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

- …….
- One sided communications
- Parallel I/O
- ………

Parallel I/O

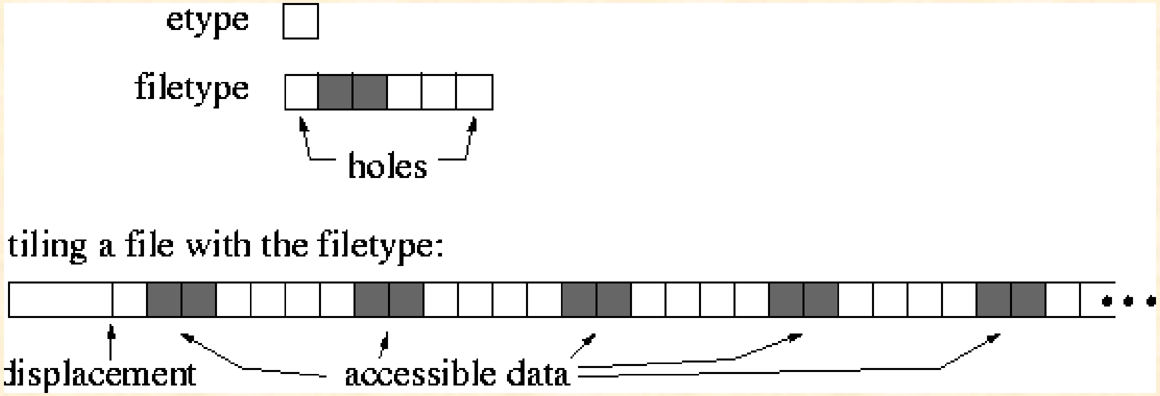
Motivation

- 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

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

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

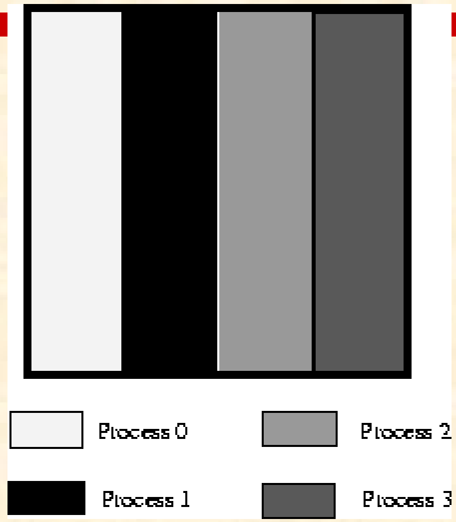
ndims number of array dimensions (positive integer)
array_of_sizes number of elements of type oldtype in each dimension of the full array (array of positive integers)
array_of_subsizes number of elements of type newtype in each dimension of the subarray (array of positive integers)
array_of_starts starting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

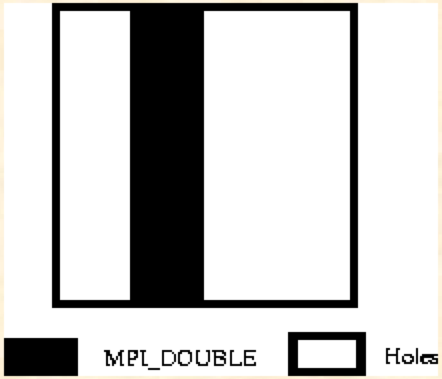
Output Parameters

newtype

new datatype (handle)

Example: 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: 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);
...
```

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

- MPI provides controls
  - Which parts of memory can be accessed by remote memory
  - During what time (synchronization – more later)
- Which parts of memory? – MPI helps creates window of memory access
- MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)

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(map)

```
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 $B(j) = \Sigma_{map(i)=j} A(i)$

```
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
```

Synchronization

- Active target communication - Both processes are explicitly involved in communication
- Passive target communication - Only origin process is involved
- Access epoch - Contains RMA calls in the origin. Starts and ends with synchronization calls.
- Exposure epoch – contains RMA calls in the active target

Synchronization

- 3 synchronization mechanisms:

  - MPI_WIN_FENCE (at origin and target) (for active target)

  - MPI_WIN_START, MPI_WIN_COMPLETE (origin)

  MPI_WIN_POST, MPI_WIN_WAIT (target) (for active target)

  - MPI_WIN_LOCK, MPI_WIN_UNLOCK (only at origin) (passive target)

Active synchronization