Message-Passing Interface

Fundamentals

Jack Dongarra

Computer Science Department
University of Tennessee

and

Mathematical Sciences Section
Oak Ridge National Laboratory

Work supported in part by ARPA, Department of Energy, and the National Science Foundation and in cooperation with the Center for Research on Parallel Computing.

Outline

- Motivation for MPI
- The process that produced MPI
- What is different about MPI?
  - the “usual” send/receive
  - the MPI send/receive
  - simple collective operations
- New in MPI: Not in MPI
- Some simple complete examples, in Fortran and C
- Communication modes, more on collective operations
- Implementation status
- MPICH - a free, portable implementation
- MPI resources on the Net
- MPI-2

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

Motivation (cont.)

Few systems offer the full range of desired features.

- modularity (for libraries)
- 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.netlib.org/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

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

New 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

New 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.

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 an integer that the destination can use to selectively screen messages.
  - (`length, address`) 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.

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

Generalizing the Buffer Description

- Specified in MPI by ***starting address***, ***datatype***, and ***count***, where datatype is:
  - elementary (all C and Fortran datatypes)
  - contiguous array of datatypes
  - strided blocks of datatypes
  - indexed array of blocks of datatypes
  - general structure
- Datatypes are constructed recursively.
- Specifications of elementary datatypes allows heterogeneous communication.
- Elimination of length in favor of count is clearer.
- Specifying application-oriented layout of data allows maximal use of special hardware.

Generalizing the Type

- A single type field is too constraining. Often overloaded to provide needed flexibility.
- Problems:
  - under user control
  - wild cards allowed
  - library use conflicts with user and with other libraries
- Solution:
  - a separate communication *context* for each family of messages, used for queueing and matching. (This has often been simulated in the past by overloading the tag field.)
  - No wild cards allowed, for security
  - Allocated by the system, for security
  - Types (*tags*, in MPI) retained for normal use (wild cards OK)

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 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)
```

Simple Fortran example

```
      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
C
      if (rank .eq. src) then
         to     = dest
         count  = 10
         tag    = 2001
         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
         tag   = MPI_ANY_TAG
         count = 10
         from  = MPI_ANY_SOURCE
         call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +                 tag, MPI_COMM_WORLD, status, ierr )
```

Simple Fortran example (cont.)

```
         call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
     +                       st_count, ierr )
         st_source = status(MPI_SOURCE)
         st_tag    = status(MPI_TAG)
C
         print *, 'Status info: source = ', st_source,
     +            ' tag = ', st_tag, ' count = ', st_count
         print *, rank, ' received', (data(i),i=1,10)
      endif

      call MPI_FINALIZE( ierr )
      end
```

Fortran example

```
      program main

      include "mpif.h"

      double precision  PI25DT
      parameter        (PI25DT = 3.141592653589793238462643d0)

      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, rc
c                                 function to integrate
      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 )

 10   if ( myid .eq. 0 ) then
         write(6,98)
 98      format('Enter the number of intervals: (0 quits)')
         read(5,99) n
 99      format(i10)
      endif

      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

Fortran example (cont.)

```
c                                 check for quit signal
      if ( n .le. 0 ) goto 30

c                                 calculate the interval size
      h = 1.0d0/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

c                                 collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)

c                                 node 0 prints the answer.
      if (myid .eq. 0) then
         write(6, 97) pi, abs(pi - PI25DT)
 97      format('  pi is approximately: ', F18.16,
     +          '  Error is: ', F18.16)
      endif

      goto 10

 30   call MPI_FINALIZE(rc)
      stop
      end
```

C example

```
#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;

    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();
}
```

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.

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

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)`
- `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)`

One can also test without waiting: `MPI_Test`, etc.

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.)
- 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

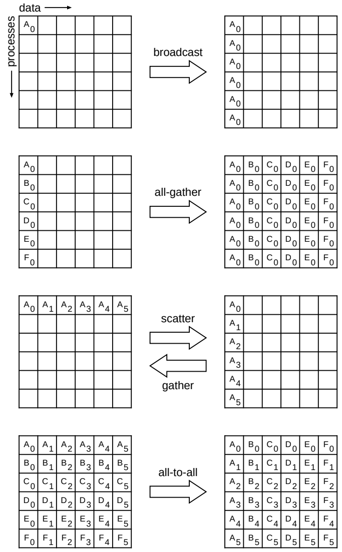
Non-blocking versions:
`MPI_Issend`, `MPI_Irsend`, `MPI_Ibsend`

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.
- Two classes of collective operations:
  - data movement
  - collective computation

Available Collective 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, Allreducev, Reduce, Reducev, 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}
```

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

We have covered

- Background and scope of MPI
- Some characteristic features of MPI (communicators, datatypes)
- Point-to-Point communication
  - blocking and non-blocking
  - multiple modes
- Collective communication
  - data movement
  - collective computation

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

Current MPI Implementation Efforts

Vendor Implementations
IBM Research (MPI-F)
IBM Kingston
Intel SSD
Cray Research
Meiko, Inc.
SGI
Kendall Square Research
NEC
Fujitsu (AP1000)
Convex
Hughes Aircraft
Portable Implementations
Argonne–Mississippi State (MPICH)
Ohio supercomputer Center (LAM)
University of Edinburgh
Technical University of Munich
University of Illinois

Other interested groups: Sun, Hewlett-Packard, Myricom (makers of high-performance network switches) and PALLAS (a German software company), Sandia National Laboratory (Intel Paragon running SUNMOS)

MPI Implementation Projects

- Variety of implementations
  - Vendor proprietary
  - Free, portable
  - World wide
  - Real-time, embedded systems
  - All MPP's and networks
- Implementation strategies
  - Specialized
  - Abstract message-passing devices
  - Active-message devices

MPICH – A Freely-available Portable MPI Implementation

- Complete MPI implementation
- On MPP's: IBM SP1 and SP2, Intel IPSC860 and Paragon, TMC CM-5, SGI, Meiko CS-2, NCube, KSR, Sequent Symmetry
- On workstation networks: Sun, Dec, HP, SGI, Linux, FreeBSD, NetBSD
- Includes multiple profiling libraries for timing, event logging, and animation of programs.
- Includes trace upshot visualization program, graphics library
- Efficiently implemented for shared-memory, high-speed switches, and network environments
- Man pages
- Source included
- Available at ftp.mcs.anl.gov in pub/mpi/mpich.tar.Z

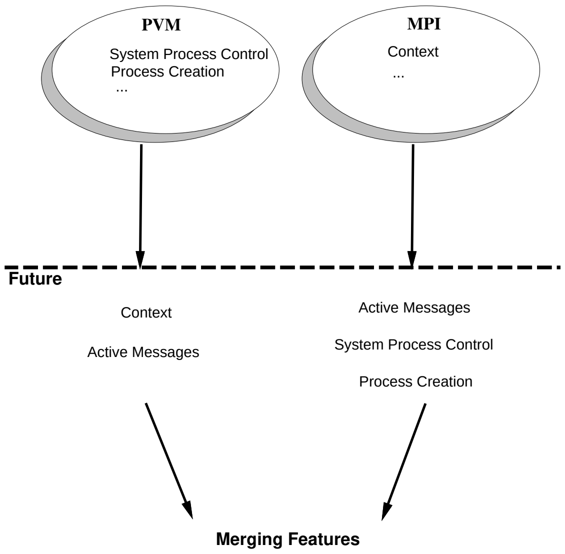
Sharable MPI Resources

- 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, Dongarra, Lederman, Snir, and Walker, in preparation.

Sharable MPI Resources, continued

- Newsgroup:
  - `comp.parallel.mpi`
- Mailing lists:
  - `mpi-comm@cs.utk.edu`: 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 (new):
  - `ftp://info.mcs.anl.gov/pub/mpi-test`

PVM and MPI Future



MPI available on:
IBM SP
Intel Paragon
Cray T3D
Meiko CS–2
PVM/p4

Conclusions

- MPI being adopted worldwide
- Standard documentation is an adequate guide to implementation
- Implementations abound
- Implementation community working together