# MPI Parallel I/O

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Materials are taken from the book, Using MPI-2: Advanced Features of the Message-Passing Interface by William Gropp, Ewing Lusk, and Rajeev Thakur.

# Topic Overview

- Non-Parallel I/O
- Using MPI for Simple I/O
- Noncontiguous Accesses and Collective I/O
- Accessing Arrays Stored in Files
- Achieving High I/O Performance with MPI

# Non-Parallel I/O

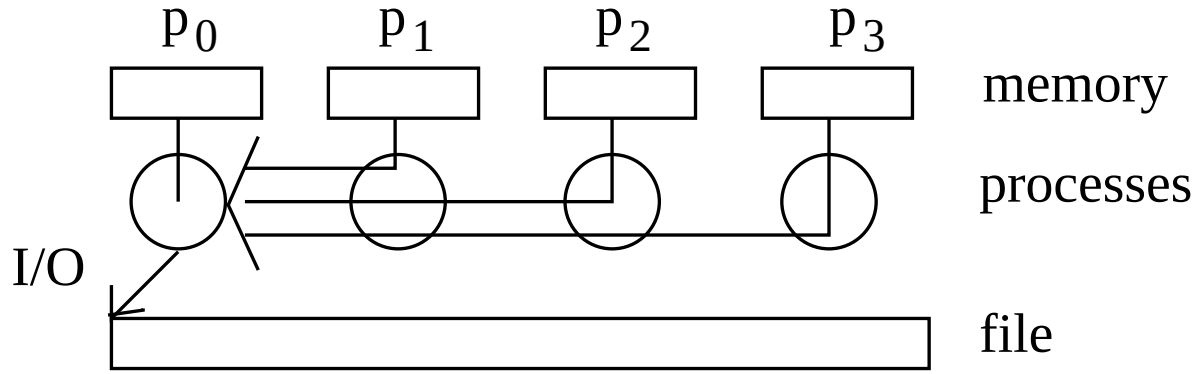
- MPI-1 does not have any explicit support for parallel I/O.

- The reasons to do this

  1. The system may support I/O only from ONLY one process.
  2. Resulting to a single file, and easy to handle outside the program.

- The reasons NOT to do this

  1. Pains in the neck.
  2. Limit performance and scalability.



# Non-Parallel I/O

- Use traditional Unix I/O from single processor.
- Pass the data to "processor 0" before I/O.

```
if (myrank != 0)
    MPI_Send(buf,BUFSIZE,MPI_INT, 0, 99, MPI_COMM_WORLD);
else{
    myfile=fopen("testfile", "w");
    fwrite(buf, sizeof (int), BUFSIZE, myfile);
    for (i=1; i<numprocs; i++){
      MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99, MPI_COMM_WORLD,
               &status);
      fwrite(buf,sizeof (int), BUFSIZE, myfile);
    }
    fclose(myfile);
}
```

- Write a mpi code to do the non-parallel I/O of a linear file.

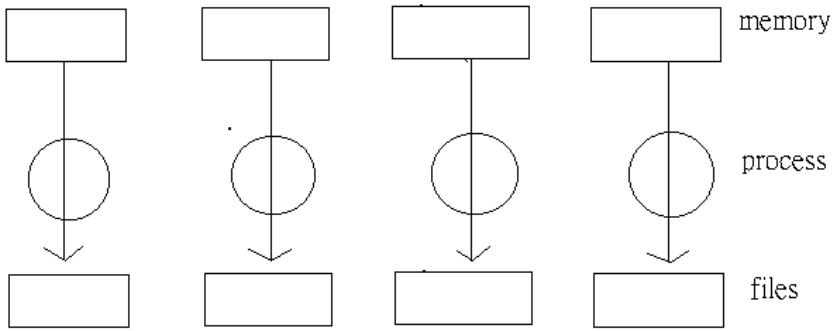
# Non-MPI Parallel I/O from a MPI Code

- Advantage

  1. Parallel I/O.

- Disadvantages

  1. Files need to be joined before further usage.
  2. Need to use fixed number of processes.
  3. Difficult to handle.



# Non-MPI Parallel I/O from a MPI Code

- Each processor writes to itself own file.

```
sprintf (filename, "testfile.%d", myrank);
myfile = fopen(filename, "w");

for(j = 0; j < BUFSIZE; j++){
  fprintf(myfile, "%d ",buf[j]);
}
fprintf(myfile,"\n");

fclose (myfile);
```

- Output files are: testfile.0, testfile.1, testfile.2, etc.

# MPI Parallel I/O to Separated Files

- Similar to Non-MPI parallel I/O

```
int MPI_File_open(MPI_Comm comm, char *filename, int
                  file_access_mode, MPI_Info info, MPI_File *fh)
int MPI_File_write(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status)
int MPI_File_close(MPI_File *fh);
```

---

```
sprintf (filename, "testfile.%d", myrank);
MPI_File_open(MPI_COMM_SELF, filename,
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              MPI_INFO_NULL, &myfile);
MPI_File_write (myfile, buf, BUFSIZE, MPI_INT,
                &status);
MPI_File_close(&myfile);
```

---

- `*fh`: File handle.
- `MPI_COMM_SELF`: For the process exclusively.
- `MPI_MODE_WRONLY`: Write/Read ONLY.

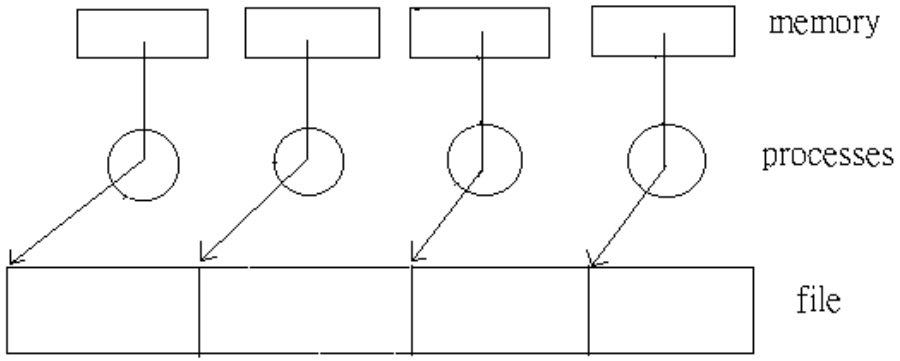
# MPI Parallel I/O to Single File

- True parallel I/O to a single file.
- New in mpi2.

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
            MPI_Datatype etype, MPI_Datatype filetype,
            char *datarep, MPI_Info info)
```

---

```
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                  MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT, &status);
```



# MPI Parallel I/O to Single File

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype, MPI_Status *status)
```

---

```
MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_RDONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_get_size (thefile, &filesize); /* in bytes */
filesize = filesize/sizeof(int);     /* in number of ints */
bufsize = filesize/numprocs ;     /* local number to read */
buf = (int *) malloc (bufsize * sizeof (int));
MPI_File_set_view(thefile, myrank * bufsize * sizeof (int),
                  MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_read(thefile, buf, bufsize, MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
```

---

- `disp`: displacement (nonnegative integer).
- `*datarep`: data representation (string). (native: as in memory; internal; external 32) which enable various degrees of file portability across machines.

# MPI Parallel I/O

- *Individual-file-pointer functions:* Use the current location of the individual file pointer of each process to read/write data.
- Examples: MPI_File_read and MPI_File_write
- There are other I/O functions in MPI which are for performance, portability, and convenience.
- *Explicit-offset functions:* Not to use the individual file pointer. Rather, the file offset is passed directly as an argument to the function. A separate seek (or MPI_File_set_view) is therefore not needed.
- Examples: MPI_File_read_at and MPI_File_write_at.

---

A file pointer is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.

# MPI Parallel I/O Explicit-offset Functions

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf,
          int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf,
          int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

---

```
MPI_File_get_size (fh, &filesize); /* in bytes */
filesize = filesize/sizeof(int);   /* in number of ints */
bufsize = filesize/nprocs;
buf = (int *) malloc(bufsize* sizeof (int));
nints = bufsize;

MPI_File_read_at(fh, myrank*bufsize* sizeof (int), buf,
                 nints, MPI_INT, &status);
```

V.S.

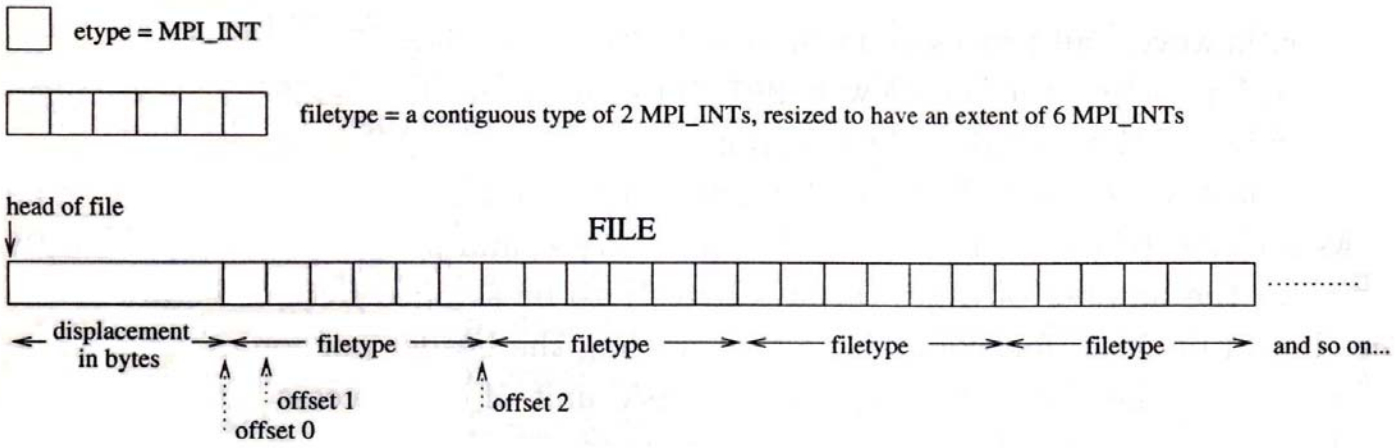
```
MPI_File_seek(fh, rank*bufsize* sizeof (int), MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
```

---

- `whence`: update mode: "MPI_SEEK_SET". The pointer is set to offset.

# Non-contiguous Accesses

- A file view in MPI defines which portion of a file is visible to a process.
- A read or write function can access data only from the visible portion of the file.
- Reasons that we want to change the file view:
  1. Indicate the type of data that the process is going to access, rather than just bytes.
  2. Indicate which parts of the file should be skipped, i.e. to specify noncontiguous access in the file.



# Non-contiguous Accesses

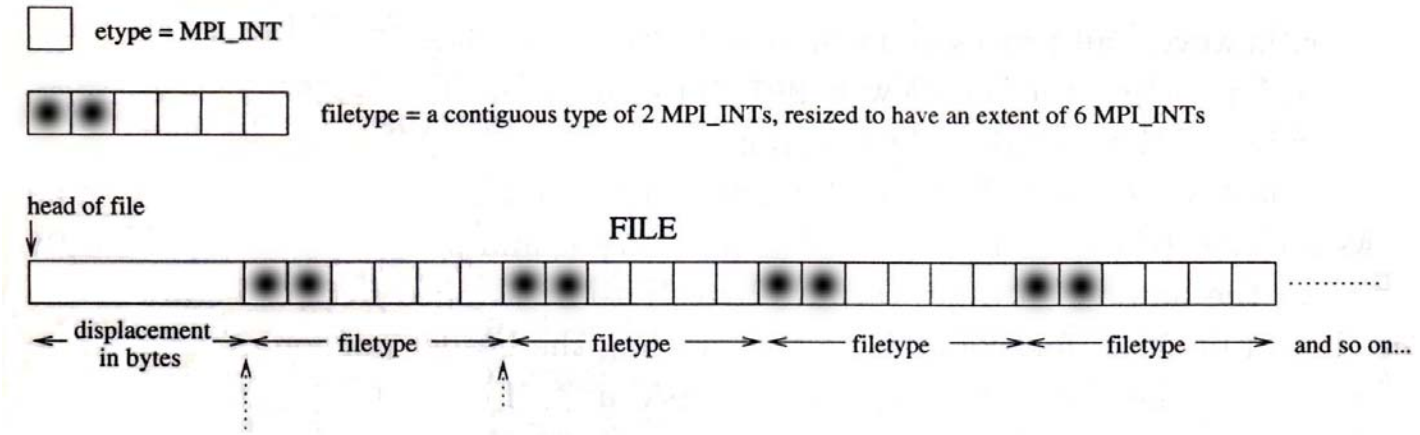
```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
                        MPI_Aint extent, MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype filetype,
                      char *datarep, MPI_Info info)
```

---

- `MPI_Aint`: C type that holds any valid address.
- `lb`: New lower bound of datatype.
- `extend`: New extent of datatype.
- `etype`: Basic unit of data access.
- `filetype`: Specify which portion of the file is visible to the process and of what type is the data.

# Non-contiguous Accesses

```
MPI_Type_contiguous(2, MPI_INT, &contig);
lb     = 0;
extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp   = 5 * sizeof(int);/* assume displacement in this file view
                            is of size equal to 5 integers */
nint = bufsize;
etype = MPI_INT;
MPI_File_set_view(fh, disp+myrank*nint/2*extent, etype,
                  filetype, "native", MPI_INFO_NULL);
```



# Collective I/O

- Collective means that the function must be called by every process in the communicator.
- This communicator information is implicitly contained in the file handle passed to MPI_File_read_all.
- When a process calls an independent I/O function, the implementation has no idea what other processes might do and must therefore satisfy the request of each process individually.
- When a process calls a collective I/O function, however, the implementation knows exactly which other processes will also call the same collective I/O function.
- Therefore, the user should, when possible, use the collective I/O instead of independent I/O functions.

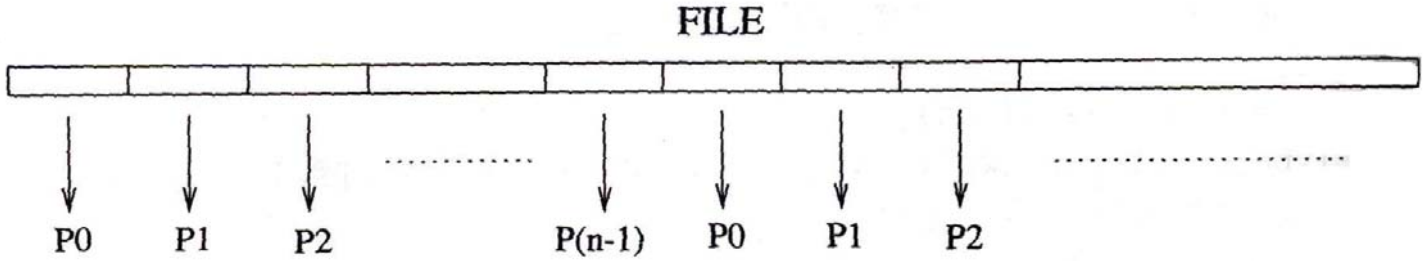
# Collective I/O

- Collective I/O with noncontiguous accesses.
- Each process reads smaller blocks of data distributed in a block-cyclic manner in the file.

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                      MPI_Datatype datatype, MPI_Status *status)
```

---

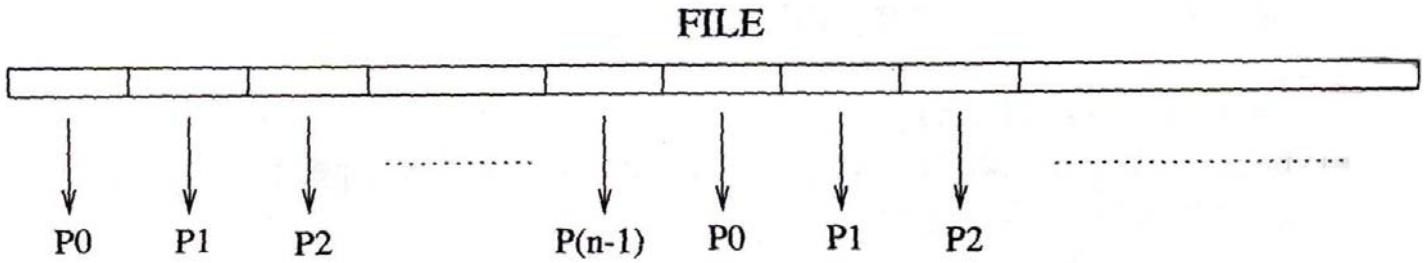
- `count`: Number of blocks.
- `blocklength`: Number of elements in each block.
- `stride`: Number of elements between start of each block.



# Collective I/O

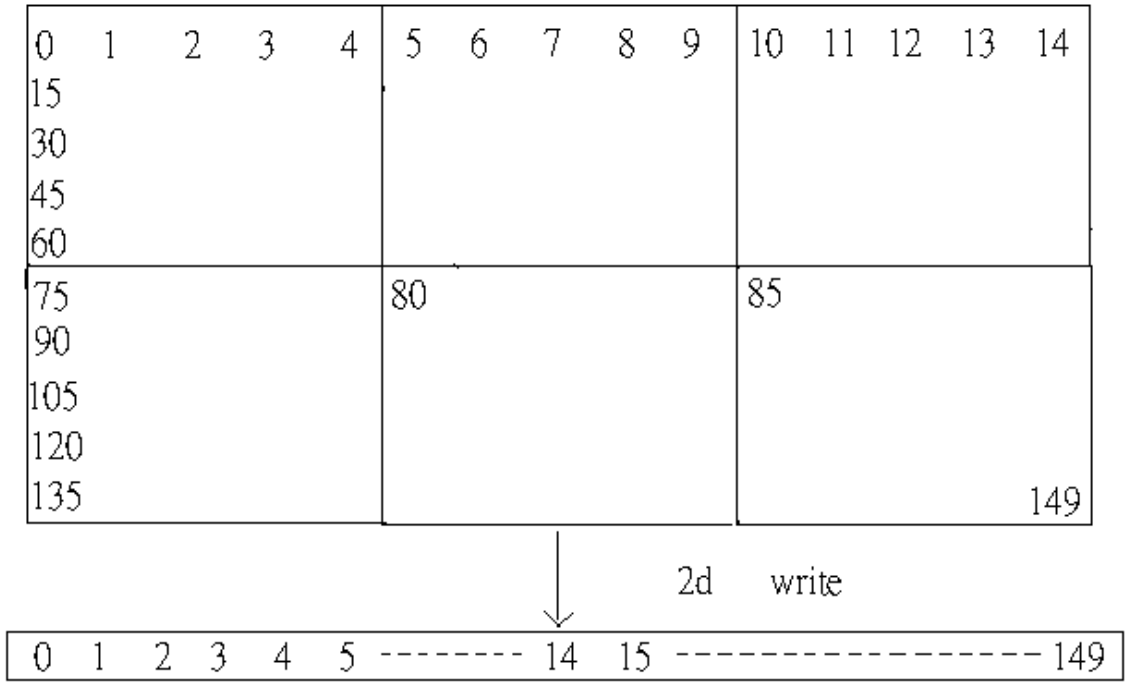
```
#define INTS_PER_BLK  5
    ........
    MPI_File_get_size (fh, &filesize); /* in bytes */
    bufsize = filesize/nprocs;
    buf = (int *) malloc(bufsize);
    nints = bufsize/sizeof(int);

    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,&status);
```



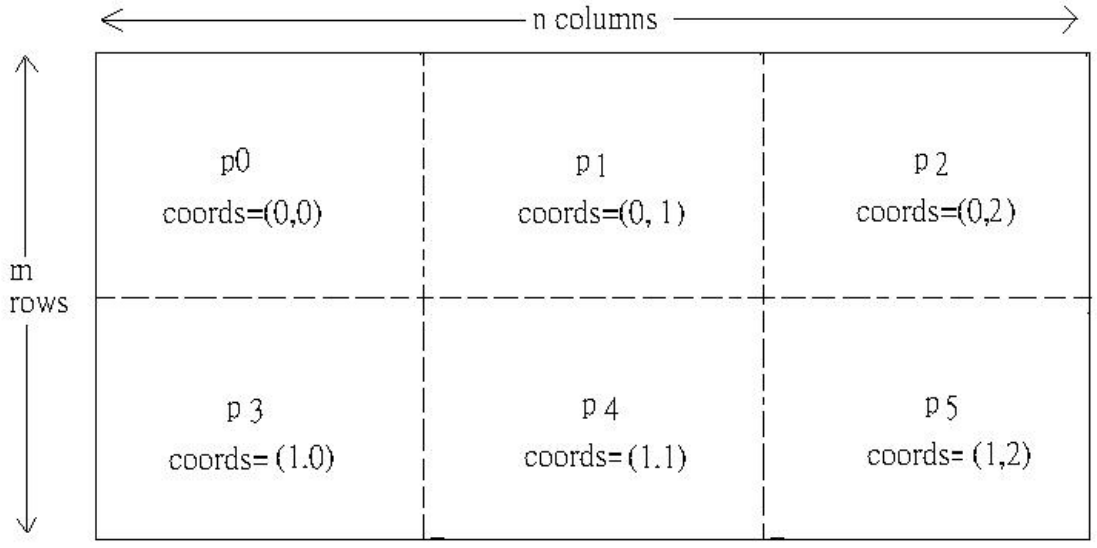
# Collective I/O

- Homework: Do a 2-d write. Is there a 2-D file stream ?
- Homework: Read in an image file to a $2 \times 3$ 2-d communicator.
- xxx_read/write access binary files; fprintf/fscanf access ASCII files.
- The local array of each process is not located contiguously in the file.



# Accessing Arrays Stores in Files

- Access subarrays and distributed arrays (both regularly and irregularly distributed) stored in files.
- Parallel program often utilize multidimensional arrays distributed among processes.
- Arrays must be read from or written to a file in which the storage order corresponds to that of global array.
- MPI provides high performance collective I/O for this kind of access, even though the accesses are noncontiguous.




2D array distributed on a 2x3 process grid

# Accessing Arrays Stores in Files–Distributed Arrays

- MPI-2 defines new datatype constructors, *darray* and *subarray*.
- Faciliate the creation of derived datatypes describing the location of a local array within a linearized global array.

```
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)
```

---

- `size`: The total number of processors over which the array is distributed.
- `rank`: The rank in process group.
- `ndims`: Number of dimensions of the global array.
- `array_of_gsizes[]`: The size of the global array in each dimension.
- `array_of_distribs[]`: Specify the way in which the global array is distributed in each dimension. Use MPI_DISTRIBUTE_BLOCK.

# Accessing Arrays Stores in Files–Distributed Arrays

```
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)
```

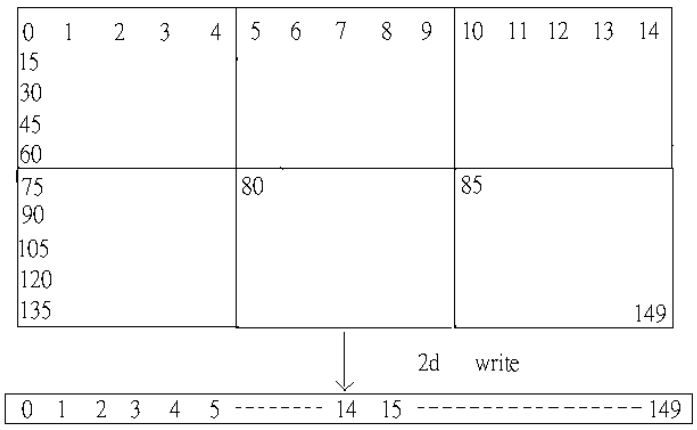
- Homework: Use MPI_Type_darray to read in an image file.

---

- `array_of_drags[]`: Distribution argument in each dimension. For block and cyclic distributution, we dont need this parameter, MPI_DISTRIBUTE_DFLT_DARG is used.
- `array_of_psizes[]`: Size of process grid in each dimension.
- `order`: array storage order flag, (MPI_ORDER_C).
- MPI_Cart_create is needed to be called before MPI_Type_create_darray is called.

# Accessing Arrays Stores in Files–Distributed Arrays

```
gsizes[0] = 10;     /* no. of rows in global array */
gsizes[1] = 15;     /* no. of columns in global array*/
distribs[0] = MPI_DISTRIBUTE_BLOCK;  /* block distribution */
distribs[1] = MPI_DISTRIBUTE_BLOCK;  /* block distribution */
dargs[0] = MPI_DISTRIBUTE_DFLT_DARG; /* default block size */
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG; /* default block size */
psizes[0] = 2;  /* no. of processes in vertical dimension
                           of process grid */
psizes[1] = 3;  /* no. of processes in horizontal dimension
                           of process grid */
MPI_Type_create_darray(6, new2drank, 2, gsizes, distribs, dargs,
                       psizes, MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write_all(fh, buf, nint, MPI_INT, &status);
```



# Accessing Arrays Stores in Files–Subarray Arrays

```
int MPI_Type_create_subarray(int ndims, int array_of_gsizes[],
                    int array_of_subsizes[], int array_of_starts[],
                    int order, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

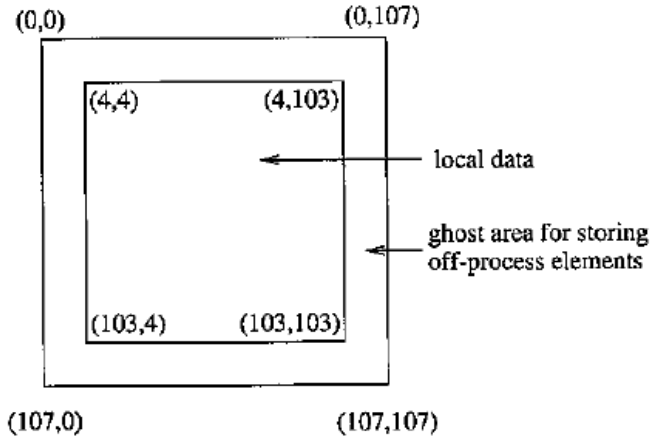
---

- `array_of_gsizes[]` : The size of the global array in each dimension.
- `array_of_subsizes[]` :The size of the local array in each dimension.
- `array_of_starts[]` : Starting coordinates of the subarray in global indices in each dimension.
- We could use MPI_Type_create_subarray to do what MPI_Type_create_darray does.

```
gsizes[0] = 10;  /* no. of rows in global array */
gsizes[1] = 15;  /* no. of columns in global array*/
lsizes[0] = gsizes[0]/psizes[0];/* no. of rows in local array */
lsizes[1] = gsizes[1]/psizes[1];/* no. of columns in local array*/
/* global indices of the first element of the local array */
start_indices[0] = my2dcoords[0] * lsizes[0];
start_indices[1] = my2dcoords[1] * lsizes[1];
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, MPI_INT, filetype, "native", MPI_INFO_NULL);
```

# Accessing Arrays Stores in Files–Subarray Arrays

- However, we need ghost region to be involved.
- Since the data are noncontiguous in memory layout, so we describe them in terms of an MPI derived datatype.
- Specify this derived datatype as the datatype argument to a single MPI_File_write_all function.
- The entire data transfer, which is noncontiguous in both memory and file, can therefore be performed with a single function.



# Accessing Arrays Stores in Files–Subarray Arrays

```
gsizes[0] = 10;  /* no. of rows in global array */
gsizes[1] = 15;  /* no. of columns in global array*/
lsizes[0] = gsizes[0]/psizes[0];/* no. of rows in local array */
lsizes[1] = gsizes[1]/psizes[1];/* no. of columns in local array*/
/* global indices of the first element of the local array */
start_indices[0] = my2dcoords[0] * lsizes[0];
start_indices[1] = my2dcoords[1] * lsizes[1];
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                    MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(&filetype);
```

---

```
MPI_File_set_view(fh, 0, MPI_INT, filetype, "native", MPI_INFO_NULL);
/* create a derived datatype that describes the layout of the local
   array in the memory buffer that includes the ghost area. This is
   another subarray datatype! */
   memsizes[0] = lsizes[0] + 2;/* no. of rows in allocated array */
   memsizes[1] = lsizes[1] + 2;/* no. of columns in allocated array*/
   start_indices[0] = start_indices[1] = 1;
/* indices of the first element of the local array
   in the allocated array */
   MPI_Type_create_subarray(2, memsizes, lsizes, start_indices,
                            MPI_ORDER_C, MPI_INT, &memtype);
   MPI_Type_commit(&memtype);
   MPI_File_write_all(fh, buf, 1, memtype, &status);
```

- Homework: Use MPI_Type_subarray to read in an image file.

# Accessing Arrays Stores in Files–An irregular distribution

- MPI can also be used for accessing irregularly disturbuted arrays, by specifying the filetype appropriately.
- An irregular distribution is one that cannot be expressed mathematically by a compact formula, unlike a block or cyclic distribution.
- Another array, called a map array, that specifies the mapping of each element of the local array to the global array is needed (array_of_displacements).

```
int MPI_Type_create_indexed_block(int count, int blocklength,
                int array_of_displacements[], MPI_Datatype oldtype,
                MPI_Datatype *newtype)
```

---

- `count`: The number of elements in local array.
- `blocklength`: The number of element in each block.
- `array_of_displacements`: Specify the displacement of each block in datatype.

# Accessing Arrays Stores in Files–An irregular distribution

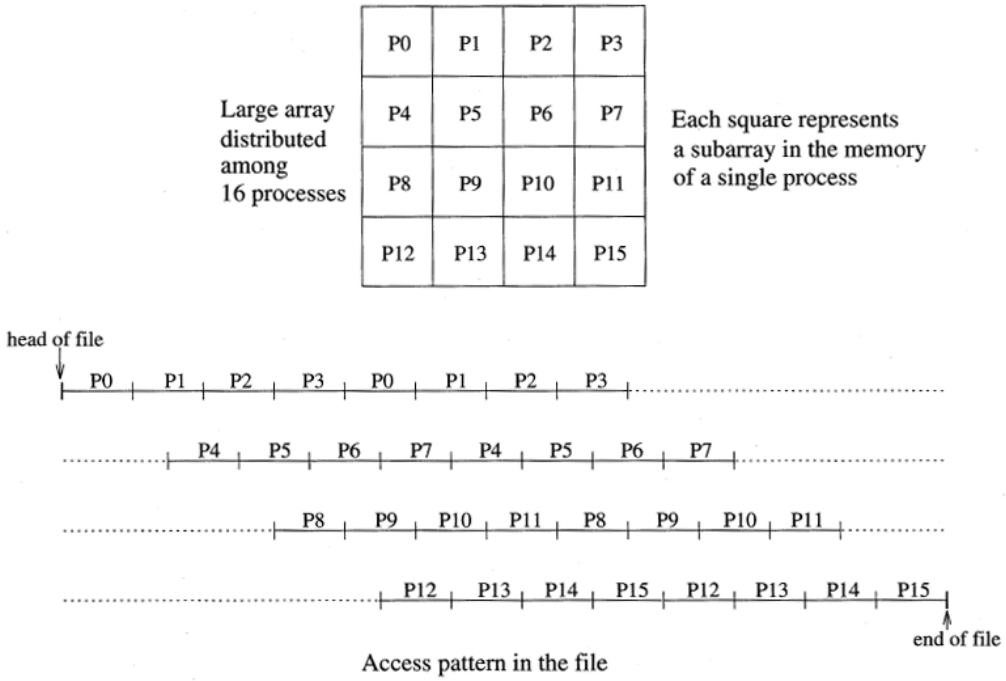
```
  if (myrank == 0){
      map[0] = 2; map[1] = 6; map[2] = 7; map[3] = 10; map[4] = 14;}
  else if (myrank == 1){
      map[0] = 0; map[1] = 1; map[2] = 3; map[3] = 4;  map[4] = 11;}
  else{
      map[0] = 5; map[1] = 8; map[2] = 9; map[3] = 12; map[4] = 13;}
    /* ... other application code ... */

  MPI_File_open(MPI_COMM_WORLD, "irregularfile",
                MPI_MODE_CREATE | MPI_MODE_WRONLY,
                MPI_INFO_NULL, &fh);
  MPI_Type_create_indexed_block(BUFSIZE,1,map,MPI_INT,&filetype);
  MPI_Type_commit(&filetype);

  MPI_File_set_view(fh, disp, etype, filetype, "native",
                    MPI_INFO_NULL);
  MPI_File_write_all(fh, buf, BUFSIZE, MPI_INT, &status);
```

```
Output:
     p0          |        p1         |       p2
0   1   2   3   4 |  5   6   7   8   9 | 10 11 12 13 14
5   6   0   7   8   10   1   2   11 12   3   9   13 14 4
```

# Achieving High I/O Performance with MPI



Access pattern in the file

- Homework: Compare the write-out time for 4 levels of I/O on a 8192x8192 array.

- Homework: Write a mpi code to analyze the the four "levels" of access v.s. data length.

# Achieving High I/O Performance with MPI

```
MPI_File_open(..., "filename", ..., &fh)
for (i=0; i<n_local_rows; i++) {
  MPI_File_seek(fh, ...)
  MPI_File_read(fh, row[i], ...)
}
MPI_File_close(&fh)
```

*Level 0*
*(many independent, contiguous requests)*

```
MPI_File_open(MPI_COMM_WORLD, "filename", ..., &fh)
for (i=0; i<n_local_rows; i++) {
  MPI_File_seek(fh, ...)
  MPI_File_read_all(fh, row[i], ...)
}
MPI_File_close(&fh)
```

*Level 1*
*(many collective, contiguous requests)*

```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(..., "filename", ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read(fh, local_array, ...)
MPI_File_close(&fh)
```

*Level 2*
*(single independent, noncontiguous request)*

```
MPI_Type_create_subarray(.., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(MPI_COMM_WORLD, "filename", ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read_all(fh, local_array, ...)
MPI_File_close(&fh)
```

*Level 3*
*(single collective, noncontiguous request)*

**Figure**
Pseudo-code that shows four ways of accessing the data in Figure 3.22 with MPI

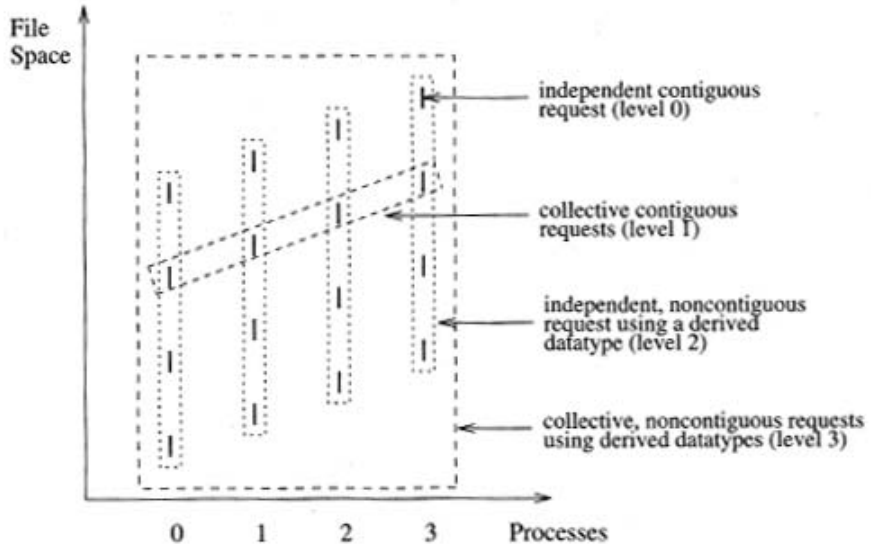



**Figure**
The four levels representing increasing amounts of data per request

# Achieving High I/O Performance with MPI

- In level 0, each process does Unix-style accesses.
- In level 2, each process creates a derived datatype to describe the noncontiguous access pattern.
- In level 1, 3, collective I/O functions are called.
- If an application needs to access only large, contiguous pieces of data, level 0 is equivalent to level 2, and level 1 is equivalent to level 3.
- The optimizations typically allow the physical I/O to take place, contiguous chunks, with higher performance.
- The more the amount of data per request, the greater is the opportunity for the implementation to deliver higher performance.
- Compute the congested ts, tm for npes = 4, 8, 12.

# Matrix Vector Multiplication

Compressed row storage.

A=

| | | | | | |
|---|---|---|---|---|---|
| 2.8 | | 5.3 | 4.3 | | |
| | 1.2 | 3.0 | | | |
| 1.0 | | | 0.4 | 8.9 | |

rowptr :

| | | | |
|---|---|---|---|
| 0 | 3 | 5 | 8 |

colind :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 1 | 2 | 0 | 3 | 4 |

values :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2.8 | 5.3 | 4.3 | 1.2 | 3.0 | 1.0 | 0.4 | 8.9 |

```
SparseMatVec(nlocal, rowptr, colind, values, b, y, MPI_COMM_WORLD);
```

Homework: Use "SparseMatVec" to implement Conjugate Gradient method.

---

- `nlocal`: $n/p$.
- `A b = y`