MPI-2

Sathish Vadhiyar

- http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm
- Using MPI2: Advanced Features of the Message-Passing Interface. http://www-unix.mcs.anl.gov/mpi/usingmpi2/

MPI-2

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

Parallel I/O

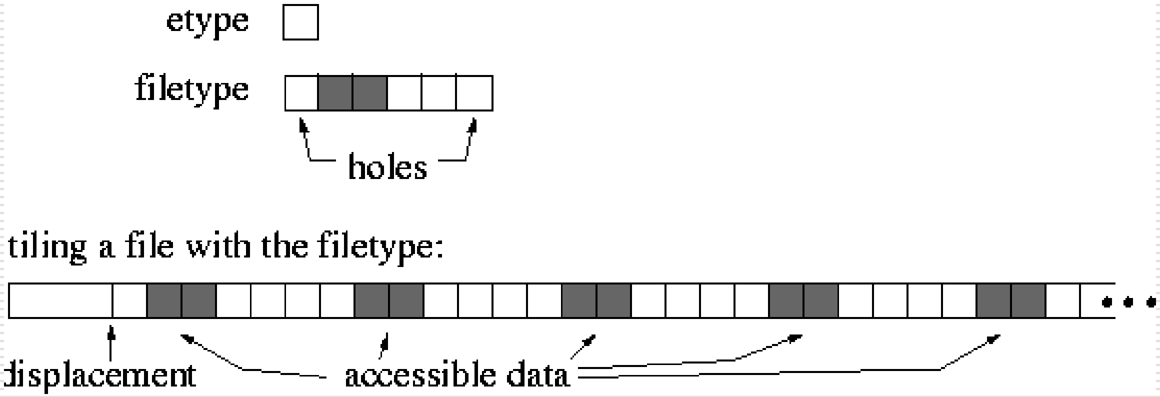
Motivation

- High level parallel I/O interface
- Supports file partitioning among processes
- Transfer of data structures between process memories and files
- Also supports
  - 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. Repetition of filetype pattern define a view
- offset – position relative to the current view

Examples


etype
filetype
holes
tiling a file with the filetype:
displacement
accessible data


etype
process 0 filetype
process 1 filetype
process 2 filetype
tiling a file with the filetypes:
displacement

File Manipulation

- MPI_FILE_OPEN(comm, filename, amode, info, fh)
- MPI_FILE_CLOSE(fh)

File View

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

- e.g.: if a file has double elements and if etype = filetype = MPI_REAL, then a process wants to read all elements

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

Positioning	**Synchronism**	**Coordination Non-Collective**	**Coordination Collective**
Explicit offsets	Blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
	Non-blocking	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END
Individual file pointers	Blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
	Non-blocking	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END
Shared file pointers	Blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
	Non-blocking	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END

Simple Example

```
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 MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
  call process_input( localbuffer, numread )
  totprocessed = totprocessed + numread
  if ( numread < bufsize ) exit
enddo

write(6,1001) 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 )
```

Simple example – Non-blocking read

```
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 )
```

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

File Consistency

- MPI_FILE_SET_ATOMICITY(fh, flag)
- MPI_FILE_GET_ATOMICITY(fh, flag)
- MPI_FILE_SYNC(fh)

Consistency Examples
Example 1 – atomic access

```
/* Process 0 */
int i, a[10] ; int TRUE = 1;

for ( i=0;i<10;i++) a[i] = 5 ;
MPI_File_open( MPI_COMM_WORLD, "workfile", MPI_MODE_RDWR |
      MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh0, TRUE ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status) ;

/* Process 1 */
int b[10] ; int TRUE = 1;
MPI_File_open( MPI_COMM_WORLD, "workfile", MPI_MODE_RDWR |
      MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh1, TRUE ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status) ;
```

Example 2 – Consistency with Barrier and SYNC

```
/* Process 0 */
int i, a[10] ;

for ( i=0;i<10;i++) a[i] = 5 ;
MPI_File_open( MPI_COMM_WORLD, "workfile", MPI_MODE_RDWR | MPI_MODE_CREATE,
        MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh0 ) ;

/* Process 1 */
int b[10] ;

MPI_File_open( MPI_COMM_WORLD, "workfile", MPI_MODE_RDWR | MPI_MODE_CREATE,
        MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_sync( fh1 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
```

Example 3 – Double buffering with split collective I/O

```
/* this macro switches which buffer "x" is pointing to */
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))

void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount) {
MPI_Status status;
float *buffer1, *buffer2;
float *compute_buf_ptr;
float *write_buf_ptr;

  compute_buf_ptr = buffer1 ;
  write_buf_ptr = buffer1 ;

  compute_buffer(compute_buf_ptr, bufcount, &done);
  MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);

  while (!done) {
    TOGGLE_PTR(compute_buf_ptr);
    compute_buffer(compute_buf_ptr, bufcount, &done);
    MPI_File_write_all_end(fh, write_buf_ptr, &status);
    TOGGLE_PTR(write_buf_ptr);
    MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
  }

  MPI_File_write_all_end(fh, write_buf_ptr, &status);
}
```

Template Creation Helper functions (User Defined Data Types)

int MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Input Parameters

count number of blocks (nonnegative integer)

blocklength number of elements in each block (nonnegative integer)

stride number of elements between start of each block (integer)

oldtype old datatype (handle)

Output Parameters

newtype

new datatype (handle)

Example 4 - 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);
  …
}
```

Helper functions

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)

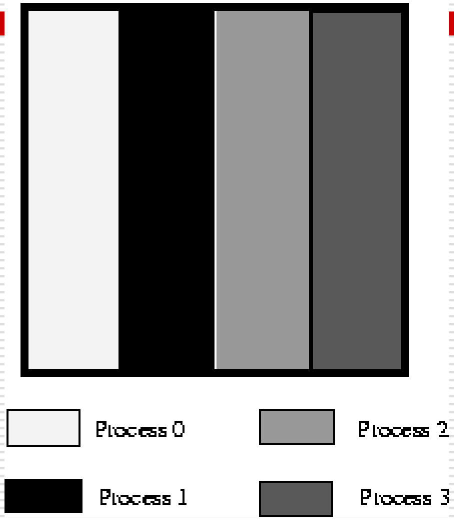
Input Parameters

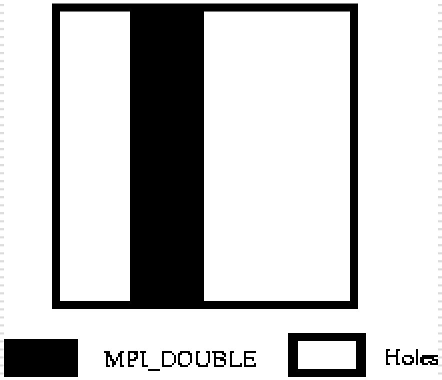
ndims number of array dimensions (positive integer)
array_of_sizes number of elements of type oldtype in each dimension of the full array (array of positive integers)
array_of_subsizes number of elements of type newtype in each dimension of the subarray (array of positive integers)
array_of_starts starting 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 5 - filetype creation





```
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 6 – writing distributed array with subarrays

```
/* This code is particular to a 2 x 3 process decomposition */
row_procs = 2;
col_procs = 3;

gsizes[0] = m; gsizes[1] = n;
psizes[0] = row_procs; psizes[1] = col_procs;
lsizes[0] = m/psizes[0]; lsizes[1] = n/psizes[1];
dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;

 MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
 MPI_Comm_rank (comm, &rank);
 MPI_Cart_coords (comm, rank, 2, coords);

/* global indices of the first element of the local array */
start_indices[0] = coords[0] * lsizes[0]; start_indices[1] = coords[1] * lsizes[1];
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);
MPI_File_set_view (fh, 0, MPI_FLOAT, filetype, "native", MPI_INFO_NULL);
local_array_size = lsizes[0] * lsizes[1];
MPI_File_write_all (fh, local_array, local_array_size, MPI_FLOAT, &status);
…
```

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 state)
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)

Example 7 – 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 */
  row_procs = 2;
  col_procs = 3;
  num_local_rows = …;
  num_local_cols = …;
  local_array = (float *)malloc( … );
  /* ... set elements of local_array ... */

  gsizes[0] = m; gsizes[1] = n;
  distribs[0] = MPI_DISTRIBUTE_BLOCK; distribs[1] = MPI_DISTRIBUTE_BLOCK;
  dargs[0] = MPI_DISTRIBUTE_DFLT_DARG; dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
  psizes[0] = row_procs; psizes[1] = col_procs;

  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)
   ...
}
```