Tutorial on MPI: The Message-Passing Interface

William Gropp

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

Types of parallel computing

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.

Communicating with other processes

Data must be exchanged with other workers

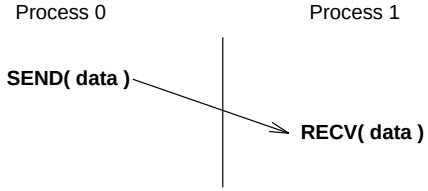
- Cooperative — all parties agree to transfer data
- One sided — one worker performs transfer of data

Cooperative operations

Message-passing is an approach that makes the exchange of data cooperative.

Data must both be explicitly sent and received.

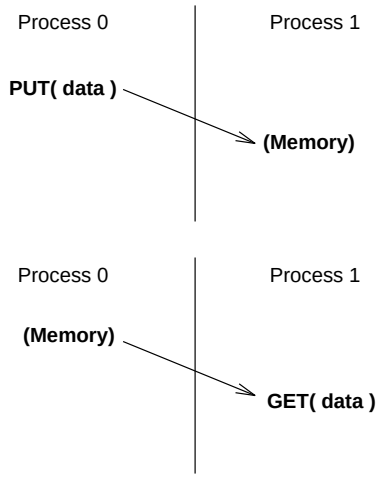
An advantage is that any change in the *receiver's* memory is made with the receiver's participation.



One-sided operations

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



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

Motivation for a New Design

- Message Passing now mature as programming paradigm
  - well understood
  - efficient match to hardware
  - many applications
- Vendor systems not portable
- Portable systems are mostly research projects
  - incomplete
  - lack vendor support
  - not at most efficient level

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

Who Designed MPI?

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

Features of MPI

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

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
- MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, intercommunicators)

Is MPI Large or Small?

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

Where to use MPI?

- 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

Where *not* 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)

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;
}
```


Commentary

- `#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

Compiling and linking

For simple programs, special compiler commands can be used. For large projects, it is best to use a standard Makefile.

The MPICH implementation provides the commands `mpicc` and `mpif77` as well as '`Makefile`' examples in '`/usr/local/mpi/examples/Makefile.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).

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.

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`.

Finding out about the environment

Two of the first questions asked in a parallel program are: How many processes are there? and Who am I?

How many is answered with `MPI_Comm_size` and who am I is answered with `MPI_Comm_rank`.

The rank is a number between zero and `size-1`.

A simple program

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

int main( argc, argv )
int argc;
char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Hello world! I'm %d of %d\n",
        rank, size );
MPI_Finalize();
return 0;
}
```


Current Message-Passing

- A typical blocking send looks like

  `send( dest, type, address, length )`

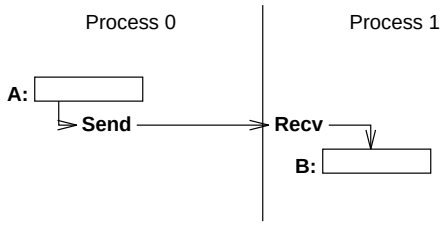
  where
  - `dest` is an integer identifier representing the process to receive the message.
  - `type` is a nonnegative integer that the destination can use to selectively screen messages.
  - (`address, length`) describes a contiguous area in memory containing the message to be sent.

  and
- A typical global operation looks like:

  `broadcast( type, address, length )`

- All of these specifications are a good match to hardware, easy to understand, but too inflexible.

Sending and Receiving messages



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

Delimiting Scope of Communication

- Separate groups of processes working on subproblems
  - Merging of process name space interferes with modularity
  - "Local" process identifiers desirable
- Parallel invocation of parallel libraries
  - Messages from application must be kept separate from messages internal to library.
  - Knowledge of library message types interferes with modularity.
  - Synchronizing before and after library calls is undesirable.

Generalizing the Process Identifier

- 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.

MPI Basic Send/Receive

Thus the basic (blocking) send has become:

```
MPI_Send( start, count, datatype, dest, tag,
          comm )
```

and the receive:

```
MPI_Recv(start, count, datatype, source, tag,
         comm, status)
```

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

Two simple collective operations:

```
MPI_Bcast(start, count, datatype, root, comm)
MPI_Reduce(start, result, count, datatype,
           operation, root, comm)
```


Getting information about a message

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG;
... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

`MPI_TAG` and `MPI_SOURCE` primarily of use when `MPI_ANY_TAG` and/or `MPI_ANY_SOURCE` in the receive.

`MPI_Get_count` may be used to determine how much data of a particular type was received.

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.

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 = 3.14159265358979323846264338327;
  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;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```

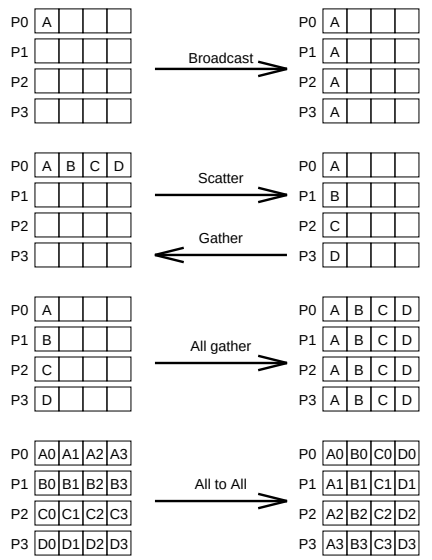

Collective Communications in MPI

- Communication is coordinated among a group of processes.
- Groups can be constructed "by hand" with MPI group-manipulation routines or by using MPI topology-definition routines.
- Message tags are not used. Different communicators are used instead.
- No non-blocking collective operations.
- Three classes of collective operations:
  - synchronization
  - data movement
  - collective computation

Synchronization

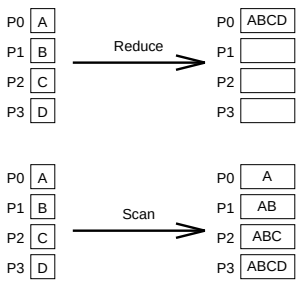
- `MPI_Barrier(comm)`
- Function blocks untill all processes in `comm` call it.

Available Collective Patterns



Schematic representation of collective data movement in MPI

Available Collective Computation Patterns



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		

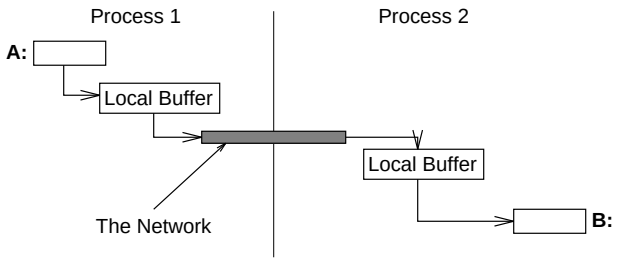
- `All` versions deliver results to all participating processes.
- `V` versions allow the chunks to have different sizes.
- Allreduce, Reduce, ReduceScatter, and Scan take both built-in and user-defined combination functions.

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

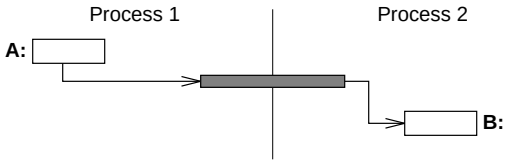
Buffering issues

Where does data go when you send it? One possibility is:



Better buffering

This is not very efficient. There are three copies in addition to the exchange of data between processes. We prefer



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.

Blocking and Non-Blocking communication

- So far we have used `blocking` communication:
  - `MPI_Send` does not complete until buffer is empty (available for reuse).
  - `MPI_Recv` does not complete until buffer is full (available for use).
- Simple, but can be "unsafe":

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

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.

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, with `MPI_Sendrecv`:

Process 0	Process 1
Sendrecv(1)	Sendrecv(0)

- Use non-blocking operations:

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

- Use `MPI_Bsend`

MPI's Non-Blocking Operations

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

- `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)`

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++) {
        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;
}
```


Fairness in message-passing

An parallel algorithm is *fair* if no process is effectively ignored. In the preceeding program, processes with low rank (like process zero) may be the only one whose messages are received.

MPI makes no guarentees about fairness. However, MPI makes it possible to write efficient, fair programs.

Providing Fairness

One alternative is

```
#define large 128
MPI_Request requests[large];
MPI_Status  statuses[large];
int         indices[large];
int         buf[large];
for (i=1; i<size; i++)
    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++) {
        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] );
        }
    }
```

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.

Communication Modes

MPI provides mulitple *modes* for sending messages:

- Synchronous mode (`MPI_Ssend`): the send does not complete until a matching receive has begun. (Unsafe programs become incorrect and usually deadlock within an `MPI_Ssend`.)
- Buffered mode (`MPI_Bsend`): the user supplies the buffer to system for its use. (User supplies enough memory to make unsafe program safe).
- Ready mode (`MPI_Rsend`): user guarantees that matching receive has been posted.
  - allows access to fast protocols
  - undefined behavior if the matching receive is not posted

Non-blocking versions:
`MPI_Issend`, `MPI_Irsend`, `MPI_Ibsend`

Note that an `MPI_Recv` may receive messages sent with *any* send mode.

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.*

Tools for writing libraries

MPI is specifically designed to make it easier to write message-passing libraries

- Communicators solve tag/source wild-card problem
- Attributes provide a way to attach information to a communicator

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 );
```


MPI Objects

- *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`.

Tools for evaluating programs

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`:

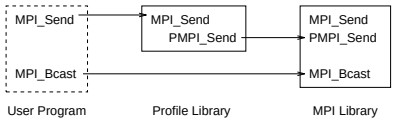
```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "Elapsed time is %f\n", t2 - t1 );
```

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`.

Profiling

- All routines have two entry points: `MPI_...` and `PMPI_...`
- This makes it easy to provide a single level of low-overhead routines to intercept MPI calls without any source code modifications.
- Used to provide "automatic" generation of trace files.




```
static int nsend = 0;
int MPI_Send( start, count, datatype, dest, tag, comm )
{
nsend++;
return PMPI_Send( start, count, datatype, dest, tag, comm )
}
```


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
```


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

Summary

- 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