Tutorial on MPI: The Message-Passing Interface

William Gropp



Mathematics and Computer Science Division Argonne National Laboratory Argonne, IL 60439 gropp@mcs.anl.gov

1

Course Outline

- Background on Parallel Computing
- Getting Started
- MPI Basics
- Intermediate MPI
- Tools for writing libraries
- Final comments

Thanks to Rusty Lusk for some of the material in this tutorial.

This tutorial may be used in conjunction with the book "Using MPI" which contains detailed descriptions of the use of the MPI routines.

P Material that beings with this symbol is 'advanced' and may be skipped on a first reading.



- Parallel Computing
- Communicating with other processes
- Cooperative operations
- One-sided operations
- The MPI process





All use different data for each worker

Data-parallel Same operations on different data. Also called SIMD

SPMD Same program, different data

MIMD Different programs, different data

SPMD and MIMD are essentially the same because any MIMD can be made SPMD

SIMD is also equivalent, but in a less practical sense.

MPI is primarily for SPMD/MIMD. HPF is an example of a SIMD interface.



Message-passing is an the exchange of data c	approach that makes cooperative.
Data must both be ex received.	plicitly sent and
An advantage is that a <i>receiver's</i> memory is m	any change in the ade with the receiver's
participation. Process 0	Process 1
SEND(data)	
	RECV(data)
	1



One-sided operations between parallel processes include remote memory reads and writes.

An advantage is that data can be accessed without waiting for another process



Tak num	e a pad of paper. Algorithm: Initialize with the ober of neighbors you have
•	Compute average of your neighbor's values and subtract from your value. Make that your new value.
٠	Repeat until done
Que	estions
1.	How do you get values from your neighbors?
2.	Which step or iteration do they correspond to? Do you know? Do you care?
3.	How do you decide when you are done?

The previous example illustrates the hardware models by how data is exchanged among workers.

- Distributed memory (e.g., Paragon, IBM SPx, workstation network)
- Shared memory (e.g., SGI Power Challenge, Cray T3D)

Either may be used with SIMD or MIMD software models.

 $\stackrel{\scriptstyle{\scriptstyle (2)}}{\geq}$ All memory is distributed.

What is MPI?

- A message-passing library specification
 - message-passing model
 - not a compiler specification
 - not a specific product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to permit (unleash?) the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
 - end users
 - library writers
 - tool developers



Fow systems offer the full range of desired feature	
 modularity (for libraries) 	:5.
 access to peak performance 	
• portability	
 heterogeneity 	
• subgroups	
 topologies 	
 performance measurement tools 	

The MPI Process

- Began at Williamsburg Workshop in April, 1992
- Organized at Supercomputing '92 (November)
- Followed HPF format and process
- Met every six weeks for two days
- Extensive, open email discussions
- Drafts, readings, votes
- Pre-final draft distributed at Supercomputing '93
- Two-month public comment period
- Final version of draft in May, 1994
- Widely available now on the Web, ftp sites, netlib (http://www.mcs.anl.gov/mpi/index.html)
- Public implementations available
- Vendor implementations coming soon



- Broad participation
- Vendors
 - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
- Library writers
 - PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
- Application specialists and consultants

Companies	Laboratories	Universities
Companies ARCO Convex Cray Res IBM Intel KAI Meiko NAG nCUBE ParaSoft Shell TMC	Laboratories ANL GMD LANL LLNL NOAA NSF ORNL PNL Sandia SDSC SRC	Universities UC Santa Barbara Syracuse U Michigan State U Oregon Grad Inst U of New Mexico Miss. State U. U of Southampton U of Colorado Yale U U of Tennessee U of Maryland Western Mich U U of Edinburgh Cornell U.
		Rice U. U of San Francisco



Features of MPI (cont.)
 Application-oriented process topologies
 Built-in support for grids and graphs (uses groups)
 Profiling
 Hooks allow users to intercept MPI calls to install their own tools
 Environmental
- inquiry
– error control

Features not in MPI			
	Non-message-passing concepts not included:		
	 process management 		
	 remote memory transfers 		
	 active messages 		
	– threads		
	 virtual shared memory 		
Ð	MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, intercommunicators)		

 MPI is large (125 functions) MPI's extensive functionality requires many functions Number of functions not necessarily a measu of complexity MPI is small (6 functions) Many parallel programs can be written with j 6 basic functions. 		
 MPI's extensive functionality requires many functions Number of functions not necessarily a measu of complexity MPI is small (6 functions) Many parallel programs can be written with j 6 basic functions. 	MPI	is large (125 functions)
 Number of functions not necessarily a measu of complexity MPI is small (6 functions) Many parallel programs can be written with j 6 basic functions. 	– N f	IPI's extensive functionality requires many unctions
 MPI is small (6 functions) Many parallel programs can be written with j 6 basic functions. 	- N C	lumber of functions not necessarily a measure of complexity
 Many parallel programs can be written with j 6 basic functions. 	MPI	is small (6 functions)
	— N 6	Aany parallel programs can be written with just basic functions.
MPI is just right	MPI	is just right
- One can access flexibility when it is required.	- (One can access flexibility when it is required.
- One need not master all parts of MPI to use	- (One need not master all parts of MPI to use it.

	You need a portable parallel program
•	You are writing a parallel library
•	You have irregular or dynamic data relationships that do not fit a data parallel model
Wł	nere <i>not</i> to use MPI:
•	You can use HPF or a parallel Fortran 90
•	You don't need parallelism at all
•	You can use libraries (which may be written in MPI)

Whv	learn	MPI?

- Portable
- Expressive
- Good way to learn about subtle issues in parallel computing



Writing MPI programs

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
int argc;
char **argv;
{
MPI_Init( &argc, &argv );
printf( "Hello world\n" );
MPI_Finalize();
return 0;
}
```



- #include "mpi.h" provides basic MPI definitions and types
- MPI_Init starts MPI
- MPI_Finalize exits MPI
- Note that all non-MPI routines are local; thus the printf run on each process

_				.,	
⊢or s	imple prog	rams, s	pecial c	ompiler	
comr	nands can	be used	. For la	rge projec	cts,
it is l	est to use	a stand	lard Mal	<efile.< td=""><td></td></efile.<>	
The	MPICH im	plement	ation pr	ovides	
the c	ommands	mpicc a	nd mpif	77	
		$fila' \Delta $	amplos	in	

Special compilation commands

The commands

mpicc -o first first.c
mpif77 -o firstf firstf.f

may be used to build simple programs when using MPICH.

These provide special options that exploit the profiling features of MPI

-mpilog Generate log files of MPI calls

-mpitrace Trace execution of MPI calls

-mpianim Real-time animation of MPI (not available on all systems)

There are specific to the MPICH implementation; other implementations may provide similar commands (e.g., mpcc and mpxlf on IBM SP2). The file 'Makefile.in' is a *template* Makefile. The program (script) 'mpireconfig' translates this to a Makefile for a particular system. This allows you to use the same Makefile for a network of workstations and a massively parallel computer, even when they use different compilers, libraries, and linker options.

mpireconfig Makefile

Note that you must have 'mpireconfig' in your PATH.

Sample Makefile.in

User configurable options ##### ARCH = @ARCH@ COMM = @COMM@INSTALL_DIR = @INSTALL_DIR@ CC = @CC@ F77 = @F77@ CLINKER = @CLINKER@ FLINKER = @FLINKER@ OPTFLAGS = @OPTFLAGS@ # LIB_PATH = -L\$(INSTALL_DIR)/lib/\$(ARCH)/\$(COMM) FLIB_PATH = @FLIB_PATH_LEADER@\$(INSTALL_DIR)/lib/\$(ARCH)/\$(COMM) LIB_LIST = @LIB_LIST@ # INCLUDE_DIR = @INCLUDE_PATH@ -I\$(INSTALL_DIR)/include ### End User configurable options

Sample Makefile.in (con't)

```
CFLAGS = @CFLAGS@ $(OPTFLAGS) $(INCLUDE_DIR) -DMPI_$(ARCH)
FFLAGS = @FFLAGS@ $(INCLUDE_DIR) $(OPTFLAGS)
LIBS = $(LIB_PATH) $(LIB_LIST)
FLIBS = $(FLIB_PATH) $(LIB_LIST)
EXECS = hello
default: hello
all: $(EXECS)
hello: hello.o $(INSTALL_DIR)/include/mpi.h
        $(CLINKER) $(OPTFLAGS) -o hello hello.o \
        $(LIB_PATH) $(LIB_LIST) -lm
clean:
        /bin/rm -f *.o *~ PI* $(EXECS)
.c.o:
        $(CC) $(CFLAGS) -c $*.c
.f.o:
        $(F77) $(FFLAGS) -c $*.f
```

Running MPI programs

```
mpirun -np 2 hello
```

'mpirun' is not part of the standard, but some version of it is common with several MPI implementations. The version shown here is for the MPICH implementation of MPI.

Solution Just as Fortran does not specify how Fortran programs are started, MPI does not specify how MPI programs are started.

The option -t shows the commands that mpirun would execute; you can use this to find out how mpirun starts programs on yor system. The option -help shows all options to mpirun.

T pi ai	wo of the first questions asked in a parallel rogram are: How many processes are there? nd Who am I?
H ai	ow many is answered with MPI_Comm_size nd who am I is answered with MPI_Comm_rank.
T si	he rank is a number between zero and ize-1.

A simple program

Caveats

These sample programs have been kept as simple as possible by assuming that all processes can do output. Not all parallel systems provide this feature, and MPI provides a way to handle this case.





Questions:

- To whom is data sent?
- What is sent?
- How does the receiver identify it?


The Buffer

Sending and receiving only a contiguous array of bytes:

- hides the real data structure from hardware which might be able to handle it directly
- requires pre-packing dispersed data
 - rows of a matrix stored columnwise
 - general collections of structures
- prevents communications between machines with different representations (even lengths) for same data type



	eneralizing the Type
)	A single type field is too constraining. Often overloaded to provide needed flexibility.
	Problems:
	– under user control
	 wild cards allowed (MPI_ANY_TAG)
	 library use conflicts with user and with other libraries







Correct Execution of Library Calls with Pending Communcication



Incorrect Execution of Library Calls with Pending Communication



44

A separ of mess (This h overload	ate communication <i>context</i> for each family sages, used for queueing and matching. as often been simulated in the past by ding the tag field.)
No wild	cards allowed, for security
Allocate	ed by the system, for security
Types (cards C	<i>tags</i> , in MPI) retained for normal use (wild K)





- Collective operations typically operated on all processes (although some systems provide subgroups).
- This is too restrictive (e.g., need minimum over a column or a sum across a row, of processes)
- MPI provides groups of processes
 - initial "all" group
 - group management routines (build, delete groups)
- All communication (not just collective operations) takes place in groups.
- A group and a context are combined in a *communicator*.
- Source/destination in send/receive operations refer to *rank* in group associated with a given communicator. MPI_ANY_SOURCE permitted in a receive.



Thus the basic (blocking) send has become:

and the receive:

The source, tag, and count of the message actually received can be retrieved from status.

Two simple collective operations:



Simple Fortran example

С

```
program main
     include 'mpif.h'
     integer rank, size, to, from, tag, count, i, ierr
     integer src, dest
     integer st_source, st_tag, st_count
     integer status(MPI_STATUS_SIZE)
     double precision data(100)
     call MPI_INIT( ierr )
     call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
     call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
    print *, 'Process ', rank, ' of ', size, ' is alive'
     dest = size - 1
     src = 0
     if (rank .eq. src) then
               = dest
        to
        count
               = 10
               = 2001
        tag
        do 10 i=1, 10
10
           data(i) = i
        call MPI_SEND( data, count, MPI_DOUBLE_PRECISION, to,
                        tag, MPI_COMM_WORLD, ierr )
    +
     else if (rank .eq. dest) then
             = MPI_ANY_TAG
        tag
        count = 10
        from = MPI_ANY_SOURCE
        call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
                       tag, MPI_COMM_WORLD, status, ierr )
    +
```



Six	Function	MPI

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

MPI_Init

MPI_Finalize

MPI_Comm_size

MPI_Comm_rank

MPI_Send

MPI_Recv

A taste of things to come

The following examples show a C and Fortran version of the same program.

This program computes PI (with a very simple method) but does not use MPI_Send and MPI_Recv. Instead, it uses *collective* operations to send data to and from all of the running processes. This gives a *different* six-function MPI set:

MPI_Init

MPI_Finalize

MPI_Comm_size

MPI_Comm_rank

MPI_Bcast

MPI_Reduce

Broadcast	and	Reduction
-----------	-----	-----------

The routine MPI_Bcast sends data from one process to all others.

The routine MPI_Reduce combines data from all processes (by adding them in this case), and returning the result to a single process.

Fortran example: PI

```
program main
     include "mpif.h"
      double precision PI25DT
                       (PI25DT = 3.141592653589793238462643d0)
      parameter
     double precision mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, rc
                                  function to integrate
с
     f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
     if (myid .eq. 0) then
 10
         write(6,98)
         format('Enter the number of intervals: (0 quits)')
 98
         read(5,99) n
 99
         format(i10)
      endif
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

Fortran example (cont.)

```
check for quit signal
С
     if ( n .le. 0 ) goto 30
                                 calculate the interval size
с
    h = 1.0 d0/n
     sum = 0.0d0
     do 20 i = myid+1, n, numprocs
        x = h * (dble(i) - 0.5d0)
        sum = sum + f(x)
 20 continue
    mypi = h * sum
                                  collect all the partial sums
С
    call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
    $
          MPI_COMM_WORLD,ierr)
С
                                 node 0 prints the answer.
     if (myid .eq. 0) then
        write(6, 97) pi, abs(pi - PI25DT)
        format(' pi is approximately: ', F18.16,
97
               ' Error is: ', F18.16)
    +
     endif
    goto 10
 30 call MPI_FINALIZE(rc)
     stop
     end
```

C example: PI

```
#include "mpi.h"
#include <math.h>
int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
C example (cont.)
```

```
while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
        = 1.0 / (double) n;
    h
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {</pre>
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```



Objective: Experiment with send/receive

Run either program for PI. Write new versions that replace the calls to MPI_Bcast and MPI_Reduce with MPI_Send and MPI_Recv.

The MPI broadcast and reduce operations use at most $\log p$ send and receive operations on each process where p is the size of MPI_COMM_WORLD. How many operations do your versions use? Objective: Experiment with send/receive Write a program to send a message around a ring of processors. That is, processor 0 sends to processor 1, who sends to processor 2, etc. The last processor returns the message to processor 0.

You can use the routine MPI_Wtime to time code in MPI. The statement

t = MPI_Wtime();

returns the time as a double (DOUBLE PRECISION in Fortran).

Topologies

MPI provides routines to provide structure to collections of processes

This helps to answer the question:

Who are my neighbors?



The deco dim	e routine : ompositic ensions g	MPI_Cart_cr on of the pro iven by the	eate creat ocesses, w ndim argu	es a Cartesi ⁄ith the num ment.	an Iber of
c c F F T T	lims(1) lims(2) periods(1) periods(2) reorder ndim	= 4 = 3) = .false.) = .false. = .true. = 2			
ب	nuim Sall MPT (CART CREATE	(MPI COMM	WORLD, ndir	n. dims.

Finding neighbors

MPI_Cart_create creates a new *communicator* with the same processes as the input communicator, but with the specified topology.

The question, Who are my neighbors, can now be answered with MPI_Cart_shift:

The values returned are the ranks, in the communicator comm2d, of the neighbors shifted by ± 1 in the two dimensions.



Can be answered with

Returns the Cartesian coordinates of the calling process in coords.

Partitioning

When creating a Cartesian topology, one question is "What is a good choice for the decomposition of the processors?"

This question can be answered with MPI_Dims_create:

```
integer dims(2)
dims(1) = 0
dims(2) = 0
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
call MPI_DIMS_CREATE( size, 2, dims, ierr )
```

MPI contaiı	is routines to trai	nslate between
Cartesian c	oordinates and ra	nks in a
communica [.]	or, and to access	s the properties
of a Cartesi	an topology.	
The routine	MPI_Graph_create	e allows the
creation of	a general graph t	opology.

Why are these routines in MPI?

In many parallel computer interconnects, some processors are closer to than others. These routines allow the MPI implementation to provide an ordering of processes in a topology that makes logical neighbors close in the physical interconnect.

Some parallel programmers may remember hypercubes and the effort that went into assigning nodes in a mesh to processors in a hypercube through the use of Grey codes. Many new systems have different interconnects; ones with multiple paths may have notions of near neighbors that changes with time. These routines free the programmer from many of these considerations. The reorder argument is used to request the best ordering.





Nonpei	riodic	Grids
--------	--------	-------

In the nonperiodic case, a neighbor may not exist. This is indicated by a rank of MPI_PROC_NULL.

This rank *may be used* in send and receive calls in MPI. The action in both cases is as if the call was not made.


Synchronization

- MPI_Barrier(comm)
- Function blocks untill all processes in comm call it.



Schematic representation of collective data movement in MPI



Schematic representation of collective data movement in MPI

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 chunks to have different sizes.
- Allreduce, Reduce, ReduceScatter, and Scan take both built-in and user-defined combination functions.

Built-in Collective Computation Operations

MPI Name	Operation
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 (xor)
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Defining Your Own Collective Operations

```
MPI_Op_create(user_function, commute, op)
MPI_Op_free(op)
```

user_function(invec, inoutvec, len, datatype)

The user function should perform

inoutvec[i] = invec[i] op inoutvec[i];

for i from 0 to len-1.

user_function can be non-commutative (e.g., matrix multiply).

Sample user function

For example, to create an operation that has the same effect as MPI_SUM on Fortran double precision values, use

```
subroutine myfunc( invec, inoutvec, len, datatype )
    integer len, datatype
    double precision invec(len), inoutvec(len)
    integer i
    do 10 i=1,len
10 inoutvec(i) = invec(i) + inoutvec(i)
```

```
10 inoutvec(1) = invec(1) + inoutvec(1)
   return
   end
```

```
To use, just
```

```
integer myop
call MPI_Op_create( myfunc, .true., myop, ierr )
call MPI_Reduce( a, b, 1, MPI_DOUBLE_PRECISON, myop,
```

```
The routine MPI_Op_free destroys user-functions when they are no longer needed.
```

Defining	groups
----------	--------

All MPI communication is relative to a *communicator* which contains a *context* and a *group*. The group is just a set of processes.





Similarly, to form groups of columns,



use

```
MPI_Comm_split( oldcomm, column, 0, &newcomm2 );
```

To maintain the order by rank, use

```
MPI_Comm_rank( oldcomm, &rank );
MPI_Comm_split( oldcomm, column, rank, &newcomm2 );
```

Manipulating	Groups
--------------	--------

Another way to create a communicator with specific members is to use MPI_Comm_create.

MPI_Comm_create(oldcomm, group, &newcomm);

The group can be created in many ways:

Creating Groups

All group creation routines create a group by specifying the members to take from an existing group.

- MPI_Group_incl specifies specific members
- MPI_Group_excl excludes specific members
- MPI_Group_range_incl and MPI_Group_range_excl use ranges of members
- MPI_Group_union and MPI_Group_intersection creates a new group from two existing groups.

To get an existing group, use

MPI_Comm_group(oldcomm, &group);

Free a group with

MPI_Group_free(&group);





But this requires that either that MPI_Send not return until the data has been delivered *or* that we allow a send operation to return before completing the transfer. In this case, we need to test for completion later.



Completion depends in general on size of message and amount of system buffering.

Send works for small enough messages but fails when messages get too large. Too large ranges from zero bytes to 100's of Megabytes.





Multiple completions

It is often desirable to wait on multiple requests. An example is a master/slave program, where the master waits for one or more slaves to send it a message.

- MPI_Waitall(count, array_of_requests, array_of_statuses)
- MPI_Waitany(count, array_of_requests, index, status)
- MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

There are corresponding versions of test for each of these.

The MPI_WAITSOME and MPI_TESTSOME may be used to implement master/slave algorithms that provide fair access to the master by the slaves.

Fairness

```
What happens with this program:
#include "mpi.h"
#include <stdio.h>
int main(argc, argv)
int argc;
char **argv;
{
int rank, size, i, buf[1];
MPI_Status status;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (rank == 0) {
    for (i=0; i<100*(size-1); i++) {</pre>
        MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,
                 MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        printf( "Msg from %d with tag %d\n",
                status.MPI_SOURCE, status.MPI_TAG );
        }
    }
else {
    for (i=0; i<100; i++)
        MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
    }
MPI_Finalize();
return 0;
}
```

An paralle	el algorithm is <i>f</i>	air if no process
is effectiv	ely ignored. In	the preceeding
program,	processes with	low rank (like
process z messages	ero) may be the are received.	e only one whose
MPI mak	es no guarentee	s about fairness.
However,	MPI makes it p	ossible to write
efficient,	fair programs.	

Providing Fairness

One alternative is

```
#define large 128
MPI_Request requests[large];
MPI_Status statuses[large];
            indices[large];
int
            buf[large];
int
for (i=1; i<size; i++)</pre>
    MPI_Irecv( buf+i, 1, MPI_INT, i,
               MPI_ANY_TAG, MPI_COMM_WORLD, &requests[i-1] );
while(not done) {
    MPI_Waitsome( size-1, requests, &ndone, indices, statuses );
    for (i=0; i<ndone; i++) {</pre>
        j = indices[i];
        printf( "Msg from %d with tag %d\n",
                 statuses[i].MPI_SOURCE,
                 statuses[i].MPI_TAG );
        MPI_Irecv( buf+j, 1, MPI_INT, j,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &requests[j] );
        }
    }
```

Providing Fairness (Fortran)

One alternative is

```
parameter( large = 128 )
   integer requests(large);
   integer statuses(MPI_STATUS_SIZE,large);
   integer indices(large);
   integer buf(large);
   logical done
   do 10 i = 1,size-1
10 call MPI_Irecv( buf(i), 1, MPI_INTEGER, i,
               MPI_ANY_TAG, MPI_COMM_WORLD, requests(i), ierr )
20 if (.not. done) then
    call MPI_Waitsome( size-1, requests, ndone,
                       indices, statuses, ierr )
    do 30 i=1, ndone
        j = indices(i)
        print *, 'Msg from ', statuses(MPI_SOURCE,i), ' with tag',
                     statuses(MPI_TAG,i)
     *
        call MPI_Irecv( buf(j), 1, MPI_INTEGER, j,
                   MPI_ANY_TAG, MPI_COMM_WORLD, requests(j), ierr )
        done = \dots
30
   continue
    goto 20
    endif
```



Objective: Use nonblocking communications Complete the program fragment on "providing fairness". Make sure that you leave no uncompleted requests. How would you test your program?

More on nonblocking communication

In applications where the time to send data between processes is large, it is often helpful to cause communication and computation to overlap. This can easily be done with MPI's non-blocking routines.

For example, in a 2-D finite difference mesh, moving data needed for the boundaries can be done at the same time as computation on the interior.

```
MPI_Irecv( ... each ghost edge ... );
MPI_Isend( ... data for each ghost edge ... );
... compute on interior
while (still some uncompleted requests) {
    MPI_Waitany( ... requests ... )
    if (request is a receive)
        ... compute on that edge ...
    }
```

Note that we call MPI_Waitany several times. This exploits the fact that after a request is satisfied, it is set to MPI_REQUEST_NULL, and that this is a valid request object to the wait and test routines.



Buffered Send

MPI provides a send routine that may be used when MPI_Isend is awkward to use (e.g., lots of small messages).

MPI_Bsend makes use of a *user-provided* buffer to save any messages that can not be immediately sent.

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

The MPI_Buffer_detach call does not complete until all messages are sent.

The performance of MPI_Bsend depends on the implementation of MPI and may also depend on the size of the message. For example, making a message one byte longer may cause a significant drop in performance.

Reusing the same buffer

```
Consider a loop
MPI_Buffer_attach( buf, bufsize );
while (!done) {
    ...
    MPI_Bsend( ... );
    }
where the buf is large enough to hold the message in
the MPI_Bsend. This code may fail because the
```

```
{
void *buf; int bufsize;
MPI_Buffer_detach( &buf, &bufsize );
MPI_Buffer_attach( buf, bufsize );
}
```



ΜF	'I datatypes have two main purposes
•	Heterogenity — parallel programs between different processors
•	Noncontiguous data — structures, vectors with non-unit stride, etc.
Ba: uno	sic datatype, corresponding to the derlying language, are predefined.
Th tim	e user can construct new datatypes at run ie; these are called <i>derived datatypes</i> .

Elementary: Language-defined types (e.g., MPI_INT or MPI_DOUBLE_PRECISION) Vector: Separated by constant "stride" Contiguous: Vector with stride of one Hvector: Vector, with stride in bytes Indexed: Array of indices (for scatter/gather) Hindexed: Indexed, with indices in bytes Struct: General mixed types (for C structs etc.)

Basic Datatypes (Fortran)

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Basic Datatypes (C)

MPI datatype	C datatype
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	

Vectors

29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

To specify this row (in C order), we can use

The exact code for this is

```
MPI_Type_vector( 5, 1, 7, MPI_DOUBLE, &newtype );
MPI_Type_commit( &newtype );
```

Structures

Structures are described by arrays of

- number of elements (array_of_len)
- displacement or location (array_of_displs)
- datatype (array_of_types)

Example: Structures

```
struct {
             display[50]; /* Name of display */
    char
    int
             maxiter;
                            /* max # of iterations */
             xmin, ymin; /* lower left corner of rectangle */
    double
                          /* upper right corner */
    double
             xmax, ymax;
                            /* of display in pixels */
    int
             width;
    int
             height;
                            /* of display in pixels */
} cmdline;
/* set up 4 blocks */
            blockcounts[4] = {50,1,4,2};
int
MPI_Datatype types[4];
MPI_Aint
             displs[4];
MPI_Datatype cmdtype;
/* initialize types and displs with addresses of items */
MPI_Address( &cmdline.display, &displs[0] );
MPI_Address( &cmdline.maxiter, &displs[1] );
MPI_Address( &cmdline.xmin,
                             &displs[2] );
MPI_Address( &cmdline.width, &displs[3] );
types[0] = MPI_CHAR;
types[1] = MPI_INT;
types[2] = MPI_DOUBLE;
types[3] = MPI_INT;
for (i = 3; i >= 0; i--)
    displs[i] -= displs[0];
MPI_Type_struct( 4, blockcounts, displs, types, &cmdtype );
MPI_Type_commit( &cmdtype );
```


```
Vectors revisited
```

This code creates a datatype for an arbitrary number of element in a row of an array stored in Fortran order (column first).

```
int blens[2], displs[2];
MPI_Datatype types[2], rowtype;
blens[0] = 1;
blens[1] = 1;
displs[0] = 0;
displs[1] = number_in_column * sizeof(double);
types[0] = MPI_DOUBLE;
types[0] = MPI_DUBLE;
types[1] = MPI_UB;
MPI_Type_struct( 2, blens, displs, types, &rowtype );
MPI_Type_commit( &rowtype );
```

To send n elements, you can use

MPI_Send(buf, n, rowtype, ...);

Structures revisited

When sending an array of a structure, it is important to ensure that MPI and the C compiler have the same value for the size of each structure. The most portable way to do this is to add an MPI_UB to the structure definition for the end of the structure. In the previous example, this is

```
/* initialize types and displs with addresses of items */
MPI_Address( &cmdline.display, &displs[0] );
MPI_Address( &cmdline.maxiter, &displs[1] );
MPI_Address( &cmdline.xmin,
                               &displs[2] );
MPI_Address( &cmdline.width, &displs[3] );
MPI_Address( &cmdline+1,
                               &displs[4] );
types[0] = MPI_CHAR;
types[1] = MPI_INT;
types[2] = MPI_DOUBLE;
types[3] = MPI_INT;
types[4] = MPI_UB;
for (i = 4; i \ge 0; i--)
    displs[i] -= displs[0];
MPI_Type_struct( 5, blockcounts, displs, types, &cmdtype );
MPI_Type_commit( &cmdtype );
```

Inter	leaving	data

By moving the UB inside the data, you can interleave data.

Consider the matrix

0	8	16	24	32
1	9	17	25	33
2	10	18	26	34
3	11	19	27	35
4	12	20	28	36
5	13	21	29	37
6	14	22	30	38
7	15	23	31	39

We wish to send 0-3,8-11,16-19, and 24-27 to process 0, 4-7,12-15,20-23, and 28-31 to process 1, etc. How can we do this with MPI_Scatterv?

An interleaved datatype

```
MPI_Type_vector( 4, 4, 8, MPI_DOUBLE, &vec );
defines a block of this matrix.
blens[0] = 1; blens[1] = 1;
types[0] = vec; types[1] = MPI_UB;
displs[0] = 0; displs[1] = sizeof(double);
MPI_Type_struct( 2, blens, displs, types, &block );
```

defines a block whose extent is just 1 entries.



We set the displacements for each block as the location of the first element in the block. This works because MPI_Scatterv uses the extents to determine the start of each piece to send.

How would use use the topology routines to make this more general?

Exercises - datatypes

Objective: Learn about datatypes

1. Write a program to send rows of a matrix (stored in column-major form) to the other processors.

Let processor 0 have the entire matrix, which has as many rows as processors.

Processor 0 sends row i to processor i. Processor i reads that row into a local array that holds only that row. That is, processor 0 has a matrix A(N, M) while the other processors have a row B(M).

- (a) Write the program to handle the case where the matrix is square.
- (b) Write the program to handle a number of columns read from the terminal.

C programmers may send columns of a matrix stored in row-major form if they prefer.

If you have time, try one of the following. If you don't have time, think about how you would program these.

2. Write a program to transpose a matrix, where each processor has a part of the matrix. Use topologies to define a 2-Dimensional partitioning of the matrix across the processors, and assume that all processors have the same size submatrix.

- (a) Use MPI_Send and MPI_Recv to send the block, the transpose the block.
- (b) Use MPI_Sendrecv instead.
- (c) Create a datatype that allows you to receive the block already transposed.
- Write a program to send the "ghostpoints" of a 2-Dimensional mesh to the neighboring processors. Assume that each processor has the same size subblock.
 - (a) Use topologies to find the neighbors
 - (b) Define a datatype for the "rows"
 - (c) Use MPI_Sendrecv or MPI_IRecv and MPI_Send with MPI_Waitall.
 - (d) Use MPI_Isend and MPI_Irecv to start the communication, do some computation on the interior, and then use MPI_Waitany to process the boundaries as they arrive

The same approach works for general datastructures, such as unstructured meshes.

4. Do 3, but for 3-Dimensional meshes. You will need MPI_Type_Hvector.



Private	communicators
---------	---------------

One of the first thing that a library should normally do is create private communicator. This allows the library to send and receive messages that are known only to the library. MPI_Comm_dup(old_comm, &new_comm);

Attributes

Attributes are data that can be attached to one or more communicators.

Attributes are referenced by *keyval*. Keyvals are created with MPI_KEYVAL_CREATE.

Attributes are attached to a communicator with MPI_Attr_put and their values accessed by MPI_Attr_get.

Operations are defined for what happens to an attribute when it is copied (by creating one communicator from another) or deleted (by deleting a communicator) when the keyval is created. What is an attribute?

In C, an attribute is a pointer of type void *. You must allocate storage for the attribute to point to (make sure that you don't use the address of a local variable).

In Fortran, it is a single INTEGER.



Sequential Sections

```
#include "mpi.h"
#include <stdlib.h>
static int MPE_Seq_keyval = MPI_KEYVAL_INVALID;
/*@
   MPE_Seq_begin - Begins a sequential section of code.
   Input Parameters:
. comm - Communicator to sequentialize.
      - Number in group. This many processes are allowed
. ng
to execute
   at the same time. Usually one.
@*/
void MPE_Seq_begin( comm, ng )
MPI_Comm comm;
int
         ng;
{
int
           lidx, np;
int
           flag;
           local_comm;
MPI_Comm
MPI_Status status;
/* Get the private communicator for the sequential
operations */
if (MPE_Seq_keyval == MPI_KEYVAL_INVALID) {
    MPI_Keyval_create( MPI_NULL_COPY_FN,
                       MPI_NULL_DELETE_FN,
                       &MPE_Seq_keyval, NULL );
    }
```

Sequential Sections II

```
MPI_Attr_get( comm, MPE_Seq_keyval, (void *)&local_comm,
              &flag );
if (!flag) {
    /* This expects a communicator to be a pointer */
    MPI_Comm_dup( comm, &local_comm );
    MPI_Attr_put( comm, MPE_Seq_keyval,
                  (void *)local_comm );
    }
MPI_Comm_rank( comm, &lidx );
MPI_Comm_size( comm, &np );
if (lidx != 0) {
    MPI_Recv( NULL, 0, MPI_INT, lidx-1, 0, local_comm,
              &status );
    }
/* Send to the next process in the group unless we
   are the last process in the processor set */
if ( (lidx % ng) < ng - 1 && lidx != np - 1) {
    MPI_Send( NULL, 0, MPI_INT, lidx + 1, 0, local_comm );
    }
}
```

Sequential Sections III

```
/*@
   MPE_Seq_end - Ends a sequential section of code.
   Input Parameters:
. comm - Communicator to sequentialize.
. ng
        - Number in group.
@*/
void MPE_Seq_end( comm, ng )
MPI_Comm comm;
int
         ng;
{
int
           lidx, np, flag;
MPI_Status status;
           local_comm;
MPI_Comm
MPI_Comm_rank( comm, &lidx );
MPI_Comm_size( comm, &np );
MPI_Attr_get( comm, MPE_Seq_keyval, (void *)&local_comm,
&flag );
if (!flag)
    MPI_Abort( comm, MPI_ERR_UNKNOWN );
/* Send to the first process in the next group OR to the
first process
   in the processor set */
if ( (lidx % ng) == ng - 1 || lidx == np - 1) {
    MPI_Send( NULL, 0, MPI_INT, (lidx + 1) % np, 0,
local_comm );
    }
if (lidx == 0) {
    MPI_Recv( NULL, 0, MPI_INT, np-1, 0, local_comm,
&status );
    }
}
```



Example: Managing tags

Problem: A library contains many objects that need to communicate in ways that are not known until runtime.

Messages between objects are kept separate by using different message tags. How are these tags chosen?

- Unsafe to use compile time values
- Must allocate tag values at runtime

Solution:

Use a private communicator and use an attribute to keep track of available tags in that communicator.

Caching tags on communicator

```
#include "mpi.h"
static int MPE_Tag_keyval = MPI_KEYVAL_INVALID;
/*
   Private routine to delete internal storage when a
communicator is freed.
 */
int MPE_DelTag( comm, keyval, attr_val, extra_state )
MPI_Comm *comm;
int
         *keyval;
        *attr_val, *extra_state;
void
{
free( attr_val );
return MPI_SUCCESS;
}
```

Caching tags on communicator II

```
/*0
  MPE_GetTags - Returns tags that can be used in
communication with a
  communicator
  Input Parameters:
. comm_in - Input communicator
. ntags - Number of tags
  Output Parameters:
. comm_out - Output communicator. May be 'comm_in'.
. first_tag - First tag available
@*/
int MPE_GetTags( comm_in, ntags, comm_out, first_tag )
MPI_Comm comm_in, *comm_out;
         ntags, *first_tag;
int
ſ
int mpe_errno = MPI_SUCCESS;
int tagval, *tagvalp, *maxval, flag;
if (MPE_Tag_keyval == MPI_KEYVAL_INVALID) {
    MPI_Keyval_create( MPI_NULL_COPY_FN, MPE_DelTag,
                       &MPE_Tag_keyval, (void *)0 );
    }
```

Caching tags on communicator III

```
if (mpe_errno = MPI_Attr_get( comm_in, MPE_Tag_keyval,
&tagvalp, &flag ))
    return mpe_errno;
if (!flag) {
    /* This communicator is not yet known to this system,
so we
       dup it and setup the first value */
    MPI_Comm_dup( comm_in, comm_out );
    comm_in = *comm_out;
    MPI_Attr_get( MPI_COMM_WORLD, MPI_TAG_UB, &maxval,
&flag );
    tagvalp = (int *)malloc( 2 * sizeof(int) );
    printf( "Mallocing address %x\n", tagvalp );
    if (!tagvalp) return MPI_ERR_EXHAUSTED;
    tagvalp = *maxval;
    MPI_Attr_put( comm_in, MPE_Tag_keyval, tagvalp );
    return MPI_SUCCESS;
    }
```

Caching tags on communicator IV

```
*comm_out = comm_in;
if (*tagvalp < ntags) {
    /* Error, out of tags. Another solution would be to do
        an MPI_Comm_dup. */
    return MPI_ERR_INTERN;
    }
*first_tag = *tagvalp - ntags;
*tagvalp = *first_tag;
return MPI_SUCCESS;
}
```

Caching tags on communicator V

```
/*@
  MPE_ReturnTags - Returns tags allocated with MPE_GetTags.
  Input Parameters:
. comm - Communicator to return tags to
. first_tag - First of the tags to return
. ntags - Number of tags to return.
@*/
int MPE_ReturnTags( comm, first_tag, ntags )
MPI_Comm comm;
int
         first_tag, ntags;
{
int *tagvalp, flag, mpe_errno;
if (mpe_errno = MPI_Attr_get( comm, MPE_Tag_keyval,
&tagvalp, &flag ))
    return mpe_errno;
if (!flag) {
    /* Error, attribute does not exist in this communicator
*/
    return MPI_ERR_OTHER;
    }
if (*tagvalp == first_tag)
    *tagvalp = first_tag + ntags;
return MPI_SUCCESS;
}
```

Caching tags on communicator VI

```
/*@
   MPE_TagsEnd - Returns the private keyval.
@*/
int MPE_TagsEnd()
{
   MPI_Keyval_free( &MPE_Tag_keyval );
   MPE_Tag_keyval = MPI_KEYVAL_INVALID;
}
```



- Use MPI_KEYVAL_INVALID to detect when keyval must be created
- Use flag return from MPI_ATTR_GET to detect when a communicator needs to be initialized

Exercise - Writing libraries

Objective: Use private communicators and attributes

Write a routine to circulate data to the next process, using a nonblocking send and receive operation.

```
void Init_pipe( comm )
void ISend_pipe( comm, bufin, len, datatype, bufout )
void Wait_pipe( comm )
A typical use is
Init_pipe( MPI_COMM_WORLD )
for (i=0; i<n; i++) {
    ISend_pipe( comm, bufin, len, datatype, bufout );
    Do_Work( bufin, len );
    Wait_pipe( comm );
    t = bufin; bufin = bufout; bufout = t;
    }
</pre>
```

What happens if Do_Work calls MPI routines?

What do you need to do to clean up Init_pipe?
 How can you use a user-defined topology to determine the next process? (Hint: see MPI_Topo_test and MPI_Cartdim_get.)



MPI has a variety of objects (communicators, groups, datatypes, etc.) that can be created and destroyed. This section discusses the types of these data and how MPI manages them.

> This entire chapter may be skipped by beginners.

The MPI Objects

- MPI_Request Handle for nonblocking communication, normally freed by MPI in a test or wait
- MPI_Datatype MPI datatype. Free with MPI_Type_free.
- MPI_Op User-defined operation. Free with MPI_Op_free.
- MPI_Comm Communicator. Free with MPI_Comm_free.
- MPI_Group Group of processes. Free with MPI_Group_free.
- MPI_Errhandler MPI errorhandler. Free with MPI_Errhandler_free.

Consi	der this code
MPI_T MPI_T	<pre>vector(ly, 1, nx, MPI_DOUBLE, &newx1); /pe_hvector(lz, 1, nx*ny*sizeof(double), newx1,</pre>
MPI_T	<pre>/pe_commit(&newx);</pre>
(This	creates a datatype for one face of a 3-D
decon	position.) When should news1 be freed?

Reference	counting
-----------	----------

MPI keeps track of the use of an MPI object, and only truely destroys it when no-one is using it. newx1 is being used by the user (the MPI_Type_vector that created it) and by the MPI_Datatype newx that uses it.

If newx1 is not needed after newx is defined, it should be freed:

```
Why reference counts
```

Without the MPI_Type_free(&newx1), it would be very awkward to later free newx1 when new was freed.



MPI provides some tools for evaluating the performance of parallel programs.

These are

- Timer
- Profiling interface

The MPI Timer

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("Elapsed time is %f\n", t2 - t1);

The value returned by a single call to MPI_Wtime has little value.

The times are local; the attribute MPI_WTIME_IS_GLOBAL may be used to determine if the times are also synchronized with each other for all processes in MPI_COMM_WORLD.



Writing profiling routines

The MPICH implementation contains a program for writing *wrappers*.

```
This description will write out each MPI routine that
is called :
#ifdef MPI BUILD PROFILING
#undef MPI_BUILD_PROFILING
#endif
#include <stdio.h>
#include "mpi.h"
{{fnall fn_name}}
  {{vardecl int llrank}}
  PMPI_Comm_rank( MPI_COMM_WORLD, &llrank );
  printf( "[%d] Starting {{fn_name}}...\n",
llrank ); fflush( stdout );
  \{\{callfn\}\}
  printf( "[%d] Ending {{fn_name}}\n", llrank );
fflush( stdout );
{{endfnall}}
```

The command

wrappergen -w trace.w -o trace.c

converts this to a C program. The complie the file 'trace.c' and insert the resulting object file into your link line:

```
cc -o a.out a.o ... trace.o -lpmpi -lmpi
```

Another profiling example

```
This version counts all calls and the number of bytes sent with
MPI_Send, MPI_Bsend, Or MPI_Isend.
#include "mpi.h"
{{foreachfn fn_name MPI_Send MPI_Bsend MPI_Isend}}
static long {{fn_name}}_nbytes_{{fileno}};{{endforeachfn}}
{{forallfn fn_name MPI_Init MPI_Finalize MPI_Wtime}}int
{{fn_name}}_ncalls_{{fileno}};
{{endforallfn}}
{{fnall this_fn_name MPI_Finalize}}
  printf( "{{this_fn_name}} is being called.\n" );
  \{\{callfn\}\}
  {{this_fn_name}}_ncalls_{{fileno}}++;
{{endfnall}}
{{fn fn_name MPI_Send MPI_Bsend MPI_Isend}}
  {{vardecl int typesize}}
  {{callfn}}
  MPI_Type_size( {{datatype}}, (MPI_Aint *)&{{typesize}} );
  {{fn_name}}_nbytes_{{fileno}}+={{typesize}}*{{count}}
  {{fn_name}}_ncalls_{{fileno}}++;
\{\{ endfn \} \}
```


Generating and viewing log files

Log files that contain a history of a parallel computation can be very valuable in understanding a parallel program. The upshot and nupshot programs, provided in the MPICH and MPI-F implementations, may be used to view log files



This is very easy with the MPICH implementation of MPI. Simply replace -lmpi with -llmpi -lpmpi -lm in the link line for your program, and relink your program. You do not need to recompile.

On some systems, you can get a real-time animation by using the libraries -lampi -lmpe -lm -lX11 -lpmpi.

Alternately, you can use the -mpilog or -mpianim options to the mpicc Or mpif77 commands.

Connecting several programs together

MPI provides support for connection separate message-passing programs together through the use of *intercommunicators*.

Programs	share MPI_C	OMM_WORLD.	
Programs	have separa	ate and disjo	oint
communic	ators.		
	App1	App2	
	Comm_	intercomm	
	MPI_COI	MM_WORLD	



Collective operations

Use MPI_INTERCOMM_MERGE to create an intercommunicator.



Sharable MPI Resources

- The Standard itself:
 - As a Technical report: U. of Tennessee. report
 - As postscript for ftp: at info.mcs.anl.gov in pub/mpi/mpi-report.ps.
 - As hypertext on the World Wide Web: http://www.mcs.anl.gov/mpi
 - As a journal article: in the Fall issue of the Journal of Supercomputing Applications
- MPI Forum discussions
 - The MPI Forum email discussions and both current and earlier versions of the Standard are available from netlib.
- Books:
 - Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994
 - MPI Annotated Reference Manual, by Otto, et al., in preparation.

Sharable MPI Resources, continued

- Newsgroup:
 - comp.parallel.mpi
- Mailing lists:
 - mpi-comm@mcs.anl.gov: the MPI Forum discussion list.
 - mpi-impl@mcs.anl.gov: the implementors' discussion list.
- Implementations available by ftp:
 - MPICH is available by anonymous ftp from info.mcs.anl.gov in the directory pub/mpi/mpich, file mpich.tar.Z.
 - LAM is available by anonymous ftp from tbag.osc.edu in the directory pub/lam.
 - The CHIMP version of MPI is available by anonymous ftp from ftp.epcc.ed.ac.uk in the directory pub/chimp/release.
- Test code repository:
 - ftp://info.mcs.anl.gov/pub/mpi/mpi-test

MPI-2

- The MPI Forum (with old and new participants) has begun a follow-on series of meetings.
- Goals
 - clarify existing draft
 - provide features users have requested
 - make extensions, not changes
- Major Topics being considered
 - dynamic process management
 - client/server
 - real-time extensions
 - "one-sided" communication (put/get, active messages)
 - portable access to MPI system state (for debuggers)
 - language bindings for C++ and Fortran-90
- Schedule
 - Dynamic processes, client/server by SC '95
 - MPI-2 complete by SC '96



- The parallel computing community has cooperated to develop a full-featured standard message-passing library interface.
- Implementations abound
- Applications beginning to be developed or ported
- MPI-2 process beginning
- Lots of MPI material available