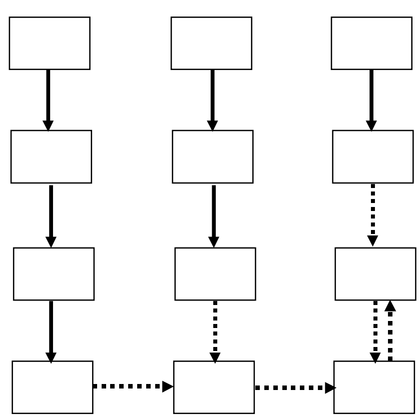
- Does not perform well in practice. Why?

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)

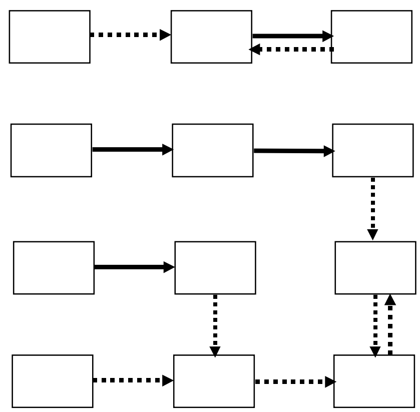
Mesh Exchange - Step 1

- Exchange data on a mesh



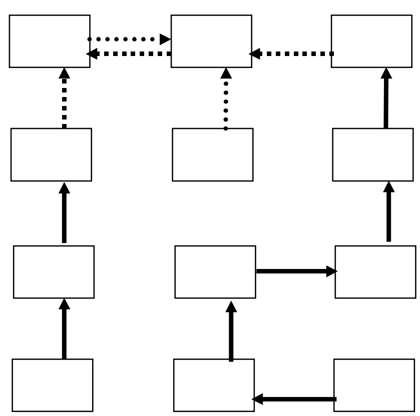
Mesh Exchange - Step 2

- Exchange data on a mesh



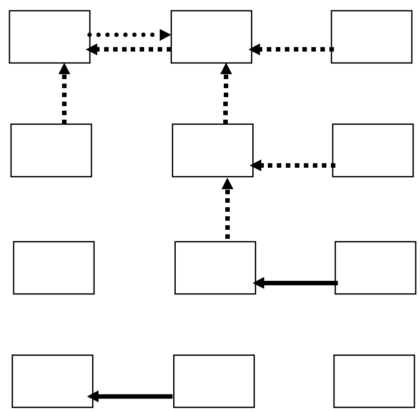
Mesh Exchange - Step 3

- Exchange data on a mesh



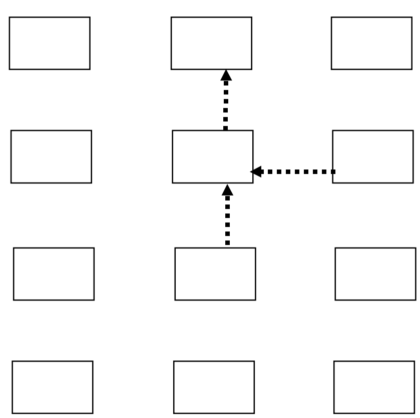
Mesh Exchange - Step 4

- Exchange data on a mesh



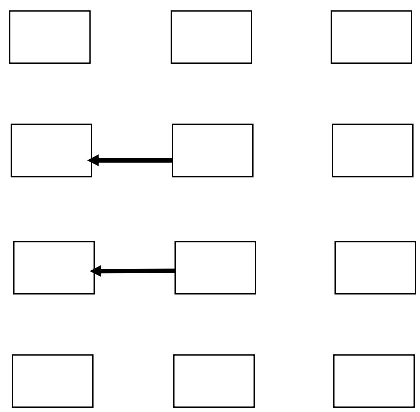
Mesh Exchange - Step 5

- Exchange data on a mesh

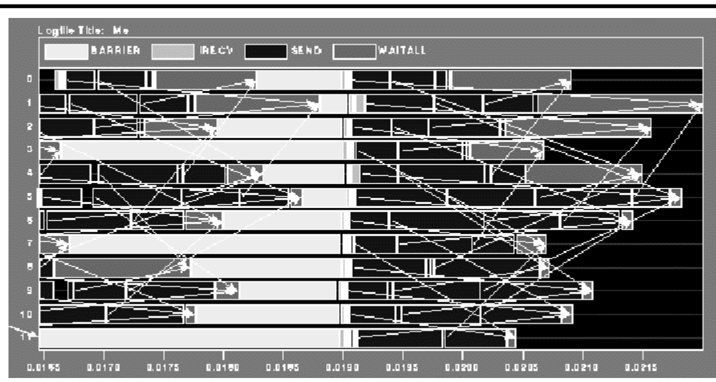


Mesh Exchange - Step 6

- Exchange data on a mesh

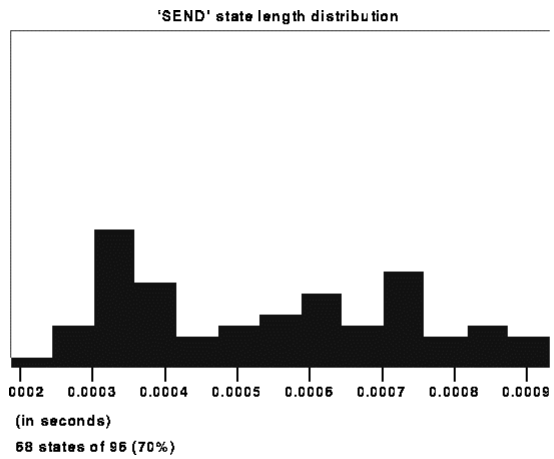


Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

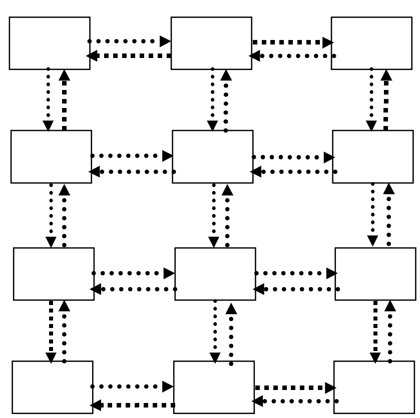
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

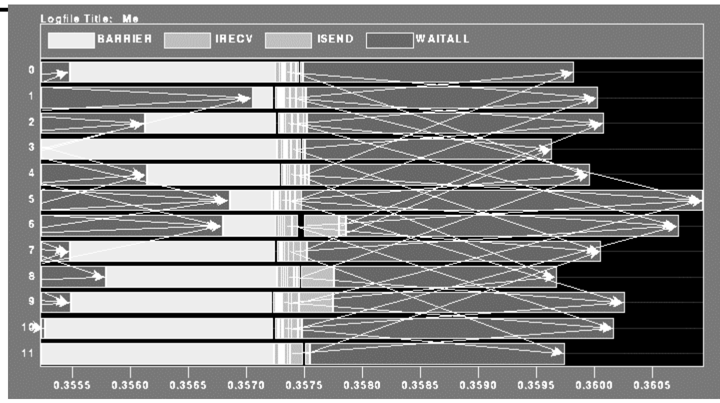
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
  comm, requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)
- (We'll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

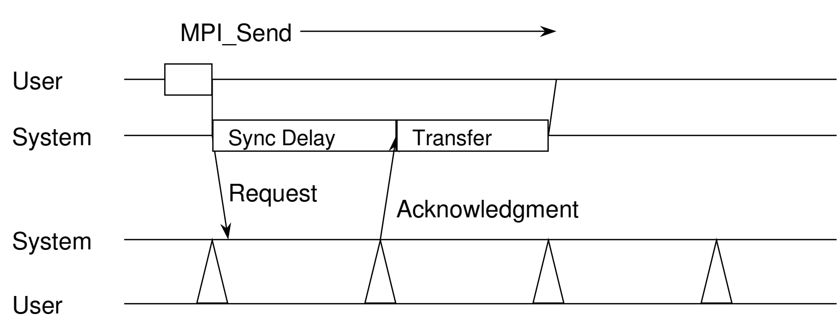
- Write code that has each processor send to all of the other processors. Use MPI_Irecv and MPI_Send. Compare:
  - All processors send to process 0, then process 1, etc., in that order
  - Each process sends to process (myrank+1), then (myrank+2), etc.
  - Compare with the MPI routine MPI_Alltoall
-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

Break

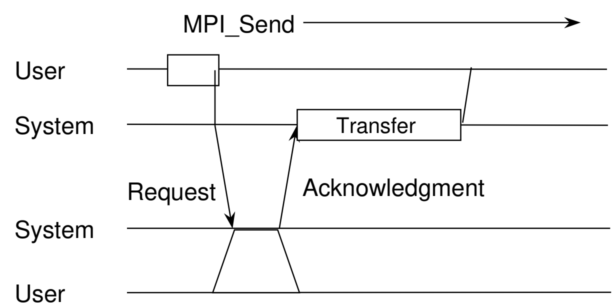
When are control messages handled?

- Polling
  - During MPI calls or periodic “polls”
- Interrupts
  - When message arrives at process (e.g., OS signals an I/O event)
- No method is “best”

Polling Mode MPI



Interrupt Mode MPI



- Cost of interrupt higher than polling (usually)

Multi-threaded MPI Implementations

- Multithreaded MPI implementations (where a separate thread watches for envelopes) can act either like a polling or interrupt-driven implementation:
  - Polling if there is an available (extra) CPU
    - Constantly polling, even when user's main thread is computing
  - Interrupt-driven otherwise
    - OS (or user's process) must schedule thread

Example of the effect of Polling

- IBM SP2 MPI_Allreduce times for each mode

Processors	Polling	Interrupt
2	105	260
4	179	490
8	259	746
16	357	1020
32	468	1302
64	670	1614

- Times in usecs. Similar effects on other operations.
- ***But*** some programs (with extensive computing) can show better performance with interrupt mode

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
  - MPI_Test_cancelled required on IBM (!), then 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 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 (threads discussed later)

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 fro process 2, etc.
  - That order would starve processes 2 and highe 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
```

Test Yourself: Providing Fairness

- Write a program with one master; have each worker process receive 10 work requests and send 10 responses. Make the workers' computation trivial (e.g., just return a counter)
  Use the simple algorithm (MPI_Send/MPI_Recv)
  - Is the MPI implementation fair?
- Write a new version using MPI_Irecv/MPI_Waitall
  - Be careful of uncancelled Irecv requests
  - Be careful of the meaning of array_of_indices (zero versus one origin)

Dinner

MPI Datatypes

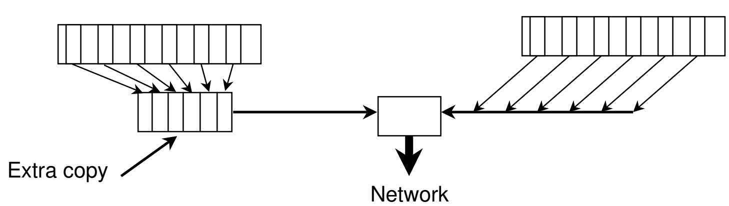
- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided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

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 *allo* MPI implementations to avoid copy



  - Not widely implemented yet
- Some MPI implementations can handle important special cases
  - Constant stride
  - Contiguous structures

Potential Performance Advantage in MPI Datatypes

- Handling non-contiguous data
- Assume must pack/unpack on each end
  - $cn + (s + r\,n) + cn = s + (2c + r)n$
- Can move directly
  - $s + r'\,n$
  - r' probably > r but < $(2c+r)$
- MPI implementation must copy data anyway (into network buffer or shared memory); having the datatype permits removing 2 copies

Performance of MPI Datatypes

- Test of 1000 element vector of doubles with stride of 24 doubles (MB/sec).
  - MPI_Type_vector
  - MPI_Type_struct (MPI_UB for stride)
  - User packs and unpacks by hand

	SP2	**SP3**	**SGI**	**Origin**	**T3D**
Vector	4.0	5.4	6.7	8.0	**12.7**
Struct	3.0	4.3	3.0	3.2	0.4
User	**15**	**7.9**	**40**	**47**	10.8

- Performance very dependent on implementation; should improve with time

Performance of MPI Datatypes II

- Performance of MPI_Type_Vector
  - 1000 doubles with stride 24
  - Performance in MB/sec

Platform	Vendor MPI	MPICH	User
		Equivalent Type_Struct	
IBM SP Blue Pacific	3.79 2.94	6.71 1.02	9.52
SGI O2000 Blue Mountain	7.63 3.09	20.2 2.6	43.8
Intel Tflops Red	32.3 3.29	MPICH *is* Vendor MPI	34.7

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 operation
  - 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

Sending Rows of a Matrix

- From Fortran, assume you want to send a row o the matrix
  A(n,m),
  that is, A(row,j), for j=1,…,
- 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?

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

Solutions for Vectors

- MPI_Type_vector is for very specific uses
  - rarely makes sense to use count other than 1
- Two 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

Sending Vectors of Different Sizes

- How would you send A(i,2:m) and A(i+1,3:m) with a single MPI datatype?
  - Use "count" to select the number, as in
    call MPI_Send(A(i,2),m-1,type,…)
    call MPI_Send(A(i+1,3),m-2,type,…)
- Hint: Use an extent of n elements

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
    ```

Test Your Understanding of Datatypes

- Write a program that sends rows of a matrix from one processor to another. Use both MPI_Type_vector and MPI_Type_struct methods
  - Which is most efficient?
  - Which is easier to use?
- Write a program that sends a matrix from one processor to another. Arrange the datatypes so that the matrix is received in transposed order
  - A(i,j) on sender arrives in A(j,i) on receiver

Break

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/RECEIV`** can be replaced by **`BCAST/REDUC`** , 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.

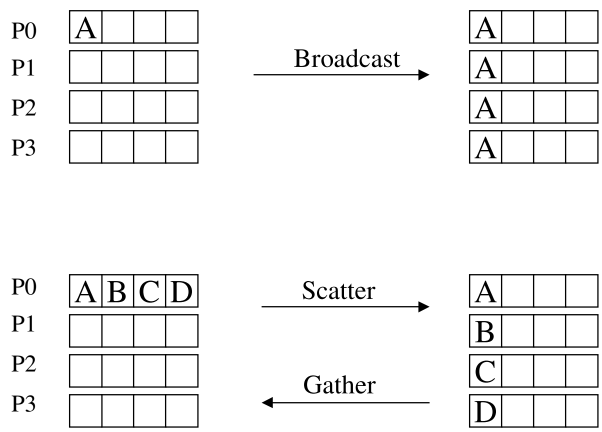
Synchronization

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

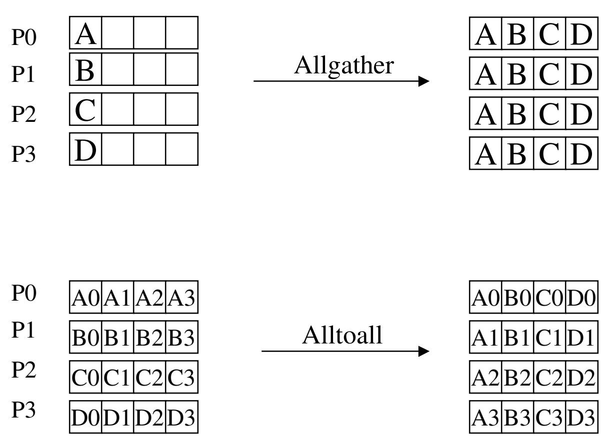
Synchronization

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

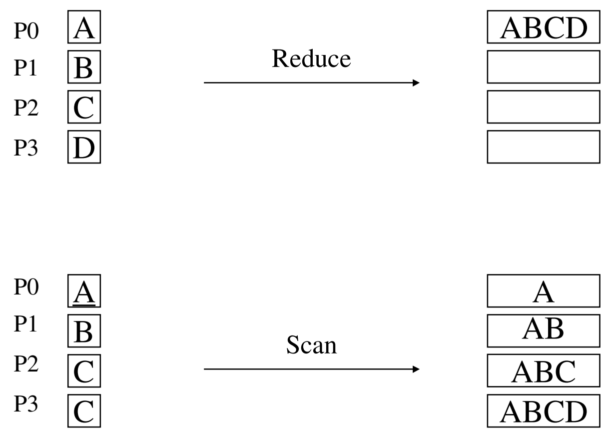
Collective Data Movement



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.

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

How Deterministic are Collective Computations?

- In exact arithmetic, you always get the same results
  - but roundoff error, truncation can happen
- MPI does *not* require that the same input give the same output
  - Implementations are encouraged but not required to provide *exactly* the same output given the same input
  - Round-off error may cause slight differences
- Allreduce does guarantee that the *same* value is received by all processes for each call
- Why didn't MPI mandate determinism?
  - Not all applications need it
  - Implementations can use "deferred synchronization" ideas to provide better performance

Defining your own Collective Operations

- Create your own collective computations with:
```
MPI_Op_create( user_fcn, commutes, &op );
MPI_Op_free( &op );

user_fcn( invec, inoutvec, len, datatype );
```
- The user function should perform:
```
inoutvec[i]  =  invec[i]  op  inoutvec[i];
```
  for i from 0 to len-1.
- The user function can be non-commutative.

Defining your own Collective Operations

- Create your own collective computations with:
```
call MPI_Op_create( user_fcn, commutes, op, ierr )
call MPI_Op_free( op, ierr )

subroutine user_fcn( invec, inoutvec, len, &
      datatype )
```
- The user function should perform:
```
inoutvec(i)  =  invec(i)  op  inoutvec(i)
```
  for i from 1 to len.
- The user function can be non-commutative.

When *not* to use Collective Operations

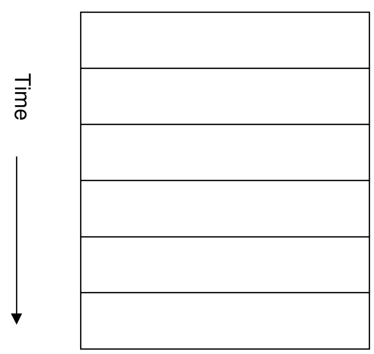
- Sequences of collective communication can be pipelined for better efficiency
- Example: Processor 0 reads data from a file and broadcasts it to all other processes.
  - ```
    Do i=1,m
      if (rank .eq. 0) read *, a
      call mpi_bcast( a, n, MPI_INTEGER, 0, comm, ierr )
    EndDo
    ```
  - Takes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

Pipeline the Messages

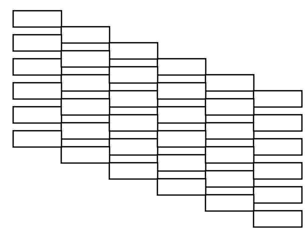
- Processor 0 reads data from a file and sends it to the next process. Other forward the data.
  - ```
    Do i=1,m
      if (rank .eq. 0) then
        read *, a
        call mpi_send(a, n, type, 1, 0, comm,ierr)
      else
        call mpi_recv(a,n,type,rank-1, 0,comm,status, ierr)
        call mpi_send(a,n,type,next, 0, comm,ierr)
      endif
    EndDo
    ```

Concurrency between Steps

- Broadcast:



- Pipeline



Each broadcast takes less time then 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
- Sometimes called "digital orrery"
  - Circular particles in n-body problem
  - Even better performance if pipeline never stops
- "Elegance" of collective routines can lead to fine-grain synchronization
  - and hence a performance penalty

Implementation Variations

- Implementations vary in goals and quality
  - Short messages (minimize separate communication steps)
  - Long messages (pipelining, network topology)
- MPI's general datatype rules make some algorithms more difficult to implement
  - Datatypes can be different on different processes; only the type signature must match

Using Datatypes in Collective Operations

- Datatypes allow noncontiguous data to be moved (or computed with)
- As for all MPI communications, only the *type signature* (basic, language defined types) must match
  - Layout in memory can differ on each process

Example of Datatypes in Collective Operations

- Distribute a matrix from one processor to four
  - Processor 0 gets A(0:n/2,0:n/2),
    Processor 1 gets A(n/2+1:n,0:n/2),
    Processor 2 gets A(0:n/2,n/2+1:n),
    Processor 3 get A(n/2+1:n,n/2+1:n)
- Scatter (One to all, different data to each)
  - Data at source is not contiguous (n/2 numbers, separated by n/2 numbers)
  - Use vector type to represent submatrix

Matrix Datatype

- MPI_Type_vector(n/2 per block, n/2 blocks, dist from beginning of one block to next = n, MPI_DOUBLE_PRECISION, &subarray_type)
- Can use this to send
  - ```
    Do j=0,1
        Do i=0,1
         call MPI_Send( a(1+i*n/2:i*n/2+n/2,
                              1+j*n/2:j*n/2+n/2),1,
                             subarray_type, … )
    ```
  - Note sending **ONE** type contain multiple basic elements
    ```

Scatter with Datatypes

- Scatter is like
  - Do i=0,p-1
    call mpi_send(a(1+i*extent(datatype)),….)
    - “1+” is from 1-origin indexing in Fortran
  - Extent is the distance from the beginning of the first to the end of the last data element
  - For subarray_type, it is
    ((n/2-1)n+n/2) * extent(double)
  - “extent(double)” is simply the number of bytes in DOUBLE PRECISION item (often 8)
    - You can use
      MPI_Type_size(MPI_DOUBLE_PRECISION, extent, ierr)
      to get this value

Layout of Matrix in Memory

N = 8 example

Process 0									Process 2
	0	8	16	24	32	40	48	56	
	1	9	17	25	33	41	49	57	
	2	10	18	26	34	42	50	58	
	3	11	19	27	35	43	51	59	
Process 1									**Process 3**
	4	12	20	28	36	44	52	60	
	5	13	21	29	37	45	53	61	
	6	14	22	30	38	46	54	62	
	7	15	23	31	39	47	55	63	

Using MPI_UB

- Set Extent of each datatype to n/2
  - Size of contiguous block all are built from
- Use Scatterv (independent multiples of extent)
- Location (beginning location) of blocks
  - Processor 0: 0 * 4
  - Processor 1: 1 * 4
  - Processor 2: 8 * 4
  - Processor 3: 9 * 4
- MPI-2: Use MPI_Type_create_resized instead

Changing Extent

- MPI_Type_struct
  - ```
    types(1)    = subarray_type
    types(2)    = MPI_UB
    displac(1) = 0
    displac(2) = (n/2) * 8    ! Bytes!
    blklens(1) = 1
    blklens(2) = 1
    call MPI_Type_struct( 2, blklens, displac, types,&
          newtype, ierr )
    ```
- `newtype` contains all of the data of `subarra _type`.
  - Only change is “extent,” which is used *only* when computing where in a buffer to get or put data relative to other data
    ```

Scattering A Matrix

- sdisplace(1) = 0
  sdisplace(2) = 1
  sdisplace(3) = n
  sdisplace(4) = n + 1
  scounts(1,2,3,4)=1
  call MPI_Scatterv(a, scounts, sdispls, newtype,&
  alocal, n*n/4, MPI_DOUBLE_PRECISION,&
  0, comm, ierr)
  - Note that process 0 sends 1 item of newtype but all processes receive $n^2/4$ double precision elements
- Test youself: Work this out and convince yourself that it is correct

Dense Matrix Vector Multiply

- Let the matrix be partitioned across processes by columns, and the vector by corresponding rows.
  - If process I has columns M:N of the matrix, it also has elements M:N of the vectors
  - Simple partition (process 0 has the first block of columns, process 1 the second block, etc.)
  - process i has columns col(i):col(i+1)-1
- Problem: Compute the matrix-vector product with the distributed data structures
  - Send/receive requires intermediate buffers
  - Collective solution

Using MPI_Reduce_scatter

- Each process needs to accumulate a contribution from every process to its part of the result vector

```
do i=1,p
   recvcounts(i) = col(i+1)-col(i)
enddo
do j=1,n
   sum = 0
   do k=1, recvcounts(myrank)
       sum = sum + mv(j,k) * v(k)
   enddo
    localmv(j) = sum
enddo
call MPI_Reduce_scatter( localmv, my_vec, recvcounts, &
          MPI_DOUBLE_PRECISION, MPI_SUM, comm, ierr )
```

Matrix times vector

Meaning of Reduce Scatter

- Reduce_scatter
  - Scatters contributions from all processes to all others
  - Combines (reduces) incoming contributions into a single buffer

Why Contexts?

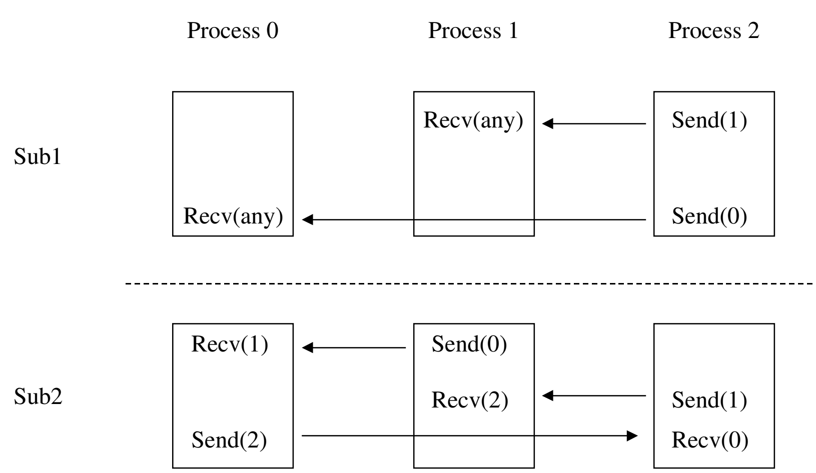
- Parallel libraries require isolation of messages from one another and from the user that cannot be adequately handled by tags.
- The following examples are due to Marc Snir.
  - Sub1 and Sub2 are from different libraries

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

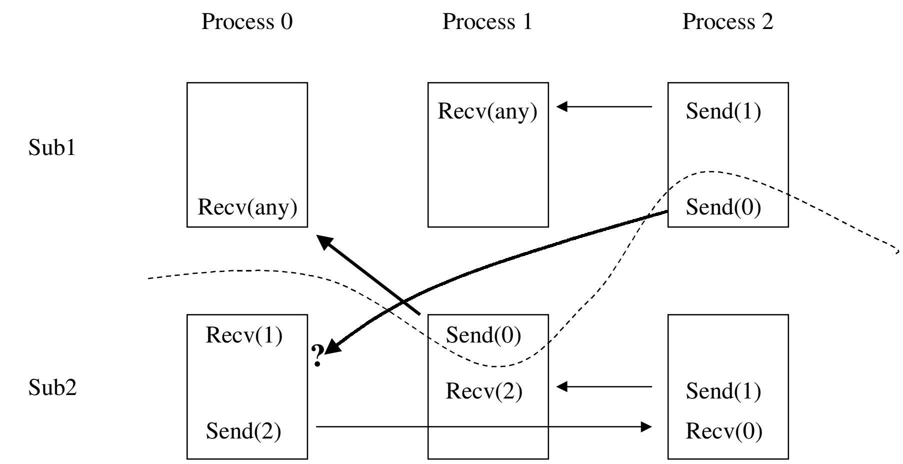
  - Sub1a and Sub1b are from the same library

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

Correct Execution of Library Calls

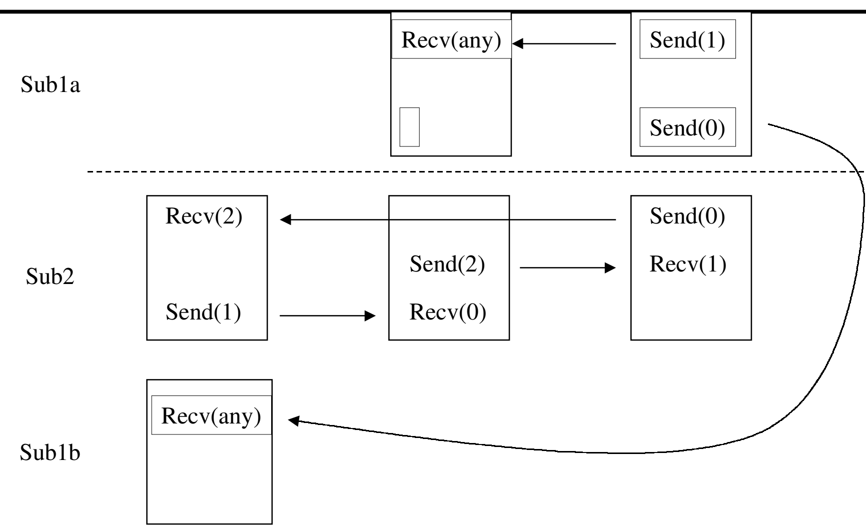


Incorrect Execution of Library Calls

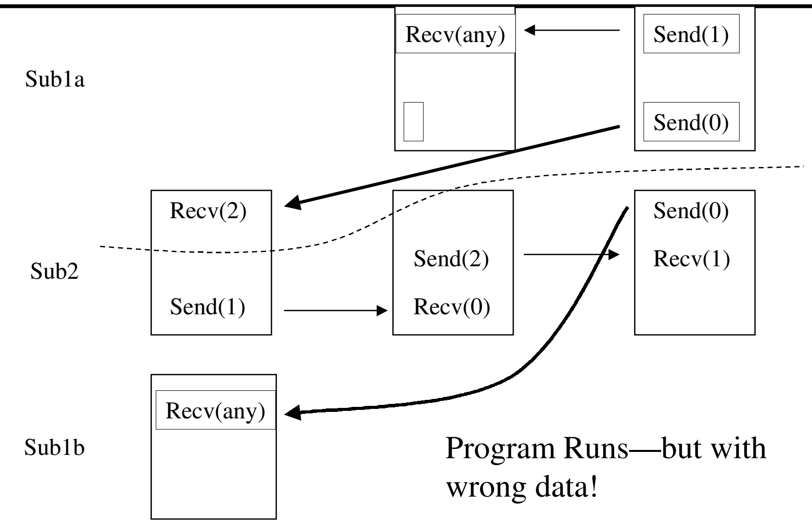


Program hangs (Recv(1) never satisfied)

Correct Execution of Library Calls with Pending Communication



Incorrect Execution of Library Calls with Pending Communication



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 UNET, VIA, 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

Using Persistent Operations

- Replace
  MPI_Isend(buf, count, datatype, tag, dest, comm,
  request, ierr)
  with
  MPI_Send_init(buf, count, datatype, tag, dest, comm,
  request, ierr)
  MPI_Start(request, ierr)
- 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 interative 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

- Do i=1,n
  ! Exchange vector information
  Call MPI_Isend(…)
  Call MPI_Irecv(…)
  Call MPI_Waitall(…)
- Replace with
  Call MPI_Send_init(…)
  Call MPI_Recv_init(…)
  Do i=1,n
  Call MPI_Startall(2, requests, ierr)
  Call MPI_Waitall(2, requests, statuses, ierr)
  EndDo
  Call MPI_Request_free(request(1), ierr)
  Call MPI_Request_free(request(2), ierr)

Identical arguments

Choosing MPI Alternatives

- MPI offers may ways to accomplish the same task
- Which is best?
  - Just like everything else, it depends on the vendor, system architecture
  - Like C and Fortran, MPI provides the programmer with the tools to achieve high performance without sacrificing portability
- The best choice depends on the use:
  - Consider choices based on system and MPI implementation
  - Example: Experiments with a Jacobi relaxation example

Tuning for MPI's Send/Receive Protocols

- Aggressive Eager
  - Performance problem: extra copies
  - Possible deadlock for inadequate eager buffering
  - Ensure that receives are posted before sends
  - MPI_Issend can be used to express "wait until receive is posted"
- Rendezvous with sender push
  - Extra latency
  - Possible delays while waiting for sender to begin
- Rendezvous with receiver pull
  - Possible delays while waiting for receiver to begin

Rendezvous Blocking

- What happens once sender and receiver rendezvous?
  - Sender (push) or receiver (pull) may complete operation
  - May block other operations while completing
- Performance tradeoff
  - If operation does *not* block (by checking for other requests), it adds latency or reduces bandwidth.
- Can reduce performance if a receiver, having acknowledged a send, must wait for the sender to complete a separate operation that it has started.

Tuning for Rendezvous with Sender Push

- Ensure receives posted before sends
  - better, ensure receives match sends before computation starts; may be better to do sends before receives
- Ensure that sends have time to start transfers
- Can use short control messages
- Beware of the cost of extra messages
  - Intel i860 encouraged use of control messages with ready send (force type)

Tuning for Rendezvous with Receiver Pull

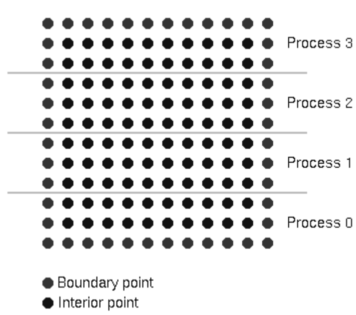
- Place MPI_Isends before receives
- Use short control messages to ensure matches
- Beware of the cost of extra messages

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)
- Others at http://www.mcs.anl.gov/mpi/tutorials/perf

Jacobi Iteration (C Ordering)

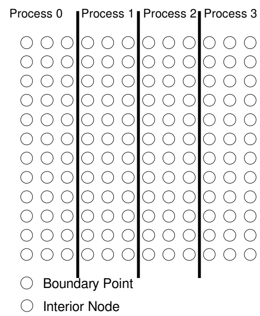
- Simple parallel data structure



- 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 (EPCC/T3D can prefer Ssend over Send; uses Send for buffered send)
  - Manually advance automaton
- 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
- More details at http://www.mcs.anl.gov/mpi/tutorial/perf/mpiexmpl/src3/runs.html

Send and Recv

- 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

- 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

- 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 );
}
```

Better to start receives first

- 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_Waital ( 4, requests, statuses, ierr )
```

Ensure recvs posted before sends

- 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] );
}
```

Ensure recvs posted before sends

- Irecv, Sendrecv/Barrier, Rsend, Waitall

```
integer statuses(MPI_STATUS_SIZE,2), requests(2)
! Post initial Irecv's (not shown)
do while (.not. Converged)
  ….
  call MPI_Rsend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                 right_nbr, ring_comm, ierr )
  call MPI_Rsend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                 left_nbr, ring_comm, ierr )
  call MPI_Waitall( 2, requests, statuses, ierr )
  ! Process ghost points
  call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION,&
                 left_nbr, ring_comm, requests(1), ierr )
  call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION,&
                 right_nbr, ring_comm, requests(2), ierr )
  call MPI_Allreduce( … )
enddo
```

Use of Ssend versions

- 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 );
}
```

Use of Ssend versions

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

```
call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, left_nbr, 0,&
                ring_comm, request, ierr )
call MPI_Ssend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, right_nbr, &
                0, ring_comm, ierr )
call MPI_Wait( request, status, ierr )
call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                right_nbr, 0, ring_comm, request, ierr )
call MPI_Ssend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, left_nbr, &
                0, ring_comm, ierr)
call MPI_Wait( request, status, ierr )
```

Nonblocking Operations, Overlap Effective

- Isend, Irecv, Waitall
- A variant uses Waitsome with computation

```
void ExchangeStart( mesh )
Mesh *mesh;
{
    /* Send up, then receive from below */
   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] );
   MPI_Isend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
              ring_comm, &mesh->rq[2] );
   MPI_Isend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm,
              &mesh->rq[3] );
}

void ExchangeEnd( mesh )
Mesh *mesh;
{
   MPI_Status statuses[4];
   MPI_Waitall( 4, mesh->rq, statuses );
}
```

Nonblocking Operations, Overlap Effective

- Isend, Irecv, Waitall
- A variant uses Waitsome with computation

```
integer statuses(MPI_STATUS_SIZE,4), requests(4)
call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, left_nbr, 0,&
                ring_comm, requests(1), ierr )
call MPI_Isend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, right_nbr, &
                0, ring_comm, requests(2), ierr )
call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                right_nbr, 0, ring_comm, requests(3), ierr )
call MPI_Isend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, left_nbr, &
                0, ring_comm, requests(4), ierr)
… computation ...
call MPI_Waital ( 4, requests, statuses, ierr )
```

Persistent Operations

- Potential saving
  - Allocation of MPI_Request
  - Validating and storing arguments
  - Fewer interactions with “polling” engine
- Variations of example
  - sendinit, recvinit, startall, waitall
  - startall(recvs), sendrecv/barrier, startall(rsends), waitall
- Some vendor implementations are buggy
- Persistent operations may be slightly *slower*
  - if vendor optimizes for non-persistent operations

Polling Engine

- Some implementations poll for incoming control message on “important” MPI calls
  - Each polling operation costs time
  - HP/Convex is one vendor implementation that does this
- Operations that *create* persistent operations don’t poll

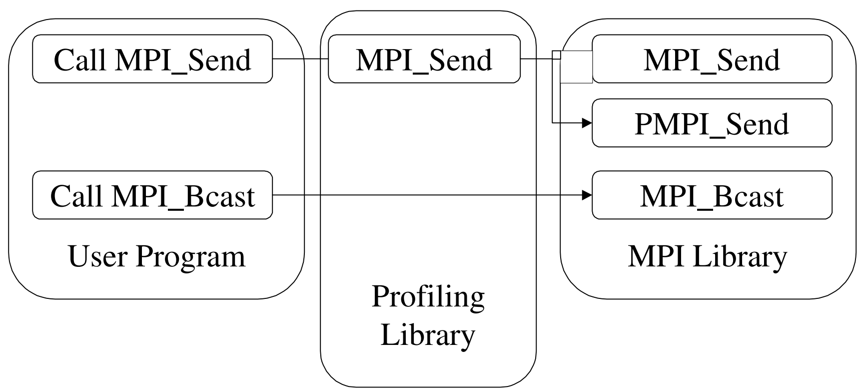
Summary of Results

- Better to start sends before receives
  - Most implementations use rendezvous protocols for long messages (Cray, IBM, SGI, MPICH, LAM)
- 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

Using PMPI routines

- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- Some libraries already make use of PMPI
- Some MPI implementations have PMPI bugs
  - IBM — PMPI_Wtime() returns 0
  - Cray — PMPI in a separate library that som installations have not installed

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

- Upshot, Jumpshot, and MPE tools
  http://www.mcs.anl.gov/mpi/mpich
- Pallas VAMPIR
  http://www.pallas.com/
- IBM VT
  http://www.rs6000 ibm.com/software/sp_products/pe.html
- Pablo http://www-pablo.cs.uiuc.edu/Projects/Pablo/pablo.html
- Paragraph
  http://www.ncsa.uiuc.edu/Apps/MCS/ParaGraph/ParaGraph.html
- Paradyn http://www.cs.wisc.edu/~paradyn
- Many other vendor tools exist
  - e.g., xmpi (SGI and HP)

Lunch

MPI-2: What is it, Where is, and When will it be?

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

Reaction to MPI-1

- Initial public reaction:
  - It's too big!
  - It's too small!
- Implementations appeared quickly
  - Freely available (MPICH, LAM, CHIMP) helped expand the user base
  - MPP vendors (IBM, Intel, Meiko, HP-Convex, SGI, Cray) found they could get high performance from their machines with MPI.
- MPP users:
  - quickly added MPI to the set of message-passing libraries they used;
  - gradually began to take advantage of MPI capabilities.
- MPI became a requirement in procurements.

1995 OSC Users Poll Results

- Diverse collection of users
- All MPI functions in use, including "obscure" ones.
- Extensions requested:
  - parallel I/O
  - process management
  - connecting to running processes
  - put/get, active messages
  - interrupt-driven receive
  - non-blocking collective
  - C++ bindings
  - Threads, odds and ends

MPI-2 Origins

- Began meeting in March 1995, with
  - veterans of MPI-1
  - new vendor participants (especially Cray and SGI, and Japanese manufacturers)
- Goals:
  - Extend computational model beyond message-passing
  - Add new capabilities
  - Respond to user reaction to MPI-1
- MPI-1.1 released in June, 1995 with MPI-1 repairs, some bindings changes
- MPI-1.2 and MPI-2 released July, 1997
  - http://www.mpi-forum.org

Contents of MPI-2

- Extensions to the message-passing model
  - Dynamic process management
  - One-sided operations
  - **Parallel I/O**
- Making MPI more robust and convenient
  - C++ and Fortran 90 bindings
  - External interfaces, handler
  - Extended collective operations
  - Language interoperability
  - MPI interaction with threads

Informal Status Assessment

- Released July, 1997
- All MPP vendors now have MPI-1. (1.0, 1.1, or 1.2)
- Free implementations (MPICH, LAM, CHIMP) support heterogeneous workstation networks.
- MPI-2 implementations are being undertaken now by all vendors.
  - Fujitsu has a complete MPI-2 implementation
- MPI-2 is harder to implement than MPI-1 was.
- MPI-2 implementations appearing piecemeal, with I/O first.
  - I/O available in most MPI implementations
  - One-sided available in some (e.g., HP and Fujitsu)

MPI-2 Summary

- MPI-2 provides major extensions to the original message-passing model targeted by MPI-1.
- MPI-2 can deliver to libraries and applications portability across a diverse set of environments.
- Implementations are under way.
- The MPI standard documents are available at
  http://www.mpi-forum.org

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

Overview of I/O Material

- Unix I/O API and its performance implications
- MPI-IO and its solution to Unix I/O performance problems
- The ROMIO MPI-IO implementation and performance results

Unix I/O API

- Most parallel file systems have a Unix-like API
  - *open* Open a file
  - *lseek* Move the file pointer to a particular offset in the file
  - *read* Read *n* bytes starting from the current location of the file pointer
  - *write* Write *n* bytes starting from the current location of the file pointer
  - *close* Close the file

Performance

- Open
  - usually very expensive
  - can take more than 1 second
- Seek
  - performance depends on implementation
  - very expensive on Paragon, inexpensive on SP
- Read/Write
  - Performs well if I/O size is large -- 1MB or more
  - Performs very poorly if I/O size is small -- 8KB or less
- Close
  - not as expensive as open

Some Other I/O Functions

- *readv, writev*
  - Reads into/writes from multiple buffers in memory
  - In the file, however, data is assumed to be contiguously located
  - Most implementations have a limit of 16 on the number of distinct memory buffers in one call
  - Of limited use because users need to specify noncontiguity in the file more often than in memory

Asynchronous I/O

- *aio_read, aio_write* in POSIX
  - nonblocking: user can test for completion or have the implementation issue a signal
  - most implementations have a limit on the maximum number of concurrently active asynchronous operations; the limit is a small number, but not predefined
  - performance is not good enough yet

List I/O

- *lio_listio* in POSIX
  - user can specify a list of operations at a time
  - can be a mixture of reads and writes
  - each operation is internally performed as a separate *aio_read*, *aio_write*
  - user can specify if *lio_listio* should return immediately (nonblocking) or return after completion of all requests
- Drawbacks:
  - it doesn't treat the list as a *single* request
  - implementation can't perform data sieving
  - no notion of collective I/O
  - not implemented everywhere

I/O Access Patterns in Parallel Applications

- Different from those in sequential programs
- Sequential programs typically access data in contiguous chunks
- In many parallel programs, each process may need to access several noncontiguous pieces of data from a file
  - e.g., bytes 0--10, 20--30, 40--50, ...
- In addition, groups of processes may need to access the file simultaneously, and the accesses of different processes may be interleaved in the file

Problems with Unix API for Parallel I/O

- Noncontiguous accesses in the file cannot be expressed in a single function call
- Each contiguous piece must be accessed separately, resulting in too many system calls and poor performance
- No notion of collective I/O

Why MPI is a Good Setting for Parallel I/O

- Writing is like sending a message and reading is like receiving
- Any parallel I/O system will need a mechanism to:
  - specify groups of processes for collective operations
  - specify noncontiguous accesses
  - test/wait for completion of nonblocking operations
- That is, it needs lots of MPI-like machinery

MPI-IO Background

- Marc Snir (IBM Watson) paper exploring MPI as context for parallel I/O (1994, after MPI-1)
- MPI-IO e-mail discussion group led by Jean-Pierre Prost (IBM) and Bill Nitzberg (NASA)
- MPI-IO group joins MPI-2 Forum in June 1996
- MPI-2 standard released in July 1997

Introduction to MPI-IO

- MPI-IO can be considered as Unix I/O plus (lots of) other stuff
- Basic operations:
  `MPI_File_{open, close, read, write, seek}`
- Parameters to these operations match Unix, aiding straightforward port from Unix I/O to MPI-IO
- However, to get performance and portability, more advanced features must be used

MPI-IO Features

- Noncontiguous access in both memory and file
- Use of explicit offset
- Individual and shared file pointers
- Nonblocking I/O
- Collective I/O
- File interoperability
- Portable data representation
- Mechanism for providing hints applicable to a particular implementation and I/O environment (e.g. number of disks, striping factor): **`info`**

MPI Datatypes

- In MPI-1, the data to be sent or received is described by a triple (address, count, datatype)
- Datatypes in MPI are of two kinds: basic and derived
- Basic datatypes are those that correspond to the basic datatypes in the host programming language, e.g., MPI_INT, MPI_FLOAT, MPI_DOUBLE_PRECISION, …
- Derived datatypes can be constructed out of multiple copies of basic datatypes with gaps

Derived Datatype Constructors

- Datatype constructor functions can be used recursively to create
  - a contiguous array of datatypes
  - a strided array of blocks of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
  - datatypes describing portions of matrices
- Therefore, any noncontiguous access pattern can be represented in terms of an MPI derived datatype

Noncontiguous Accesses

- Noncontiguous data in both memory and file is specified using MPI datatypes, both predefined and derived
- Data layout in memory specified on each call, as in message-passing
- Data layout in file is defined by a file *view*
- A process can access data only within its view
- Views can be changed

Basic Data Access Functions

- Individual file pointer: **`MPI_File_read`**
- Explicit file offset**:** **`MPI_File_read_at`**
- Shared file pointer: **`MPI_File_read_shared`**
- Nonblocking I/O: **`MPI_File_iread`**
- Similarly for writes

Collective I/O Operations

- Blocking:

  `MPI_File_read_all`

- Non-blocking:

  `MPI_File_read_all_begin`

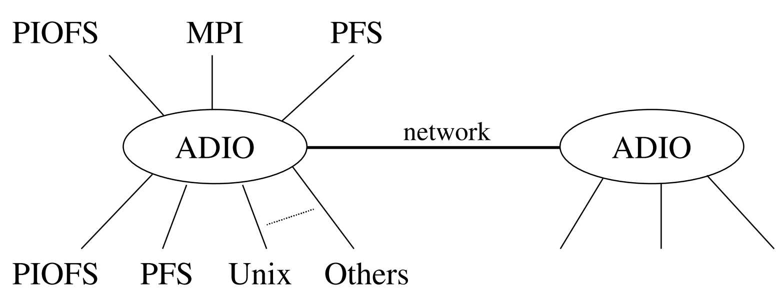
  `MPI_File_read_all_end`

Some Other MPI-IO Features

- Multiple file data representations
  - native
  - internal
  - portable
- Built-in file type constructors for distributed arrays and subarrays.

ROMIO - A Portable Implementation of MPI-IO

- Implementation strategy: an abstract device for I/O (ADIO)
- Tested for low overhead
- Can use any MPI implementation (MPICH, vendor)



Two-Phase Collective I/O

- ROMIO has an optimized implementation of two-phase collective I/O
- I/O is done in two phases: an I/O phase and a communication phase
- In the I/O phase, data is read/written in large chunks to minimize I/O latency
- Message-passing among compute nodes is used to redistribute data as needed

Current Status of ROMIO

- Latest version, 1.0.1, released in July 1998
- A substantial portion of the standard has been implemented:
  - noncontiguous accesses in memory and file
  - collective I/O
  - hints
  - asynchronous I/O
- Supports large files --- greater than 2 Gbytes
- Works with MPICH and vendor MPI implementations (SGI, HP, NEC)
- Included as part of HP MPI and MPICH
- Web site: http://www.mcs.anl.gov/romio

Hints used in ROMIO

- cb_buffer_size
- cb_nodes
- striping_unit
- striping_factor

MPI-2 predefined hints

- ind_rd_buffer_size
- ind_wr_buffer_size

New Algorithm Parameters

- start_iodevice
- pfs_svr_buf

Platform-specific hints

Using MPI-IO Functions

- 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
- We classify the different ways of expressing I/O access patterns in MPI-IO into four "levels", level 0 -- level 3. (SC98 paper)
- Each level represents increasing amounts of data per I/O request

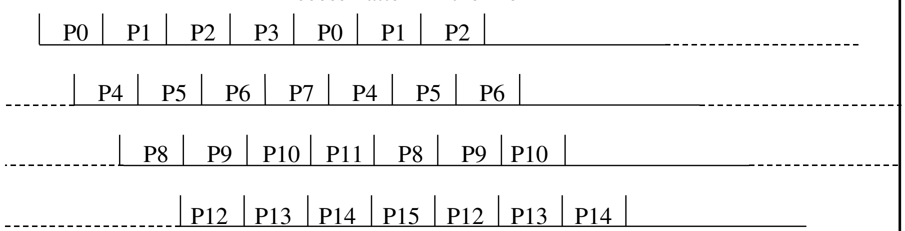
Example: Distributed Array Access

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



Level-0 Access

- 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, row[i], ...)
}
MPI_File_close(&fh)
```

Level-0 Access

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

```
call MPI_File_open(..., file, ..., fh, ierr)
do i=1, n_local_cols
    call MPI_File_seek(fh, ...)
    call MPI_File_read(fh, a(1,i), ...)
enddo
call MPI_File_close(fh, ierr)
```

Level-1 Access

- 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, row[i], ...)
}
MPI_File_close(&fh)
```

Level-1 Access

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

```
call MPI_File_open(MPI_COMM_WORLD, file, ..., fh, ierr)
do i=1, n_local_cols
    call MPI_File_seek(fh, ...)
    call MPI_File_read_all(fh, a(1,i), ...)
enddo
call MPI_File_close(fh, ierr)
```

Level-2 Access

- 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, localarray, ...)
MPI_File_close(&fh)
```

Level-2 Access

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

```
call MPI_Type_create_subarray(..., subarray, ...)
call MPI_Type_commit(subarray, ierr)
call MPI_File_open(..., file, ..., fh, ierr)
call MPI_File_set_view(fh, ..., subarray, ...)
call MPI_File_read(fh, localarray, ...)
call MPI_File_close(fh, ierr)
```

Level-3 Access

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

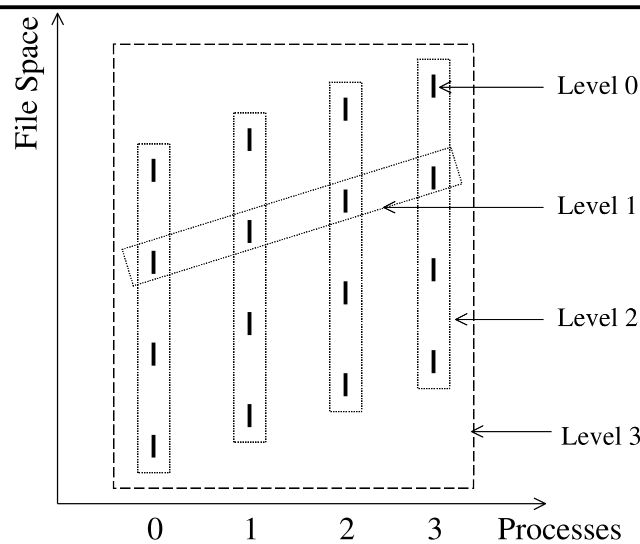
```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read_all(fh, localarray, ...)
MPI_File_close(&fh)
```

Level-3 Access

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

```
call MPI_Type_create_subarray(..., subarray, ...)
call MPI_Type_commit(subarray, ierr)
call MPI_File_open(MPI_COMM_WORLD, file, ..., fh, ierr)
call MPI_File_set_view(fh, ..., subarray, ...)
call MPI_File_read_all(fh, localarray, ...)
call MPI_File_close(fh, ierr)
```

The Four Levels of Access



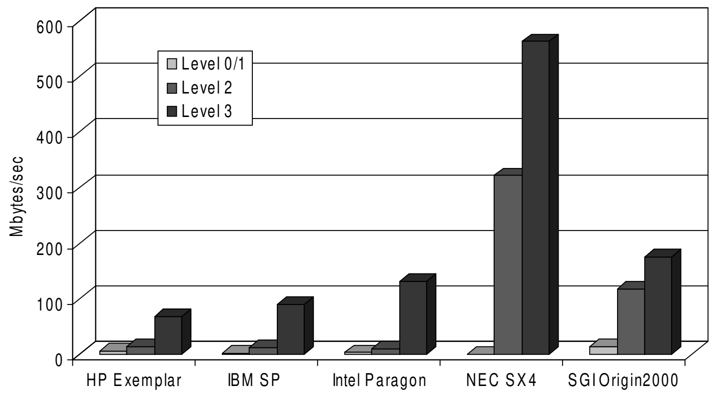
Optimizations

- 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

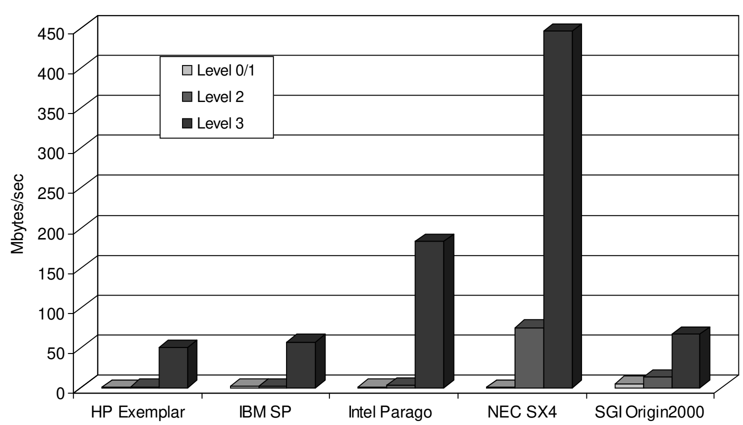
Distributed Array Access: Read Bandwidth



64 procs 64 procs 256 procs 8 procs 32 procs

Array size: 512 x 512 x 512

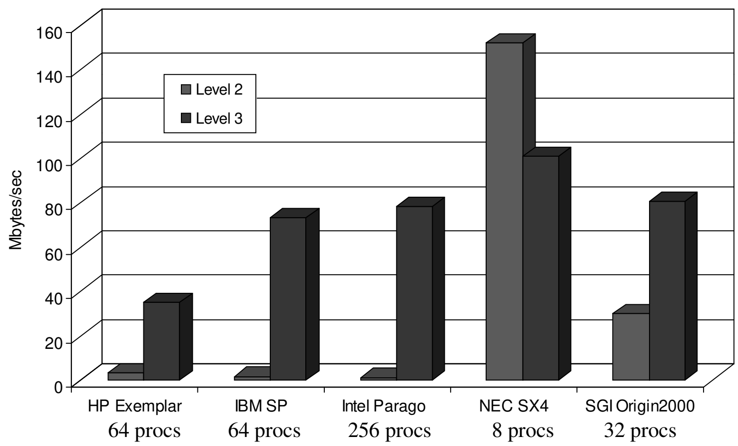
Distributed Array Access: Write Bandwidth



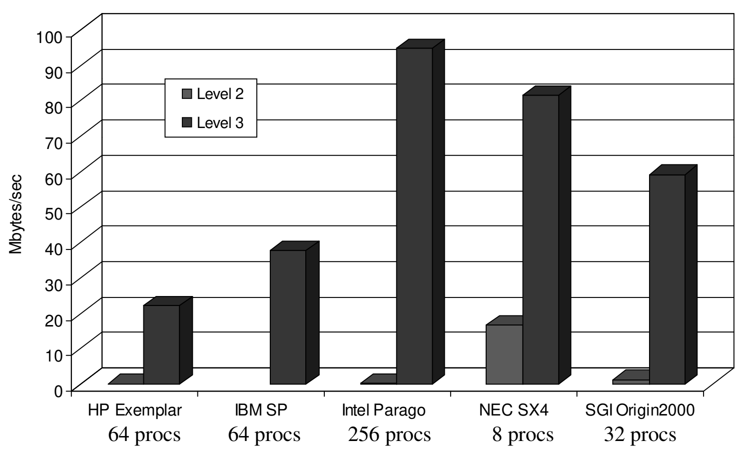
64 procs 64 procs 256 procs 8 procs 32 procs

Array size: 512 x 512 x 512

Unstructured Code: Read Bandwidth

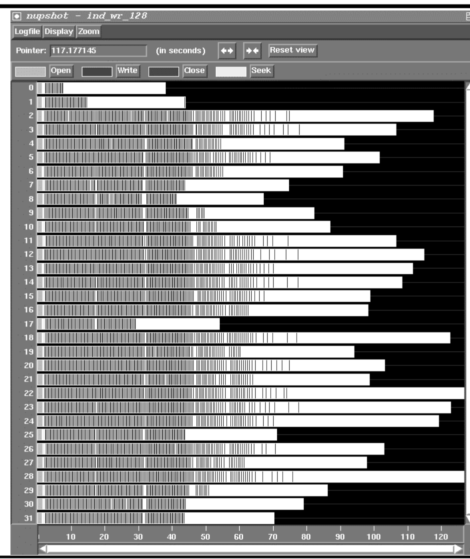


Unstructured Code: Write Bandwidth



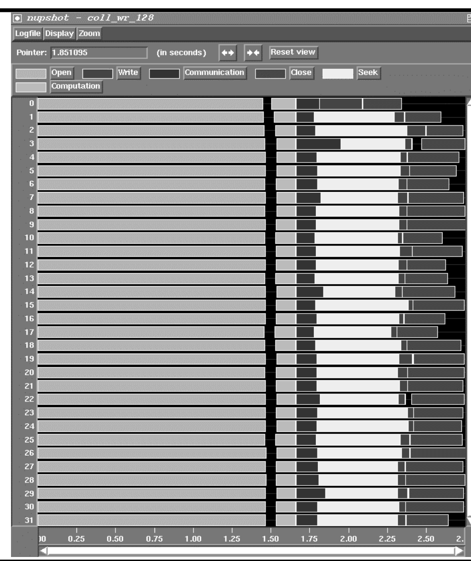
Independent Writes

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



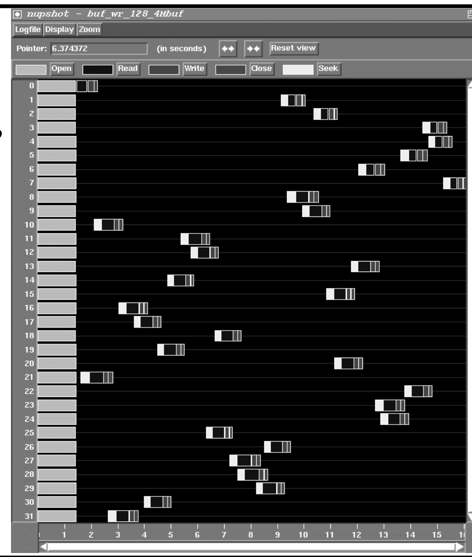
Collective Write

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



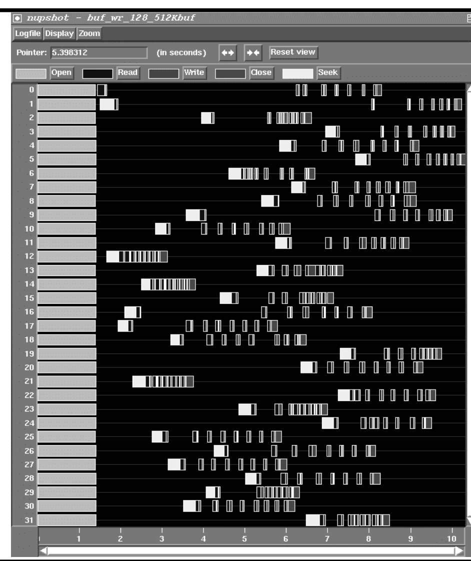
Independent Writes with Data Sieving

- On Paragon
- Use large blocks, write multiple "real" blocks plus "gaps"
- Requires lock, read, modify, write, unlock for writes
- 4 MB blocks
- Time = 16 seconds



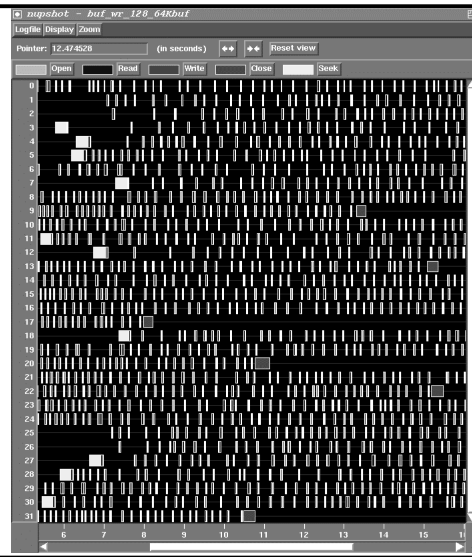
Changing the Block Size

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



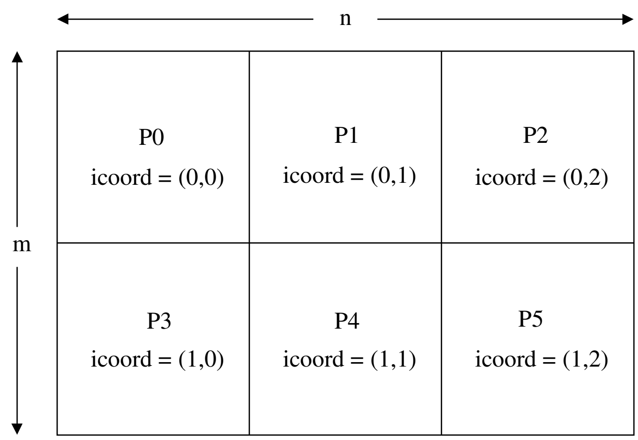
Data Sieving with Small Blocks

- 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



Detailed Example: Distributed Array Access

- Global array size: *m* x *n*
- Array is distributed on a 2D grid of processes: *nproc(1) x nproc(2)*
- Coordinates of a process in the grid: *icoord(1), icoord(2)* (0-based indices)
- Local array stored in file in a layout corresponding to global array in column-major (Fortran) order
- Two cases
  - local array is stored contiguously in memory
  - local array has "ghost area" around it in memory



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

CASE I: LOCAL ARRAY CONTIGUOUS IN MEMORY

```
iarray_of_sizes(1) = m
iarray_of_sizes(2) = n
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = icoord(1) * iarray_of_subsizes(1)      ! 0-based, even in Fortran
iarray_of_starts(2) = icoord(2) * iarray_of_subsizes(2)      ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
        iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, ifiletype, ierr)
call MPI_TYPE_COMMIT(ifiletype, ierr)

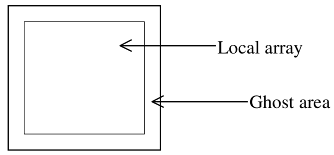
call MPI_FILE_OPEN(MPI_COMM_WORLD, '/home/thakur/test',
        MPI_MODE_CREATE + MPI_MODE_RDWR, MPI_INFO_NULL, ifh, ierr)

call MPI_FILE_SET_VIEW(ifh, 0, MPI_REAL, ifiletype, 'native', MPI_INFO_NULL, ierr)

ilocal_size = iarray_of_subsizes(1) * iarray_of_subsizes(2)
call MPI_FILE_WRITE_ALL(ifh, local_array, ilocal_size, MPI_REAL, istatus, ierr)

call MPI_FILE_CLOSE(ifh, ierr)
```

CASE II: LOCAL ARRAY HAS GHOST AREA OF SIZE 2 ON EACH SIDE




```
!  Create derived datatype describing layout of local array in memory

iarray_of_sizes(1) = m/nproc(1) + 4
iarray_of_sizes(2) = n/nproc(2) + 4
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = 2                  ! 0-based, even in Fortran
iarray_of_starts(2) = 2                  ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
        iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, imemtype, ierr)
call MPI_TYPE_COMMIT(imemtype, ierr)
```

Continued...

CASE II continued...

```
! Create derived datatype to describe layout of local array in the file

iarray_of_sizes(1) = m
iarray_of_sizes(2) = n
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = icoord(1) * iarray_of_subsizes(1)      ! 0-based, even in Fortran
iarray_of_starts(2) = icoord(2) * iarray_of_subsizes(2)      ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
        iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, ifiletype, ierr)
call MPI_TYPE_COMMIT(ifiletype, ierr)

! Open the file, set the view, and write

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/home/thakur/test',
        MPI_MODE_CREATE + MPI_MODE_RDWR, MPI_INFO_NULL, ifh, ierr)
call MPI_FILE_SET_VIEW(ifh, 0, MPI_REAL, ifiletype, 'native', MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE_ALL(ifh, local_array, 1, imemtype, istatus, ierr)
call MPI_FILE_CLOSE(ifh, ierr)
```

I/O Summary

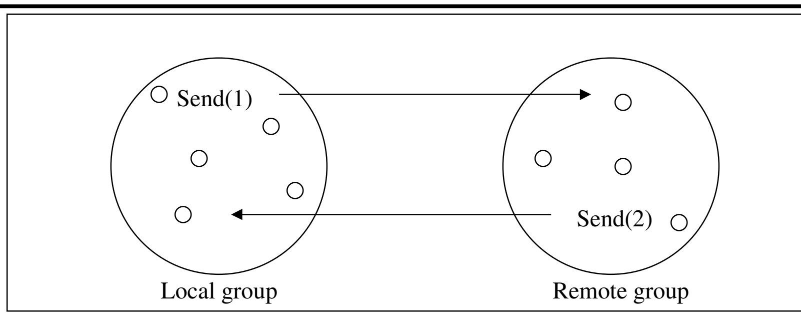
- MPI-IO can help users achieve better I/O performance
- On their part, users must use some of the special features of MPI-IO
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

Break

Intercommunicators

- Contain a *local* group and a *remote* group
- Point-to-point communication is between a process in one group and a process in the other.
- Can be merged into a normal (intra) communicator.
- Created by **`MPI_Intercomm_create`** in MPI-1.
- Play a more important role in MPI-2, created in multiple ways.

Intercommunicators



- In MPI-1, created out of separate intracommunicators.
- In MPI-2, created by partitioning an existing intracommunicator.
- In MPI-2, the intracommunicators may come from different MPI_COMM_WORLDs

Dynamic Process Management

- Issues
  - maintaining simplicity, flexibility, and correctness
  - interaction with operating system, resource manager, and process manager
  - connecting independently started processes
- Spawning new processes is *collective,* returning an intercommunicator.
  - Local group is group of spawning processes.
  - Remote group is group of new processes.
  - New processes have own **`MPI_COMM_WORLD`**.
  - **`MPI_Comm_get_parent`** lets new processes find parent communicator.

Spawning Processes

```
MPI_Comm_spawn(command, argv, numprocs, info,
  root, comm, intercomm, errcodes)
```

- Tries to start **`numprocs`** process running **`command`**, passing them command-line arguments **`argv`**.
- The operation is collective over **`comm`**.
- Spawnees are in remote group of **`intercomm`**.
- Errors are reported on a per-process basis in **`errcodes.`**
- **`Info`** used to optionally specify hostname, archname, wdir, path, file, softness.

Spawning Multiple Executables

- **`MPI_Comm_spawn_multiple( ... )`**

- Arguments **`command, argv, numprocs, info`** all become arrays.
- Still collective

In the Children

- **`MPI_Ini`** (only MPI programs can be spawned)
- **`MP  _COMM_WORLD`** is processes spawned with one call to **`MPI_Comm_spawn.`**
- **`MPI_Comm_get_parent`** obtains parent intercommunicator.
  - Same as intracommunicator returned by **`MP  _Comm_spaw`** in parents.
  - Remote group is spawners.
  - Local group is those spawned.

Manager-Worker Example

- Single manager process decides how many workers to create and which executable they should run.
- Manager spawns n workers, and addresses them as 0, 1, 2, ..., n-1 in new intercomm.
- Workers address each other as 0, 1, ... n-1 in **`MPI_COMM_WORLD`**, address manager as 0 in parent intercomm.
- One can find out how many processes can usefully be spawned.

Establishing Connections

- Two sets of MPI processes may wish to establish connections, e.g.,
  - Two parts of an application started separately.
  - A visualization tool wishes to attach to an application.
  - A server wishes to accept connections from multiple clients. Both server and client may be parallel programs.
- Establishing connections is collective but asymmetric ("Client"/"Server").
- Connection results in an intercommunicator.

Connecting Processes

- Server:
  - **`MPI_Open_port( info, port_name )`**
    - system supplies port_name
    - might be host:num; might be low-level switch #
  - **`MPI_Comm_accept( port_name, info, root, comm, intercomm )`**
    - collective over comm
    - returns intercomm; remote group is clients
- Client:
  - **`MPI_Comm_connect( port_name, info, root, comm, intercomm )`**
    - remote group is server

Optional Name Service

- **`MPI_Publish_name( service_name, info, port_name )`**
- **`MPI_Lookup_name( service_name, info port_name )`**
- allow connection between **`service_name`** known to users and system-supplied **`port_name`**

Bootstrapping

- **`MPI_Join( fd, intercomm )`**
- collective over two processes connected by a socket.
- **`fd`** is a file descriptor for an open, quiescent socket.
- **`intercomm`** is a new intercommunicator.
- Can be used to build up full MPI communication.
- **`fd`** is *not* used for MPI communication.

One-Sided Operations: Issues

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

Remote Memory Access Windows

`MPI_Win_create( base, size, disp_unit, info, comm, win`

- Exposes memory given by **`(base, size)`** to RMA operations by other processes in **`com`** .
- **`wi`** is window object used in RMA operations.
- **`Disp_unit`** scales displacements:
  - 1 (no scaling) or **`sizeof(type)`**, where window is an array of elements of type **`type`**.
  - Allows use of array indices.
  - Allows heterogeneity.

One-Sided Communication Calls

- **`MPI_Put`** - stores into remote memory
- **`MPI_Get`** - reads from remote memory
- **`MPI_Accumulate`** - updates remote memory
- All are non-blocking: data transfer is initiated, but may continue after call returns.
- Subsequent synchronization on window is needed to ensure operations are complete.

Put, Get, and Accumulate

- **`MPI_Put( origin_addr, origin_count, origin_datatype, target_offset, target_count,target_datatype, window )`**
- **`MPI_Get(  ... )`**
- **`MPI_Accumulate( ..., op, ... )`**
- **`op`** is as in **`MPI_Reduce`**, but no user-defined operations are allowed.

Synchronization

Multiple methods for synchronizing on window:

- **`MPI_Win_fence`** - like barrier, supports BSP model
- **`MPI_Win_{start, complete, post, wait}`** - for closer control, involves groups of processes
- **`MPI_Win_{lock, unlock`** - provides shared-memory model.

Comments on MPI One-Sided

- Strengths
  - Scalable synchronization
  - Matches a wide variety of systems (including the highest-performance, non-coherent shared-memory)
  - Unordered operations provide opportunity for highest performance
- Weaknesses
  - Lack of an Read-Modify-Write operation makes data structure modification awkward
  - Unordered operations complicate programming

Extended Collective Operations

- In MPI-1, collective operations are restricted to ordinary (intra) communicators.
- In MPI-2, most collective operations apply also to intercommunicators, with appropriately different semantics.
- E.g, Bcast/Reduce in the intercommunicator resulting from spawning new processes goes from/to root in spawning processes to/from the spawned processes.
- In-place extensions

External Interfaces

- Purpose: to ease extending MPI by layering new functionality portably and efficiently
- Aids integrated tools (debuggers, performance analyzers)
- In general, provides portable access to parts of MPI implementation internals.
- Already being used in layering I/O part of MPI on multiple MPI implementations.

Components of MPI External Interface Specification

- Generalized requests
  - Users can create custom non-blocking operations with an interface similar to MPI's.
  - **`MPI_Waitall`** can wait on combination of built-in and user-defined operations.
- Naming objects
  - Set/Get name on communicators, datatypes, windows.
- Adding error classes and codes
- Datatype decoding
- Specification for thread-compliant MPI

C++ Bindings

- C++ binding alternatives:
  - use C bindings
  - Class library (e.g., OOMPI)
  - “minimal” binding
- Chose “minimal” approach
- Most MPI functions are member functions of MPI classes:
  - example: **`MPI::COMM_WORLD.send( ... )`**
- Others are in MPI namespace
- C++ bindings for both MPI-1 and MPI-2

Fortran Issues

- “Fortran” now means Fortran-90.
- MPI can’t take advantage of some new Fortran (-90) features, e.g., array sections.
- Some MPI features are incompatible with Fortran-90.
  - e.g., communication operations with different types for first argument, assumptions about argument copying.
- MPI-2 provides “basic” and “extended” Fortran support.

Fortran

- Basic support:
  - **`mpif.h`** must be valid in both fixed- and free-from format.
- Extended support:
  - mpi module
  - some new functions using parameterized types

Language Interoperability

- Single **`MPI_Ini`**
- Passing MPI objects between languages
- Constant values, error handlers
- Sending in one language; receiving in another
- Addresses
- Datatypes
- Reduce operation

Performance Issues in MPI-2

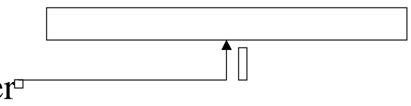
- Threads
  - Use of threads requires thread-safe code; may require expensive locks inside the implementation
  - MPI_INIT_THREAD options
- Dynamic Processes
  - Adding processes with MPI_Spawn may not provide peak performance
- One-sided
  - New ways to defer synchronization
  - Efficient implementation on non-shared memory hardware is challenging.
- Parallel I/O
  - Implementations exist...

MPI and Threads

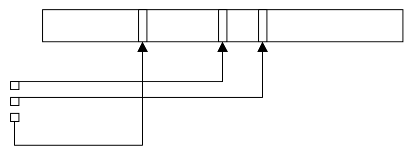
- Symmetric Multiprocessors (SMPs) are a common building block of many parallel machines
- The preferred programming model for SMPs with *threads*
  - POSIX ("pthreads")
  - OpenMP
  - sproc (SGI)
  - compiler-managed parallelism

What are Threads?

- Executing program (process) is defined by
  - Address space
  - Program Counter

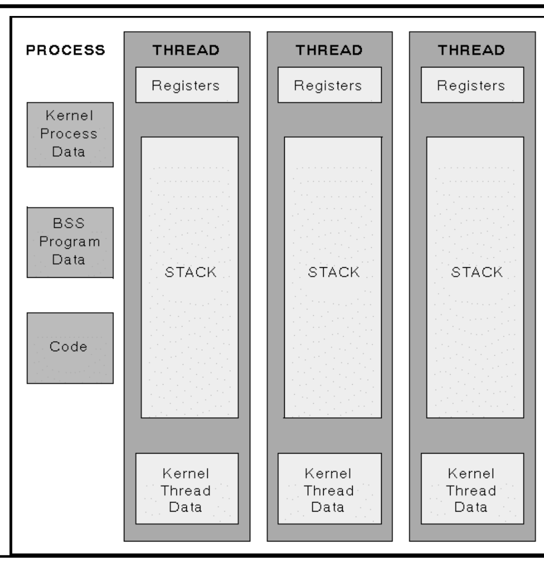


- Threads are multiple program counters



Inside a Thread

- http://www.spc.ibm.com/spcdocs/aixdocs/aix41gthr.html#threads



Kinds of Threads

- Almost a process
  - Kernel (Operating System) schedules
  - Each thread can make independent system calls
- Co-routines and lightweight processe
  - User schedules (sort of…)
- Memory references
  - Hardware schedules

Kernel Threads

- System calls (e.g., read, accept) block calling thread but not process
- Alternative to “nonblocking” or “asynchronous” I/O:
  - create_thread
    thread calls blocking read
- Can be expensive

User Threads

- System calls (may) block all threads in process
- Allows multiple processors to cooperate on data operations
  - loop: create # threads = # processors - 1
    each thread does part of loop
- Cheaper than kernel threads
  - Still must save registers (if in same processor)
  - Parallelism requires OS to schedule threads on different processors

Hardware Threads

- Hardware controls threads
- Allows single processor to interleave memory references and operations
  - Unsatisfied memory reference changes thread
  - Separate registers for each thread
- Single cycle thread switch with appropriate hardware
  - Basis of Tera MTA computer http://www.tera.com
  - Like kernel threads, replaces nonblocking hardware operations - multiple pending loads
  - Even lighter weight—just change PC

Why Use Threads?

- Manage multiple points of interaction
  - Low overhead steering/probing
  - Background checkpoint save
- Alternate method for nonblocking operations
  - CORBA method invocation (no funky nonblocking calls)
- Hiding memory latency
- Fine-grain parallelism
  - Compiler parallelis

Latency Hiding

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
  - Provides memory synchronization model: methods (procedures) declared "synchronized" executed by one thread at a time
  - (don't mention Ada, which has tasks)
- OpenMP (Fortran only for now; C in draft form)
  - Mostly directive-based parallel loops
  - Some thread features (lock/unlock)
  - http://www.openmp.org

Thread Issues

- Synchronization
  - Avoiding conflicting operations
- Variable Name Space
  - Interaction between threads and the language
- Scheduling
  - Will the OS do what you want?

Synchronization of Access

- Read/write model

```
a = 1;                 b = 1;
barrier();             barrier();
b = 2;                 while (a==1) ;
a = 2;                 printf( "%d\n", b );
```

  What does thread 2 print?
- Need lock/unlock to synchronize/orde
  - OpenMP has `FLUSH`, (overkill)
  - `volatile` in C
  - Fortran has no corresponding concept
- Java has “synchronized” methods (procedures)

Variable Names

- Each thread can access all of a processes memory (except for the thread's stack)
  - Named variables refer to the address space—thus visible to all threads
  - Compiler doesn't distinguish `A` in one thread from `A` in another
  - No modularity
  - Like using Fortran blank COMMON for all variables
- NEC has a variant where all variables names refer to different variables unless specified
  - All variables are on thread stack by default (even globals)
  - More modular
- Similar variations (e.g., Cray taskprivate)

Scheduling Threads

- If threads used for latency hiding
  - Schedule on the same processor
    - Provides better data locality, cache usage
  - Appropriate for data parallelism
- If threads used for parallel execution
  - Schedule on different processors using different memory pathways
  - Appropriate for task parallelism

The Changing Computing Model

- More interaction
  - Threads allow low-overhead agents on any compution
    - OS schedules if necessary; no overhead if nothing happens (almost…)
  - Changes the interaction model from batch (give commands, wait for results) to constant interaction
- Fine-grain parallelis
  - Simpler SMP programming model
- Lowering the Memory Wall
  - CPU speeds increasing much faster than memory
  - hardware threads hide memory latency

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
    - 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)

Threads and MPI

- `MPI_Init_thread(required,provided,ierr)`
  - Thread modes:
    - MPI_THREAD_SINGLE — One thread (MPI_Init)
    - MPI_THREAD_FUNNELED — One thread making MPI calls
    - 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)

Using Threads with MPI

- MPI defines what it means to support threads but does not require that support
  - Some vendors (such as IBM and Sun) support multi-threaded MPI processes
  - Others (such as SGI) do not
    - Interoperation with other thread systems (essentially MPI_THREAD_FUNNELED) may be supported
    - For example, SGI supports sproc (doacross) threading with MPI programs
- Active messages, interrupt receives, etc. are essentially MPI calls, such as a blocking receive, in a separate thread

Conclusion

- MPI is a proven, effective, portable parallel programming model
- 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