MPI: Message Passing Interface

David W. Walker
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367

walker@msr.epm.ornl.gov

WEB: http://www.epm.ornl.gov/~walker/

Presented at
CERN School of Computing, Arles, France
August 25, 1995

What Is In MPI?

- Point-to-point message passing
- Collective communication
- Support for process groups
- Support for design of safe, modular libraries
- Support for application topologies
- Environmental inquiry routines
- Profiling interface

Point-to-point Communication

- MPI provides for point-to-point communication between pairs of processes.
- Message selectivity is by rank and message tag.
- Rank and tag are interpreted relative to the scope of the communication.
- The scope is specified by the communicator.
- Rank and tag may be wildcarded.
- The components of a communicator may not be wildcarded.

Communication Completion

- A communication operation is **locally complete** on a process if the process has completed its part in the operation.
- A communication operation is **globally complete** if all processes involved have completed their part in the operation.

A communication operation is globally complete
if and only if
it is locally complete for all processes.

Blocking Send

- Consider the standard blocking send routine:

```
mpi_send (
           IN  start_of_buffer,
           IN  number_of_items,
           IN  datatype,
           IN  destination_rank,
           IN  tag,
           IN  communicator,
           OUT error_code)
```

Blocking Receive

- Consider the standard blocking receive routine:

```
mpi_recv (
          OUT start_of_buffer,
          IN  max_number_of_items,
          IN  datatype,
          IN  source_rank,
          IN  tag,
          IN  communicator,
          OUT return_status,
          OUT error_code)
```

- In a receive routine, `source_rank` and `tag` can have the values `mpi_any_source` and `mpi_any_tag`.

Return Status Objects

- The return status object is used after completion of a receive to find the actual length, source, and tag of a message.
- Return status object is MPI-defined type.
- In C **`status`** is a structure.

  status.source gives source process

  status.tag gives the message tag
- In Fortran **`status`** is an integer array

  status(MPI_SOURCE) gives source process

  status(MPI_TAG) gives message tag
- Number of elements in message is given by

```
mpi_get_count (status, datatype, count, ierr)
```

Blocking Behavior

- **Nonblocking send:**

  returns “immediately”

  message buffer should not be written to after return

  must check for local completion

- **Blocking send:**

  returns when send is locally complete

  message buffer can be written to after return

- **Nonblocking receive:**

  returns “immediately”

  message buffer should not be read from after return

  must check for local completion

- **Blocking receive:**

  returns when receive is locally complete

  message buffer can be read from after return

Nonblocking Send

- Consider the standard nonblocking send routine:

```
mpi_isend (
          IN  start_of_buffer,
          IN  number_of_items,
          IN  datatype,
          IN  destination_rank,
          IN  tag,
          IN  communicator,
          OUT request_id,
          OUT error_code)
```

Nonblocking Receive

- Consider the standard nonblocking receive routine:

```
mpi_irecv (
          OUT start_of_buffer,
          IN  max_number_of_items,
          IN  datatype,
          IN  source_rank,
          IN  tag,
          IN  communicator,
          OUT request_id,
          OUT error_code)
```

- Receive is not passed a return status object.

Completion Routines

- Two basic ways of checking on nonblocking sends and receives:

  call a wait routine that blocks until completion

  call a test routine that returns a flag to indicate if complete

- Use of nonblocking and completion routines allow computation and communication to be overlapped.

```
mpi_wait (request_id,
          return_status, ierr)
mpi_test (request_id, flag,
          return_status, ierr)
```

- `mpi_wait` blocks until the communication is complete.
- `mpi_test` returns "immediately", and sets `flag` to `true` is the communication is complete.

Multiple Completions

- Versions of wait and status routines that act on multiple communication operations.
- Block until all communication operations in a given list have completed.

```
mpi_waitall (count, list_requests,
             list_status, ierr)
```

- Block until one of the communication operations in a given list has completed

```
mpi_waitany (count, list_requests,
             index, return_status, ierr)
```

- Block until at least one of the communication operations in a given list has completed

```
mpi_waitsome (count, list_requests,
              count_done, list_index,
              list_status, ierr)
```

- There are similar test routines for checking completion status of a list of communication operations.

Communication Modes

- The mode of a point-to-point communication operation governs when a send operation is initiated, or when it completes.
- **Standard mode:**
  - A send may be initiated even if a matching receive has not been initiated.
- **Ready mode:**
  - A send may be initiated only if a matching receive has been initiated.
- **Synchronous mode:**
  - The same as standard mode, except the send will not complete until message delivery is guaranteed.
- **Buffered mode:**
  - Similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this.

Buffered Mode

- In buffered mode a user-supplied buffer may be used to buffer messages so that the sending process can *always* return from the send before the message just been received.
- To supply the system with the user buffer:

  `mpi_buffer_attach (buffer, size, ierr)`

- To get user buffer back from system:

  `mpi_buffer_detach (buffer, size, ierr)`

- This will block until all communication using `buffer` has completed.

Naming Conventions

SEND	Blocking	Nonblocking
Standard	`mpi_send`	`mpi_isend`
Ready	`mpi_rsend`	`mpi_irsend`
Synchronous	`mpi_ssend`	`mpi_issend`
Buffered	`mpi_bsend`	`mpi_ibsend`

RECEIVE	Blocking	Nonblocking
Standard	`mpi_recv`	`mpi_irecv`

- Any type of receive routine can be used to receive messages from any type of send routine.

Send/Receive Operations

- In many applications, processes send to one process while receiving from another.
- Deadlock may arise if care is not taken.
- MPI provides routines for such send/receive operations.
- For distinct send/receive buffers:

  `mpi_sendrecv`

- For identical send/receive buffers:

  `mpi_sendrecv_replace`

Communicating Non-Primitive Datatypes

- Primitive datatypes:

```
MPI_INTEGER, MPI_REAL, MPI_DOUBLE
MPI_COMPLEX, MPI_DOUBLE_COMPLEX
MPI_LOGICAL, MPI_CHARACTER, MPI_BYTE
```

- MPI also supports array sections and structures through general datatypes.
- A general datatype is a sequence of primitive types and integer byte displacements

$$\text{Datatype} = \{\, (\text{type}_0, \text{disp}_0),\ (\text{type}_1, \text{disp}_1), \ldots$$
$$\ldots,\ (\text{type}_{n-1}, \text{disp}_{n-1})\,\}$$

- Together with a base address, a datatype defines a communication buffer.

Contiguous Datatype Constructor

- A general datatype is built up hierarchically from simpler components.

```
mpi_type_contiguous (count, oldtype,
                     newtype, ierr)
```

- The above creates a new datatype made up of `count` repetitions of `oldtype`.
- For example,

```
oldtype = {(double, 0), (char, 8)}
```

then if `count` $= 3$,

```
newtype = {(double,  0), (char,  8)
           (double, 16), (char, 24)
           (double, 32), (char, 40)}
```

Vector Datatype Constructor

- This constructor replicates a datatype, taking blocks at fixed offsets.

```
mpi_type_vector (count, blocklen,stride,
                 oldtype, newtype, ierr)
```

- The new datatype consists of:
  - `count` blocks,
  - each block is a repetition of `blocklen` items of `oldtype`,
  - the start of successive blocks is offset by `stride` items of `oldtype`.
- If `count=2`, `stride=4`, `blocklen=3`, then `newtype` is:

```
{(double,  0), (char,   8),
 (double, 16), (char,  24),
 (double, 32), (char,  40),
 (double, 64), (char,  72),
 (double, 80), (char,  88),
 (double, 96), (char, 104)}
```

Indexed Datatype Constructor

- This constructor replicates a datatype, taking blocks at fixed offsets.

```
mpi_type_indexed (count, B, I, oldtype,
                  newtype, ierr)
```

- The new datatype consists of:
  - `count` blocks,
  - the ith block is of length `B[i]` items of `oldtype`,
  - the offset of the start of the ith block is `I[i]` items of `oldtype`.
- If `count=2`, `I` $= \{64, 0\}$, and `B` $= \{3, 1\}$, then `newtype` is:

```
{(double, 64),  (char,  72),
 (double, 80),  (char,  88),
 (double, 96),  (char, 104),
 (double,  0),  (char,   8)}
```

Structure Datatype Constructor

- This constructor generalizes the indexed datatype by allowing each block to be of a different datatype.

```
mpi_type_struct (count, B, I, T,
                 newtype, ierr)
```

- The new datatype consists of:
  - `count` blocks,
  - the length of the ith block is `B[i]` items of type `T[i]`,
  - the offset of the start of the ith block is `I[i]` bytes.
- If `count=3`, `T` $= \{$`MPI_FLOAT`, `type1`, `MPI_CHAR`$\}$, `I` $= \{0, 16, 26\}$, and `B` $= \{2, 1, 3\}$, then `newtype` is:

```
{(float, 0), (float, 4),
 (double, 16), (char, 24),
 (char, 26), (char, 27), (char, 28)}
```

Other Datatype Routines

- The size of a datatype is termed the extent.
- The extent is given by:

```
mpi_type_extent (datatype, extent, ierr)
```

- `mpi_type_hvector` is same as `mpi_type_vector`, except stride is given in bytes.
- `mpi_type_hindexed` is same as `mpi_type_indexed`, except offsets in array of offsets are given in bytes.
- The offsets in a datatype may be given relative to a "base address," given by the MPI constant `mpi_bottom`.
- The address of a location can be found thus:

```
mpi_address (location, address, ierr)
```

Persistent Communication Requests

- Can bind together arguments of a communication call to take overhead out of a loop
- `mpi_send_init` creates a communication request that completely specifies a standard send operation. It binds:
  - start of buffer, and number of elements sent,
  - datatype,
  - destination rank, message tag, and communicator
- `mpi_recv_init` creates a communication request that completely specifies a receive operation.
- Similar routines for ready, synchronous, and buffered send modes.

Persistent Communication Requests

- To start a send or receive:

```
mpi_start (request, ierr)
mpi_start_all (count, requestarray, ierr)
```

- The wait and test routines can be used to block until completion, or to check on status.

MPI Integration Example

Want to find

$$\int_0^{\pi} \sin x \, dx$$

1. **Initialization**
   (a) initialize MPI
   (b) communicate problem parameters
2. **Compute**
   (a) each process computes its contribution
   (b) reduction operation sums process contributions
3. **Output**

   Process with rank 0 outputs the result
4. **Tidy Up**

   All processes call `mpi_finalize`

MPI Code for Integration

Initialization...

```
program integrate
include 'mpif.h'
parameter (pi=3.141592654)
integer rank
call mpi_init (ierr)
call mpi_comm_rank (mpi_comm_world, rank, ierr)
call mpi_comm_size (mpi_comm_world, nprocs, ierr)
if (rank .eq. 0) then
   open (7, file='input.dat')
   read (7,*) npts
end if
call mpi_bcast (npts, 1, mpi_integer, 0, mpi_comm_world, ierr)
nlocal = (npts-1)/nprocs + 1
nbeg   = rank*nlocal + 1
nend   = min (nbeg+nlocal-1,npts)
```

MPI Code for Integration

…computation, output, and exit

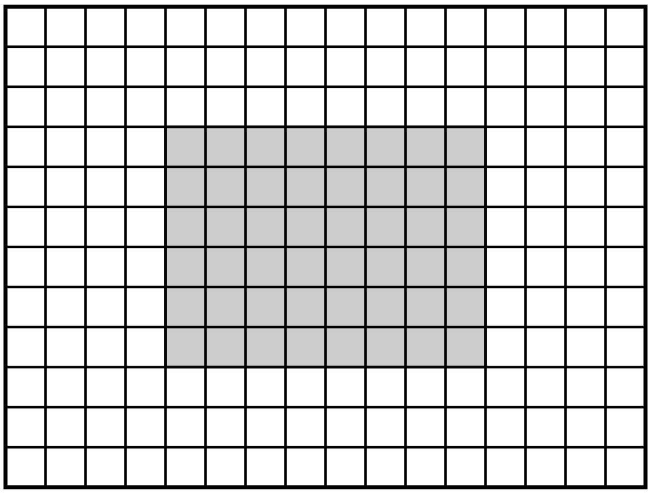
```
 deltax = pi/npts
 psum   = 0.0
 do i=nbeg,nend
    x = (i-0.5)*deltax
    psum = psum + sin(x)
 end do
 call mpi_reduce (psum, sum, 1, mpi_real, mpi_sum, 0,
#                  mpi_comm_world, ierr)
 if (rank.eq. 0) then
    print *,'The integral is ',sum*deltax
 end if
 call mpi_finalize (ierr)
 stop
 end
```

Laplace Solver

- Solution of 2D Laplace equation

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

on rectangular domain with $\phi = 0$ on boundary, and $\phi = 1$ on an interior rectangular region



- Solution by Jacobi iteration;

$$\phi^n_{i,j} = \frac{1}{4}\left(\phi^{n-1}_{i-1,j} + \phi^{n-1}_{i+1,j} + \phi^{n-1}_{i,j-1} + \phi^{n-1}_{i,j+1}\right)$$

Skeleton Laplace Code

1. Initialize MPI and communicate problem parameters
2. Initialize grid `phi`. Use logical array, `mask`, to identify boundaries ($\phi = 0$) and fixed interior region ($\phi = 1$).
3. Set up 2D topology and identify neighbors
4. For each iteration
   (a) Copy `phi` array to `oldphi` array
   (b) Exchange boundary data
   (c) Update `phi` array
5. Output final `phi` array
6. Tidy up and exit

MPI Code for Laplace Solver

Initialization. . .

```
call mpi_init (ierr)
call mpi_comm_rank (mpi_comm_world, rank, ierr)
if (rank .eq. 0) then
   open (7, file='jacobi.input')
   read (7,*) ibuf(1), ibuf(2)
   read (7,*) ibuf(3), ibuf(4)
end if
call mpi_bcast (ibuf, 4, mpi_integer, 0, mpi_comm_world, ierr)
npx    = ibuf(1)
npy    = ibuf(2)
nptsx  = ibuf(3)
nptsy  = ibuf(4)
```

Creating a 2D Topology

```
 periods(1) = .false.
 periods(2) = .false.
 reorder    = .false.
 dims(1)    = npy
 dims(2)    = npx
 call mpi_cart_create (mpi_comm_world, 2, dims, periods,
#                       reorder, comm2d, ierr)
 call mpi_cart_coords (comm2d, rank, 2, coords, ierr)
 ipy = coords(1)
 ipx = coords(2)
 call mpi_cart_shift (comm2d, 0,  1, dsrc, ddest, ierr)
 call mpi_cart_shift (comm2d, 0, -1, usrc, udest, ierr)
 call mpi_cart_shift (comm2d, 1,  1, rsrc, rdest, ierr)
 call mpi_cart_shift (comm2d, 1, -1, lsrc, ldest, ierr)
```

Creating Datatypes

Here we create datatypes for communicating the boundary data

```
call mpi_type_contiguous (nptsy, mpi_real, contig, ierr)
call mpi_type_vector (nptsx, 1, nymax+2, mpi_real, strided, ierr)
call mpi_type_commit (contig, ierr)
call mpi_type_commit (strided, ierr)
```

- datatype `contig` picks out columns of the array `oldphi`
- datatype `strided` picks out rows of `oldphi`
- `oldphi` is declared as

  ```
  real oldphi(0:nymax+1,0:nxmax+1)
  ```

Main Loop of Laplace Solver

The iterative loop of the Laplace solver does the following

- Copy solution array `phi` to `oldphi`
- Communicate boundary grid data between process using shift operations
- Update solution as follows

```
 do j=1,nptsx
    do i=1,nptsy
       if (mask(i,j)) phi(i,j) =
#             0.25*(oldphi(i-1,j) + oldphi(i+1,j) +
#                   oldphi(i,j-1) + oldphi(i,j+1))
    end do
 end do
```

Communicating Boundary Data

- Shift left

```
 call mpi_sendrecv (oldphi(1,1), 1, contig, ldest, 30,
#                   oldphi(1,nptsx+1), 1, contig, lsrc, 30,
#                   comm2d, rstatus, ierr)
```

- Shift right

```
 call mpi_sendrecv (oldphi(1,nptsx), 1, contig, rdest, 31,
#                   oldphi(1,0), 1, contig, rsrc, 31,
#                   comm2d, rstatus, ierr)
```

- Shift down

```
 call mpi_sendrecv (oldphi(nptsy,1), 1, strided, ddest, 41,
#                   oldphi(0,1), 1, strided, dsrc, 41,
#                   comm2d, rstatus, ierr)
```

- Shift up

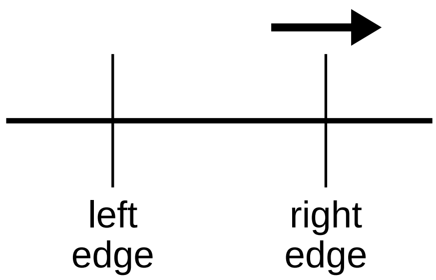
```
 call mpi_sendrecv (oldphi(1,1), 1, strided, udest, 40,
#                   oldphi(nptsy+1,1), 1, strided, usrc, 40,
#                   comm2d, rstatus, ierr)
```

Particle Migration Example

- **Problem:**

  Particles move on a domain that is block distributed. Particle data must be moved between neighboring process subdomains when they cross subdomain boundaries.

- Since each process subdomain has 6 surfaces (for 3D), the migration can be done in 6 communications. We shall just consider the migration across the righthand boundary.



Creating a 3D Topology

- A 3D virtual topology may be created as follows:

```
int dims[3]={0,0,0}, periods[3]={0,1,1};
int reorder=0, nprocs, ierr;
MPI_Comm comm, comm3d;

ierr = MPI_Comm_size (comm, &nprocs);
ierr = MPI_Dims_create (nprocs, ndims, dims);
ierr = MPI_Cart_create (comm, ndims, dims,
                        periods, reorder, comm3d);
```

Skeleton Code

```
int ierr, status, errcode=21;

ierr = MPI_Topo_test (comm3d, &status);
if (status != MPI_CART)
    ierr = MPI_Abort (comm3d, errcode);

CREATE PARTICLE DATATYPE

CREATE SEND/RECV DATATYPES

COMMUNICATE DATA
```

Creating the Particle Datatype

```
struct Pstruct {double x[3]; double v[3]; int k;}
     particle[100];
MPI_Datatype Stype[3], Dtype3, Ptype;
int Sblock[3] = {1, 1, 1};
MPI_Aint Sindex[3];
int sized3;

ierr = MPI_Type_contiguous (3, MPI_DOUBLE, &Dtype3);
ierr = MPI_Type_extent (Dtype3, &sized3);
Sindex[0] = 0;
Sindex[1] = sized3;
Sindex[2] = sized3*2;
Stype[0]  = Dtype3;
Stype[1]  = Dtype3;
Stype[2]  = MPI_INT;
ierr = MPI_Type_struct (3, Sblock, Sindex,
                        Stype, &Ptype);
```

Creating the Send/Receive Datatypes

```
MPI_Datatype Sendtype;
int npart, scount, i, rightedge;
int Pindex[100], Pblock[100];

scount = 0;
for (i=0;i<npart;i++)
   if (particle[i].x[0] > rightedge){
      Pindex[scount] = i; Pblock[scount] = 1;
      scount++;}

ierr = MPI_Type_index (scount, Pblock, Pindex,
                       Ptype, &Sendtype);
ierr = MPI_Type_commit (&Sendtype);
ierr = MPI_Type_commit (&Ptype);
```

Communicating the Data

```
int dir=0, disp=1, source, dest, rcount;
MPI_Status cstatus;

ierr = MPI_Cart_shift (comm3d, dir, disp,
                       source, dest);

ierr = MPI_Sendrecv(
          particle, 1, Sendtype, dest, 0,
          &particle[npart], space, Ptype,
          source, 0, comm3d, &cstatus);

ierr = MPI_Get_count (&cstatus, Ptype, &rcount);

REPACK PARTICLE ARRAY
```

Extensions to MPI

Possible extensions to MPI currently under consideration include:

- **External Interfaces:** This will define interfaces for external packages such as debuggers and profilers.
- **One-Sided Communications:** This will extend MPI to allow shared memory type operations (e.g., put/get), messages with interrupts (e.g., active messages), and Read-Modify-Write operations (e.g., fetch and add).
- **Dynamic Processes:** This will extend MPI to allow for the spawning and destruction of processes after `MPI_Init`
- **Extended Collective:** This will extend the collective calls to be non-blocking and apply to inter-communicators.
- **Bindings:** This will produce bindings for Fortran 90 and C++.