MPI-2

Based on notes by Sathish Vadhiyar

- Using MPI2: Advanced Features of the Message-Passing Interface.
  http://www-unix.mcs.anl.gov/mpi/usingmp2/

MPI History

- Standardization started (1992)
- MPI-1 completed (1.0) (May 1994)
- Clarifications (1.1) (June 1995)
  - MPI-1.2 issued in 1995 (minor corrections/clarifications)

MPI-2

- Dynamic process creation and management
- One sided communications
- Extended collective operations
- Parallel I/O
- Miscellany

Process Creation and Management

- MPI-1 application is static
- MPI-2: process creation after MPI application has started
- Motivation:
  - Task farming applications
  - Serial applications executing parallel codes
  - To be friendly to the PVM users

12 January 2004
Process Creation - Features

- Creation and cooperative termination
- Communication between new processes and existing application
- Communication between 2 MPI applications

API

MPI_COMM_GET_PARENT(parent)
OUT parent - the parent communicator

- for starting multiple binaries

Info Object for MPI_Spawn

- MPI_INFO_CREATE(info)
- MPI_INFO_SET(info, key, value)

Relevant {key, value} for MPI_Spawn:
{soft, a}
{soft, a:b}
{soft, a:b:c}

Can be used to say which hosts to use, and whether to return if it can't start all processes e.g.
MPI_Info_set(hostinfo, "file", "spawnhostfile");
MPI_Info_set(hostinfo, "soft", soft_limits);
Spawn Example – Master/worker

/* master */
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    if (world_size != 1) error("Top heavy with management");
    MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                 &universe_sizep, &flag);
    choose_worker_program(worker_program);
    MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size - 1,
                   MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
                   MPI_ERRCODES_IGNORE);
    …
    …
    MPI_Finalize();
    return 0;
}

Example continued

/* worker */
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_get_parent(&parent);
    if (parent == MPI_COMM_NULL) error("No parent!");
    MPI_Comm_remote_size(parent, &size);
    if (size != 1) error("Something's wrong with the parent");
    …
    …
    MPI_Finalize();
    return 0;
}

Communication between applications

• Communication between 2 independently started applications
• Follows client/server type of communication
• Collective operation
• Operates by means of port names and/or service names

Establishing Connection

• Similar to sockets but for groups of processes
• Ports, IP addresses, name servers
• Port name (MPI_OPEN_PORT, MPI_ACCEPT)
• Service name (MPI_LOOKUP_NAME)
• Portability varies
API

- MPI_OPEN_PORT(info, port_name) - server
- MPI_CLOSE_PORT(port_name) - server
- MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm) - server accepts connection and returns a new socket
- MPI_COMM_CONNECT(port_name, info, root, comm, newcomm) - client connects to server socket
- Also, MPI_COMM_JOIN(fd, intercomm)

Example 1

Server:

```c
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/*...*/
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
```

Client:

```c
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```

Example 2 – Publishing names

Server:

```c
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
```

Client:

```c
MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```
One Sided communications

Motivation

- Remote memory access (RMA)
- All communication parameters on one side (sender/receiver)
- For applications that have dynamic data access patterns
- For using hardware provided features
- Consists of communication (put, get, update) and synchronization functions

Allowing memory accesses

- Collective operations – create RMA window object
  - MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)
  - MPI_WIN_FREE(win)

Other window operations:

- MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)
- MPI_WIN_GET_GROUP(win, group)
- MPI_WIN_FENCE(assert, win) – synchronize beginning of RMA access epoch

Communication Calls

- 3 non-blocking calls:
  - MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win) for writing to remote memory
  - MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win) for reading from remote memory
  - MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win) for updating remote memory
Example - Get

- To compute $A = B(\text{map})$

```fortran
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
    USE MPI
    INTEGER m, map(m), comm, p
    REAL A(m), B(m)
    INTEGER sizeofreal, win, ierr
    CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
    CALL MPI_WIN_CREATE(B, m* sizeofreal, sizeofreal, MPI_INFO_NULL, &
                         comm, win, ierr)
    CALL MPI_WIN_FENCE(0, win, ierr)
    DO i=1,m
        j = map(i)/p
        k = MOD(map(i),p)
        CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
    END DO
    CALL MPI_WIN_FENCE(0, win, ierr)
    CALL MPI_WIN_FREE(win, ierr)
RETURN END
```

Example - Accumulate

- To update $B(j) = S_{i=0}^{m} A(i)$

```fortran
SUBROUTINE SUM(A, B, map, m, comm, p)
    CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
    CALL MPI_WIN_CREATE(B, m* sizeofreal, sizeofreal, MPI_INFO_NULL, &
                         comm, win, ierr)
    CALL MPI_WIN_FENCE(0, win, ierr)
    DO i=1,m
        j = map(i)/p
        k = MOD(map(i),p)
        CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, &
                             MPI_SUM, win, ierr)
    END DO
    CALL MPI_WIN_FENCE(0, win, ierr)
    CALL MPI_WIN_FREE(win, ierr)
RETURN END
```

Extended Collectives

- Collective communications with inter communicators
- New routines for creating inter communicators
- 2 new collective routines – generalized all-to-all, exclusive scan
- Specification of “in place” buffers

Extensions

- Collective communications with inter communicators
- New routines for creating inter communicators
- 2 new collective routines – generalized all-to-all, exclusive scan
- Specification of “in place” buffers
Intercommunicator construction

- Inter communicator versions of MPI_COMM_GROUP and MPI_SPLIT_GROUP
- MPI_COMM_CREATE(comm_in, group, comm_out)
- MPI_COMM_SPLIT(comm_in, color, key, comm_out)

Intercommunicator Collectives

- Result of Collective operations on one group is seen on the other group
- Applies to following:
  1. MPI_BCAST
  2. MPI_GATHER, MPI_GATHERV
  3. MPI_SCATTER, MPI_SCATTERV
  4. MPI_ALLGATHER, MPI_ALLGATHERV
  5. MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
  6. MPI_REDUCE, MPI_ALLREDUCE
  7. MPI_REDUCE_SCATTER
  8. MPI_BARRIER.

Illustration – Intercommunicator all-gather

IN_PLACE & Intercommunicator collectives

- IN_PLACE extension for intra communicators to optimize memory movement
- Uni-directional transfer with root – MPI_ROOT, MPI_PROC_NULL and root_value
- Bi-directional transfer – Alltoall and allgather variants
External Interfaces

Generalized Requests

- Helps user to create his own non-blocking functions.
- E.g.: int myreduce(MPI_Comm comm, int tag, int root, int valin, int *valout, MPI_Request *request)

Naming Objects, Error classes, Thread interfaces

- For error reporting
- MPI_COMM_SET_NAME (comm, comm_name)
- MPI_TYPE_SET_NAME (type, type_name)
- MPI_WIN_GET_NAME (win, win_name, resultlen)
- Functions to add error codes and classes to the MPI library
- Definite rules for multi-threaded MPI processes; functions for initializing and querying thread environments
- Enhanced attribute caching functions

Parallel I/O
Motivation

- High level parallel I/O interface
- Supports file partitioning among processes
- Transfer of data structures between process memories and files
- Asynchronous/non-blocking I/O
- Strided / Non-contiguous access
- Collective I/O

Definitions

- Displacement – file position from the beginning of a file
- etype – unit of data access
- filetype – template for accessing the file
- view – current set of data accessible by a process. Repetitions of filetype pattern define a view
- offset – position relative to the current view

Examples

• MPI_FILE_OPEN(comm, filename, amode, info, fh)
• MPI_FILE_CLOSE(fh)
• MPI_FILE_DELETE(filename, info)
File Info Object

- Can provide hints to the implementation to improve I/O performance
- MPI_FILE_SET_INFO(fh, info)
- MPI_FILE_GET_INFO(fh, info_used)
- Default hints for file access patterns and data layout

File View

- MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
- MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

Default Hints (Info values)

- {access_style} (comma separated list of strings)
  - read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random
- {collective_buffering} (boolean)
- {cb_buffer_size} (integer)
- {cb_nodes} (integer)
- {filename} (string)
- {file_perm} (string)
- {io_node_list} (comma separated list of strings)
- {nb_proc} (integer)
- {num_io_nodes} (integer)
- {striping_factor} (integer)
- {striping_unit} (integer)

Data access routines

- 3 aspects – positioning, synchronism, coordination
- Positioning – explicit file offsets, individual file pointers, shared file pointers
- Synchronism – blocking, non-blocking/split-collective
- Coordination – non-collective, collective
### API

<table>
<thead>
<tr>
<th>Positioning</th>
<th>Synchronization</th>
<th>Coordination</th>
<th>Coordination Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit offsets</td>
<td>Blocking</td>
<td>MPI_FILE_READ</td>
<td>MPI_FILE_READ_AT_ALL</td>
</tr>
<tr>
<td></td>
<td>Non-blocking</td>
<td>MPI_FILE_READ_AT</td>
<td>MPI_FILE_READ_AT_ALL_BEGIN</td>
</tr>
<tr>
<td>Individual file pointers</td>
<td>Blocking</td>
<td>MPI_FILE_READ</td>
<td>MPI_FILE_READ_AT_ALL</td>
</tr>
<tr>
<td></td>
<td>Non-blocking</td>
<td>MPI_FILE_READ_AT</td>
<td>MPI_FILE_READ_AT_ALL_BEGIN</td>
</tr>
<tr>
<td>Shared file pointers</td>
<td>Blocking</td>
<td>MPI_FILE_READ_SHARED</td>
<td>MPI_FILE_READ_SHARED_AT_ALL</td>
</tr>
<tr>
<td></td>
<td>Non-blocking</td>
<td>MPI_FILE_READ_SHARED_AT</td>
<td>MPI_FILE_READ_SHARED_AT_ALL_BEGIN</td>
</tr>
</tbody>
</table>

### Simple example – Non-blocking read

```fortran
integer bufsize, req1, req2
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
parameter (bufsize = 10)
real buf1(bufsize), buf2(bufsize)

call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                  MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
                       MPI_INFO_NULL, ierr )
call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
                    req1, ierr )
call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
                    req2, ierr )
call MPI_WAIT( req1, status1, ierr )
call MPI_WAIT( req2, status2, ierr )
call MPI_FILE_CLOSE( myfh, ierr )
```

### Simple Example

```fortran
call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                   MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
                        MPI_INFO_NULL, ierr )
totprocessed = 0
do
    call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
                       status, ierr )
call process_input(localbuffer, status, ierr)
call MPI_FILE_CLOSE(myfh, ierr)
enddo
write(*,1001) numread, bufsize, totprocessed 1001 format( "No more data: read", I3, "and expected", I3, & "Processed total of", I6, "before terminating job." )
call MPI_FILE_CLOSE(myfh, ierr)
```

### Shared file pointers

- Can be used when all processes have the same file view
- Ordering is serialized during collective usage
- Ordering is non-deterministic for non-collective usage
File interoperability

- Accessing correct data information within and outside MPI environment in homogeneous and/or heterogeneous system
- Facilitated by data representations
- 3 types –
  native – only for homogeneous
  internal – for homogeneous or heterogeneous, implementation defined
  external32 – for heterogeneous, MPI provided data representation, very generic

Data Types

- When memory and file data layout are different, displacements have to be specified in terms of file data layout
- MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent) can be used
- external32 – predefined rules for different data types
- Also supports user-defined data representations by MPI_REGISTER_DATAREP

Helper functions

- int MPI_Type_create_darray(int size, int rank, int ndims, int *array_of_gsizes, int *array_of_distribs, int *array_of_dargs, int *array_of_psizes, int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

  **Input Parameters**
  - size: size of process group (positive integer)
  - rank: rank in process group (nonnegative integer)
  - ndims: number of array dimensions as well as process grid dimensions (positive integer)
  - array_of_gsizes: number of elements of type oldtype in each dimension of global array (array of positive integers)
  - array_of_distribs: distribution of array in each dimension (array of states)
  - array_of_dargs: distribution argument in each dimension (array of positive integers)
  - array_of_psizes: size of process grid in each dimension (array of positive integers)
  - order: array storage order flag (state)
  - oldtype: old datatype (handle)

  **Output Parameters**
  - newtype: new datatype (handle)

- int MPI_Type_create_subarray(int ndims, int *array_of_sizes, int *array_of_subsizes, int *array_of_starts, int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

  **Input Parameters**
  - ndims: number of array dimensions (positive integer)
  - array_of_sizes: number of elements of type oldtype in each dimension of the full array (array of positive integers)
  - array_of_subsizes: number of elements of type oldtype in each dimension of the subarray (array of positive integers)
  - array_of_starts: starting coordinates of the subarray in each dimension (array of nonnegative integers)
  - order: array storage order flag (state)
  - oldtype: old datatype (handle)

  **Output Parameters**
  - newtype: new datatype (handle)
Example 4 - filetype creation

double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &filetype);

Example 5 - noncontiguous access with a single collective I/O function

int main(int argc, char ** argv )
{
  int *buf, rank, nprocs , nints , bufsize ;
  MPI_File fh;
  MPI_Datatype filetype;
  ...
  bufsize = FILESIZE/nprocs;
  buf = (int *) malloc(bufsize );
  nints = bufsize/sizeof(int );
  ...
  MPI_File_open (MPI_COMM_WORLD, “/ pfs/datafile “, MPI_MODE_RDONLY, MPI_INFO_NULL, & fh);
  MPI_Type_vector (nints/INTS_PER_BLK, INTS_PER_BLK, INTS_PER_BLK* nprocs , MPI_INT, & filetype );
  MPI_Type_commit (&filetype );
  MPI_File_set_view (fh, INTS_PER_BLK* sizeof(int )*rank, MPI_INT, filetype , “native”, MPI_INFO_NULL);
  MPI_File_read_all (fh, buf, nints , MPI_INT, MPI_STATUS_IGNORE);
  ...
}

Example 6 – writing distributed array

int main(int argc, char *argv[] )
{
  int gsizes[2], distribs[2], dargs[2], psizes[2], rank, size, m, n;
  /* This code is particular to a 2 x 3 process decomposition */
  col_procs = 2;
  num_local_rows = …;
  local_array = (float *)malloc( … );
  /* ... set elements of local array ... */
  MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs, psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
  MPI_Type_commit(&filetype);
  MPI_File_open(MPI_COMM_WORLD, “/pfs/datafile”, MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, “native”, MPI_INFO_NULL);
  local_array_size= num_local_rows* num_local_cols;
  MPI_File_write_all(fh, local_array , local_array_size, MPI_FLOAT, &status);
  ...
}

Example 7 – writing distributed array with subarrays

int main(int argc, char *argv[] )
{
  int gsizes[2], distribs[2], dargs[2], psizes[2], rank, size, m, n;
  /* This code is particular to a 2 x 3 process decomposition */
  col_procs = 2;
  num_local_rows = …;
  num_local_cols = …;
  local_array = (float *)malloc( … );
  /* ... set elements of local array ... */
  gsizes[0] = m; gsizes[1] = n;
  psizes[0] = row_procs ; psizes[1] = col_procs ;
  lsizes[0] = m/psizes[0]; lsizes[1] = n/psizes[1];
  ...
  MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
  MPI_Comm_rank (comm, &rank);
  ...
  MPI_Type_create_subarray (2, gsizes , lsizes, start_indices , MPI_ORDER_C, MPI_FLOAT, &filetype );
  MPI_Type_commit (&filetype);
  MPI_File_open (MPI_COMM_WORLD, “/pfs/datafile”, MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
  ...
}