

More MPI

Karl W. Schulz

Texas Advanced Computing Center
The University of Texas at Austin

UT/Portugal Summer Institute Training
Coimbra, Portugal
July 16, 2008



Outline

- Quick Review
- Additional MPI
 - wildcards
 - more on send/receives
 - deadlock issues
 - bi-directional communications
 - programming considerations
 - persistent communications
- MPI I/O (if we have time)



Review

- Warm up: what does MPI stand for?
- Where are MPI intrinsics defined?
- What is this MPI_COMM_WORLD thing?
- What are some basic function calls every MPI code must make?
- What type of architectures/memory systems do MPI codes run on?



Review: Six Basic MPI Calls

- MPI_Init
 - Initialize MPI
- MPI_Comm_Rank
 - Get the processor rank/ID
- MPI_Comm_Size
 - Get the number of processors
- MPI_Send
 - Send data to another processor
- MPI_Recv
 - Get data from another processor
- MPI_Finalize
 - Finish MPI



Review: Anything Wrong Here (2 processor message passing)?

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    int myid, numprocs, tag, source, destination, count, buffer;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, myid);
    tag=1234; source=1; destination=2; count=1;
    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD);
        printf("processor %d sent %d\n", myid, buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        printf("processor %d got %d\n", myid, buffer);
    }
    MPI_Finalize();
}
```



Review: Anything Wrong Here (fixed)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    int myid, numprocs, tag, source, destination, count, buffer;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    tag=1234; source=0; destination=1; count=1;
    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD);
        printf("processor %d sent %d\n", myid, buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        printf("processor %d got %d\n", myid, buffer);
    }
    MPI_Finalize();
}
```



MPI Data Types

- MPI has many different predefined data types
 - Recall that a data type must be specified for all send/receive primitives
 - All your favorite C/Fortran data types are included
- MPI data types can be used in any data communication operation
- MPI handles endianness conversion (though a mixed architecture system is rare)
- Packed/opaque types can be made to handle C/F90 structs



MPI Predefined Data Types in C

C MPI Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-



MPI Predefined Data Types in F90

MPI_INTEGER	Integer
MPI_REAL	Real
MPI_DOUBLE_PRECISION	Double Precision
MPI_COMPLEX	Complex
MPI_LOGICAL	Logical
MPI_CHARACTER	Character
MPI_BYTE	Raw Byte (no conversion)
MPI_PACKED	MPI calls pack/unpack



Status

- The status parameter returns additional information for some MPI routines
 - additional error status information
 - additional information with wildcard parameters
- C declaration—a predefined struct
 - `MPI_Status status;`
- Fortran declaration—an integer array
 - `INTEGER STATUS(MPI_STATUS_SIZE)`



Accessing Status Information

- The tag of a received message
C : `status.MPI_TAG`
Fortran : `status(MPI_TAG)`
- The source of a received message
C : `status.MPI_SOURCE`
Fortran : `status(MPI_SOURCE)`
- The error code of the MPI call
C : `status.MPI_ERROR`
Fortran : `status(MPI_ERROR)`



MPI Error Checking

- Recall that most MPI calls (all except `MPI_Wtime` and `MPI_Wtick`) return an error code
- Additional functions exist to get more information
 - `MPI_Errhandler_set` – sets the error handler for a communicator
 - `MPI_Error_class` - converts an error code into an error class
 - `MPI_Error_string` - Return a string for a given error code



MPI Error Checking: Example

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

int main(int argc, char *argv[])
{
    int rank, nprocs, error, eclass, len;
    char estr[1024];

    MPI_Init(&argc,&argv);
    MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
    /* Is there something wrong here? */
    error = MPI_Bcast(NULL, 0, MPI_INT, -5, MPI_COMM_WORLD);
    MPI_Error_class(error, &eclass);
    MPI_Error_string(error, estr, &len);
    printf("Error %d: %s\n", eclass, estr);fflush(stdout);
    MPI_Finalize();
    return 0;
}
```



Error 7: Invalid root -5



Wildcards

- Enables programmer to avoid having to specify a tag and/or source.
- Example:
 - MPI_Status status;
 - int buffer[5];
 - int ierr;
 - ierr = MPI_Recv(&buffer[0], 5, MPI_INT,
 - MPI_ANY_SOURCE, MPI_ANY_TAG,
 - MPI_COMM_WORLD, &status);
- MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards
- status structure is used to get wildcard values



Wildcards

- `MPI_PROC_NULL`
 - can be used for send or receive
 - operation completes immediately
 - no communications involved
- Allows for the handling of edge cases in otherwise generic algorithms (very convenient)
- Particularly useful with `MPI_Sendrecv`



MPI_PROC_NULL

- Let's think about a simple finite-difference based jacobi solver
- The value at a point is replaced by the average of the North, South, East and West neighbours (a four point pencil is used and boundary values do not change).
- Note that A is expanded to account for BCs

```

REAL A(0:n+1,0:n+1), B(1:n,1:n)
...
!Main Loop
DO WHILE (.NOT.converged)
  ! perform 4 point stencil
  DO j=1, n
    DO i=1, n
      B(i,j)=0.25*(A(i-1,j)+A(i+1,j)
                  + A(i,j-1)+A(i,j+1))
    END DO
  END DO

  ! copy result back into array A
  DO j=1, n
    DO i=1, n
      A(i,j) = B(i,j)
    END DO
  END DO
...
! convergence test omitted
END DO

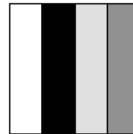
```



MPI_PROC_NULL

- Now, let's think about using MPI to parallelize in 1 dimension
- Step 1: decompose the matrix based on the number of available processors
- Dynamic memory used to store per process data

```
REAL, ALLOCATABLE A(:, :), B(:, :)  
...  
! Compute number of procs and myrank  
CALL MPI_COMM_SIZE(comm, p, ierr)  
CALL MPI_COMM_RANK(comm, myrank, ierr)  
  
! compute size of local block  
m = n/p  
IF (myrank.LT.(n-p*m)) THEN  
    m = m+1  
END IF
```



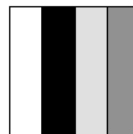
1D partition



MPI_PROC_NULL

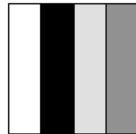
- Step 2: pre-compute neighbor exchange ranks for an East/West communication
- Step 3: allocate the arrays (and initialize from global values)

```
! Compute neighbors  
IF (myrank.EQ.0) THEN  
    left = MPI_PROC_NULL  
ELSE  
    left = myrank - 1  
END IF  
IF (myrank.EQ.p-1) THEN  
    right = MPI_PROC_NULL  
ELSE  
    right = myrank+1  
END IF  
  
! Allocate local arrays  
ALLOCATE (A(0:n+1,0:m+1), B(n,m))  
...
```



MPI_PROC_NULL

- Step 4: perform the usual Jacobi solver iteration
- Step 5: use sendrecv for data exchange



```
!Main Loop
DO WHILE(.NOT.converged)
  ! compute
  DO j=1, m
    DO i=1, n
      B(i,j)=0.25*(A(i-1,j)+A(i+1,j)
        + A(i,j-1)+A(i,j+1))
    END DO
  END DO

  DO j=1, m
    DO i=1, n
      A(i,j) = B(i,j)
    END DO
  END DO

  ...

```



MPI_PROC_NULL

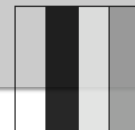
- Step 5: use sendrecv for data exchange
- Nothing special required at L/R boundaries since we are using MPI_PROC_NULL

```
! Communicate
CALL MPI_SENDRECV(B(1,1),n,
  MPI_REAL, left, tag, A(1,0),n,
  MPI_REAL, left, tag, comm,
  status, ierr)

CALL MPI_SENDRECV(B(1,m),n,
  MPI_REAL, right, tag, A(1,m+1),n,
  MPI_REAL, right, tag, comm,
  status, ierr)

END IF
...

```



More Info: <http://www.netlib.org/utk/papers/mpi-book/node46.html>



MPI_Probe

- `MPI_Probe` allows incoming messages to be checked without actually receiving them
 - the user can then decide how to receive the data
 - useful when different action needs to be taken depending on the "who, what, and how much" information of the message
 - this is a **blocking** test
 - handy in conjunction with `MPI_Get_count()`



MPI_Probe

- C

```
ierr=MPI_Probe(source, tag, comm, &status)
```
- Fortran

```
MPI_Probe(SOURCE, TAG, COMM, STATUS, IERROR)
```
- Parameters
 - Source: source rank or `MPI_ANY_SOURCE`
 - Tag: tag value or `MPI_ANY_TAG`
 - Comm: communicator
 - Status: status object



MPI_Probe Example

```
#include <stdio.h>
#include <mpi.h>
/* Program shows how to use probe and get_count to find the
   size */
/* of an incoming message */
#define MCW MPI_COMM_WORLD
int main(int argc, char *argv[])
{
    int myid, numprocs;
    MPI_Status status;
    int i, mytag, ierr, icount, src;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    /* print out my rank and this run's PE size */
    printf("Hello from %d\n", myid);
    printf("Numprocs is %d\n", numprocs);
}
```



MPI_Probe example (cont.)

```
mytag=123;
i=0;
icount=0;
if(myid == 0) {
    i=100;
    icount=1;
    ierr=MPI_Send(&i, icount, MPI_INT, 1, mytag, MCW);
}
if(myid == 1){
    ierr=MPI_Probe(MPI_ANY_SOURCE, mytag, MCW, &status);
    src=status.MPI_SOURCE;
    ierr=MPI_Get_count(&status, MPI_INT, &icount);
    printf("getting %d\n", icount);
    ierr = MPI_Recv(&i, icount, MPI_INT, src, mytag, MCW, &status);
    printf("i= %d\n", i);
}
MPI_Finalize();
}
```



Non-blocking Communication

- Non-blocking send
 - send call returns immediately
 - send actually occurs later
- Non-blocking receive
 - receive call returns immediately
 - when received data is needed, call a wait subroutine
- Non-blocking communication used to overlap communication with computation (and communication with communication!).
- Can also help prevent deadlock



Non-blocking Send with MPI_Isend

- C


```
MPI_Request request;
ierr = MPI_Isend(&buffer, count, datatype,
                dest, tag, comm, &request);
```
- Fortran


```
integer REQUEST
Call MPI_Isend(buffer, count, datatype,
                dest, tag, comm, request, ierr)
```
- `request` is a new output parameter
- Not safe to change data in `buffer` until communication is complete



Non-blocking Receive with MPI_Irecv

- C

```
MPI_Request request;
ierr = MPI_Irecv(&buffer, count, datatype,
                 source, tag, comm, &request);
```

- Fortran

```
integer request
call MPI_Irecv(buffer, count, datatype,
               source, tag, comm, request, ierr)
```

- Parameter changes

- new: request, communication request
- status parameter is missing

- Don't use data in buffer until communication is complete



MPI_Wait Used to Complete Communication

- request from MPI_Isend or MPI_Irecv
 - the completion of a send operation indicates that the sender is now free to update the data in the send buffer
 - the completion of a receive operation indicates that the receive buffer contains the received message
- MPI_Wait blocks until message specified by request completes



MPI_Wait Usage

- C


```
MPI_Request request;
MPI_Status status;
ierr = MPI_Wait(&request, &status)
```
- Fortran


```
integer request
integer status(MPI_STATUS_SIZE)
call MPI_Wait(request, status, ierr)
```



MPI_Test

- Similar to MPI_Wait, but does not block
- Value of flags signifies whether a message has been delivered
- C


```
int flag;
ierr= MPI_Test(&request,&flag, &status);
```
- Fortran


```
LOGICAL FLAG
CALL MPI_Test(request, flag, status, ierr)
```



Non-Blocking Send Example

```
        call MPI_Isend(buffer,count,datatype,dest,  
                      tag,comm,request,ierr)  
10 continue  
  
    Do some other clever work ...  
  
    call MPI_Test (request, flag, status, ierr)  
    if (.not. flag) goto 10
```



Non-blocking Send and Receive

- Program to send and receive data using MPI_Isend and MPI_Irecv
 - Initialize MPI
 - Have processor 0 send an integer to processor 1
 - Have processor 1 receive and integer from processor 0
 - Both processors check on message completion
 - Quit MPI



Non-blocking example

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

int main(int argc, char *argv[])
{
    int myid, numprocs;
    int tag, source, destination, count;
    int buffer;
    MPI_Status status;
    MPI_Request request;
```



Non-blocking example

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
tag=1234;
source=0;
destination=1;
count=1;
request=MPI_REQUEST_NULL;
```



```

if(myid == source){
    buffer=5678;
    MPI_Isend(&buffer,count,MPI_INT,destination,tag,
             MPI_COMM_WORLD,&request);
}
if(myid == destination){
    MPI_Irecv(&buffer,count,MPI_INT,source,tag,
             MPI_COMM_WORLD,&request);
}
MPI_Wait(&request,&status);
if(myid == source){
    printf("processor %d sent %d\n",myid,buffer);
}
if(myid == destination){
    printf("processor %d got %d\n",myid,buffer);
}
MPI_Finalize();
}

```



MPI Point-to-point Send Modes

- Three types of send modes
 - buffered
 - synchronous
 - ready
 - ... plus the standard mode (more on this later)
- Three extra send functions (well, six)
 - MPI_Bsend, MPI_Ibsend
 - MPI_Ssend, MPI_Issend
 - MPI_Rsend, MPI_Irsend



Buffered Mode

- Messages are required to be copied to an internal or user-supplied buffer
 - when the copy is done, the send is complete
 - `MPI_Bsend` returns
 - `MPI_Test` will be true for an `MPI_Ibsend`
 - matching receive need not be posted to call `MPI_Bsend`
 - an error occurs if there is insufficient buffer space
- Buffer space controlled with
 - `MPI_Buffer_attach`
 - `MPI_Buffer_detach`



Synchronous Mode

- Sends are complete after the matching receive is posted
 - receive need not be posted to call `MPI_Ssend`
 - but the send can't complete until it has been
- Some internal buffering may be used
 - but it is not required by the standard for the implementation to do so
- Blocking calls imply that sender and receiver rendezvous at the message



Ready Mode

- May only be called if the sender **knows** that the matching receive has been posted
 - otherwise it is an error, and the behavior is undefined
- Send complete means that the buffer is safe to use
 - ... and nothing else!
- Can be used to save some overhead on some systems
 - requires very careful programming



Standard Mode

- Implementation decides whether to use buffered or synchronous mode
 - usually small messages are buffered
 - and large messages are synchronous
 - the *eager* limit dictates the switch threshold
- Choice may also be based on the availability of the buffer space
- `MPI_Send/MPI_Isend` use standard mode



Order Semantics

- Messages with the same tag are ordered
 - the first receive always matches the first send in the following

```

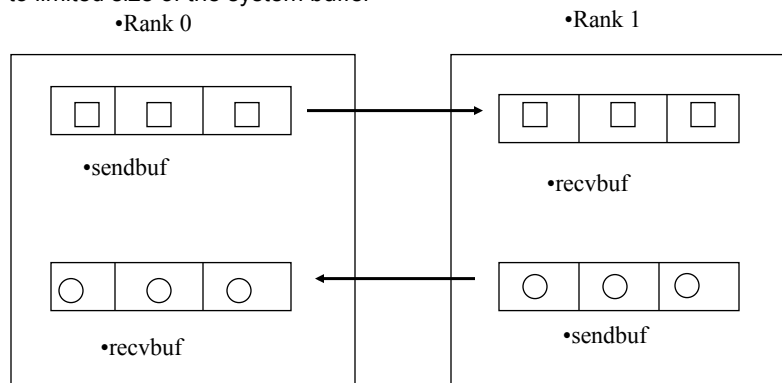
tag=123456
IF (rank.EQ.0) THEN
  CALL MPI_BSEND(b1,cnt,MPI_REAL,1,tag,comm,err)
  CALL MPI_BSEND(b2,cnt,MPI_REAL,1,tag,comm,err)
ELSE ! rank.EQ.1
  CALL MPI_RECV(b1,cnt,MPI_REAL,0,tag,comm,
               status,ierr)
  CALL MPI_RECV(b2,cnt,MPI_REAL,0,tag,comm,
               status,ierr)
END IF

```



Avoiding Deadlock

- Must be careful to avoid deadlock when two processes exchange data with each other
- Deadlock can occur due to incorrect order of send and receive, or due to limited size of the system buffer



Bidirectional Communication

- Case 1 : processors send first, then post receive

```

if (myrank == 0 ) then
    call MPI_Send(...)
    call MPI_Recv (...)
elseif (myrank == 1) then
    call MPI_Send(...)
    call MPI_Recv(...)
endif

```

- No deadlock as long as system buffer is larger than send buffer
- Deadlock if system buffer is smaller than send buf
- If you replace MPI_Send with MPI_Isend followed immediately by MPI_Wait, it is still the same
- Moral #1: there may be error in coding that only shows up for larger problem size (*very sneaky indeed*)
- Moral #2: **don't do this!**



Bidirectional Communication

- The following is free from deadlock

```

if (myrank == 0 ) then
    call MPI_Isend(...)
    call MPI_Recv (...)
    call MPI_Wait(...)
elseif (myrank == 1) then
    call MPI_Isend(...)
    call MPI_Recv (...)
    call MPI_Wait(...)
endif

```



Bidirectional Communication

- Case 2: both processes call recv first, then send


```

if (myrank == 0 ) then
    call MPI_Recv(...)
    call MPI_Send (...)
elseif (myrank == 1) then
    call MPI_Recv(...)
    call MPI_Send(...)
endif
      
```
- The above will always lead to deadlock (even if you replace MPI_Send with MPI_Isend and MPI_Wait)



Bidirectional Communication

- The following code can be safely executed

```

if (myrank == 0 ) then
    call MPI_Irecv(...)
    call MPI_Send (...)
    call MPI_Wait(...)
elseif (myrank == 1) then
    call MPI_Irecv(...)
    call MPI_Send(...)
    call MPI_Wait(...)
endif
      
```



Bidirectional Communication

- The following code can be safely executed

```

if (myrank == 0 ) then
    call MPI_Send (...)
    call MPI_Recv(...)
elseif (myrank == 1) then
    call MPI_Recv(...)
    call MPI_Send(...)
endif

```

- ...but it's specific to the 2 processor case!
- Moral #3: you probably don't want to do this either!



Bidirectional Communication with MPI_Sendrecv

- MPI_Sendrecv
 - initiates send and receive at the same time
 - blocking, standard mode
 - completes when both send and receive buffers are safe to use
 - good for avoiding deadlock, implementing shifts/rings
- C


```

int MPI_Sendrecv( void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int source,
int recvtag, MPI_Comm comm, MPI_Status *status )

ierr=MPI_Sendrecv(&sb[0],scnt,stype,dest,stag,
&rb[0],rcnt,rtype,src,rtag,comm,&status)

```
- Fortran


```

call mpi_sendrecv(sb,scnt,stype,dest,stag,
rb,rcnt,rtype,src,rtag,comm,status,ierr)

```



Bidirectional Communication

- The following code can be safely executed

```
if (myrank == 0) then
  call MPI_Sendrecv(...)
elseif (myrank == 1) then
  call MPI_Sendrecv(...)
endif
```



Programming Considerations

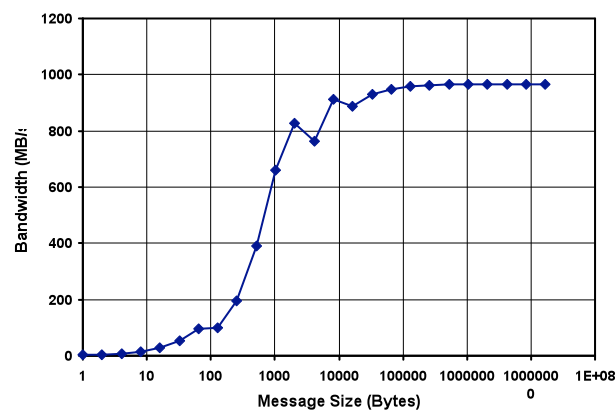


Message Size

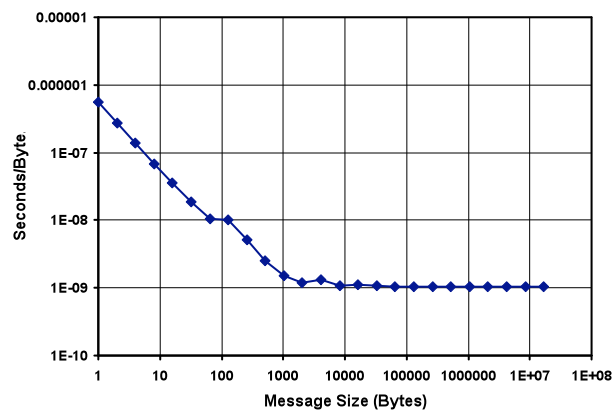
- Small messages are expensive
 - Every communication has a fixed startup overhead
 - Need to amortize this overhead over the length of the message
- Try to aggregate communications into the longest possible messages (perhaps by using derived datatypes if necessary)



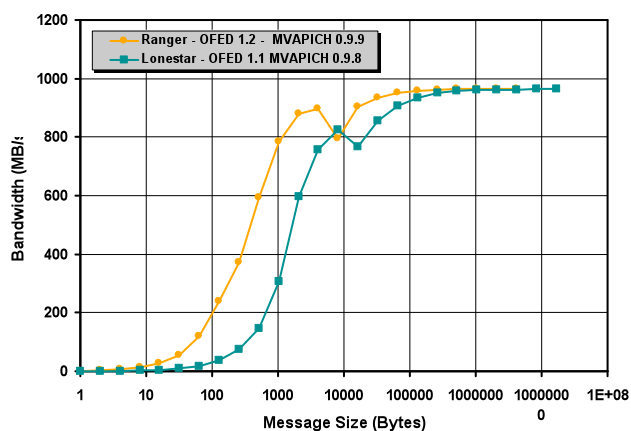
P2P Message Size



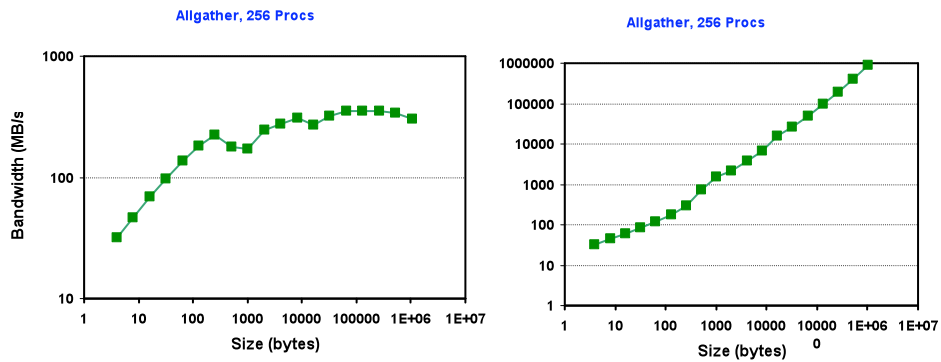
P2P Message Size



P2P Message Size



Collectives Performance (InfiniBand)



Note the absolute performance:
 1KB Allgather = .001 secs
 1MB Allgather = 1.0 sec



Blocking or Non-blocking

- Typical Code Flow

```
while(!done) {
    ...
    compute_some_stuff();
    communicate_results();
    update_local_stuff();
    ...
}
```



Blocking or Non-blocking

- One Optimization

```
while(!done) {
    ...
    compute_part_of_stuff();
    start_communicating_results();
    compute_rest_of_stuff();
    complete_communicating();
    update_local_stuff();
    ...
}
```



Blocking or Non-blocking

- Default to using non-blocking mode
 - Post receives
 - Post sends
 - (Maybe) do some work
 - complete sends & receives
- Get sophisticated when
 - buffer space requirements are too high
 - communication hardware or OS doesn't really support overlap of communication and computation
- Revert to blocking
 - for debugging and regression testing (ie. make sure you get the same answer with blocking enabled)
 - when all else fails
- Recommend abstracting your communication exchanges into specific routines (allow for non-blocking/blocking modes)
- Consider attaching additional buffer space if not using all available memory per core/task



Blocking or Non-blocking

- Non-blocking calls + overlap hide message latency
- Latencies in the Wild
 - 1.7—2.8 μs on Ranger
 - number of switch chips traversed
 - cable length (~ 5 ns/m in copper, ~ 3 ns/m in fiber)
 - Lonestar min 3.6 μs
 - Cray XT4 ~ 6 μs
 - GigE ~ 50 -100 μs



Persistent Communication

- MPI persistent communications can be used to reduce communications overhead in programs which repeatedly call the same point-to-point message passing routines with the same arguments
- Minimizes the overhead associated with redundant message setup
- An example of an application which might benefit from persistent communications would be an iterative, data decomposition algorithm that exchanges border elements with its neighbors.
- The message size, location, tag, communicator and data type remain the same each iteration.
- Persistent communication routines are non-blocking



Persistent Communication I

- Saves arguments of a communication call and reduces the overhead from subsequent calls
- The INIT call takes the original argument list of a send or receive call and creates a corresponding communication request (e.g., MPI_SEND_INIT, MPI_RECV_INIT)
- The START call uses the communication request to start the corresponding operation (e.g. MPI_START, MPI_STARTALL)
- The REQUEST_FREE call frees the persistent communication request(MPI_REQUEST_FREE)



Persistent Communication II

- A typical situation where *persistence* might be used.

```

:
MPI_Recv_init(buf1, count,type,src,tag,comm,&req[0]);
MPI_Send_init(buf2, count,type,src,tag,comm,&req[1]);

for (i=1; i < BIGNUM; i++)
{
    MPI_Start(&req[0]);
    MPI_Start(&req[1]);
    MPI_Waitall(2,req,status);
    do_work(buf1, buf2);
}

MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
:

```



Persistent Communication III

- Performance benefits (IBM SP2) from using *Persistence*

Improvement in Wallclock Time

Persistent vs. Conventional Communication

msize (bytes)	mode	improvement	mode	improvement
8	async	19 %	sync	15 %
4096	async	11 %	sync	4.7 %
8192	async	5.9 %	sync	2.9 %
800,000	-	-	sync	0 %
8,000,000	-	-	sync	0 %

