MPI-2 I/O

Introduction

We have discussed I/O in the previous lecture, and even developed our own functions for that purpose

Those functions, though useful, have significant limitations. In particular, they generally all use only one selected process to access data.

This clearly will not work efficiently when the data to be accessed is large.

MPI 2.0 - a set of extensions to standard MPI, provides a number of functions for parallel I/O

It offers other mechanisms which will not be discussed here

MPI 2.0 functions are now commonly available in most implementations of MPI

What is parallel I/O?

Multiple processes of parallel program accessing data (reading or writing) from a common file

This is in contrast to non-parallel I/O which is simple but:
- has poor performance in case a selected single process does all data access
- is awkward in the other possible case where each process writes to its own separate file

Why is MPI a good framework for parallel I/O?

Can think that

- writing to file is like sending a message
- reading from file is like receiving a message

We can thus use a lot of the concepts we have encountered already.

Examples:

- non-blocking I/O
- have a process read from the right location in the file by displacing the file pointer to the right location
- non-contiguous data

You can usually assume that the MPI implementation you are working with will be set up in such a way that parallel I/O operations will be possible as far as the hardware permits

The examples that follow will show various examples of I/O, from non-parallel we have already encountered to parallel

Code examples from
http://www.mcs.anl.gov/research/projects/mpi/usingmpi2/examples/main.htm

```
/* example of sequential Unix write into a common file */
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[]) {
    int i, myrank, numprocs, buf[BUFSIZE];
    MPI_Status status;
    FILE *myfile;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (i=0; i<BUFSIZE; i++)
   buf[i] = myrank * BUFSIZE + i;
 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); }
 MPI_Finalize();
 return 0; }
```

```c
/* example of parallel Unix write into separate files */
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100

int main(int argc, char *argv[])
{
    int i, myrank, buf[BUFSIZE];
    char filename[128];
    FILE *myfile;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
    sprintf(filename, "testfile.%d", myrank);
    myfile = fopen(filename, "w");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    fclose(myfile);
    MPI_Finalize();
    return 0;
}
```

```c
/* example of parallel MPI write into separate files */
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100

int main(int argc, char *argv[])
{
    int i, myrank, buf[BUFSIZE];
    char filename[128];
    MPI_File myfile;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    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,
          MPI_STATUS_IGNORE);
    MPI_File_close(&myfile);
    MPI_Finalize();
    return 0;
}
```

MPI_File_open

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
info, MPI_File *fh)
```

comm - communicator, file opened on all of its processes. If want to open file only on one process, then use value MPI_COMM_SELF

filename - string with name of file, must be the same file on all processes in comm. User must ensure this.

amode - access mode for file

fh - pointer to opened file

User is required to close all files opened with MPI_File_open before calling MPI_Finalize

Possibilities for amode

MPI_MODE_RDONLY --- read only,
MPI_MODE_RDWR --- reading and writing,
MPI_MODE_WRONLY --- write only,
MPI_MODE_CREATE --- create the file if it does not exist,
MPI_MODE_EXCL --- error if creating file that already exists,
MPI_MODE_DELETE_ON_CLOSE --- delete file on close,
MPI_MODE_UNIQUE_OPEN --- file will not be concurrently opened elsewhere,
MPI_MODE_SEQUENTIAL --- file will only be accessed sequentially,
MPI_MODE_APPEND --- set initial position of all file pointers to end of file.

Can specify more than one, for example:

MPI_MODE_WRONLY | MPI_MODE_DELETE_ON_CLOSE

MPI_File_Close

int MPI_File_close(MPI_File *fh)

First synchronizes the file state, then closes the file associated with fh

The file is deleted if it was opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an MPI_FILE_DELETE).

If the file is deleted on close, and there are other processes currently accessing the file, the status of the file and the behavior of future accesses by these processes are implementation dependent.

The user is responsible for ensuring that all outstanding nonblocking requests and split collective operations associated with fh made by a process have completed before that process calls MPI_FILE_CLOSE.

The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to MPI_FILE_NULL.

MPI_File_read

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

fh - file handle

buf - initial address of buffer to be written

count - number of elements

datatype - datatype of each buffer element

status - status object, set to MPI_STATUS_IGNORE if not used

MPI_File_read_all is a collective version of this routine (useful if want to have all processes read from a file at some point)

Other types of MPI_File_read_... exist

MPI_File_write

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

fh - file handle

buf - initial address of buffer to be written

count - number of elements in buffer

datatype - datatype of each buffer element

status -status object, set to MPI_STATUS_IGNORE if not used


MPI maintains one individual file pointer per process per file handle.

Routines like MPI_File_write, MPI_File_read will update this pointer

MPI_File_write_all is a collective version, where each process must
have the right value of fh

Other functions of type MPI_File_write_... exist
```

```c
/* example of parallel MPI write into a single file */
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100

int main(int argc, char *argv[])
{
    int i, myrank, buf[BUFSIZE];
    MPI_File thefile;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    MPI_File_open(MPI_COMM_WORLD, "testfile",
         MPI_MODE_CREATE | MPI_MODE_WRONLY,
         MPI_INFO_NULL, &thefile);
    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,
          MPI_STATUS_IGNORE);
    MPI_File_close(&thefile);
    MPI_Finalize();
    return 0;
}
```

MPI_File_set_view

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

fh - file handle

disp - displacement

etype - datatype, specifies data layout in file

filetype - datatype, offers more control over data access, for simple cases it’s the same as etype

datarep - data representation, technical setting, usually “native” works, might have to be more careful when machines used heterogenous

info - info object, if not used, set MPI_INFO_NULL

Collective operation called on all processes on which file opened

Basically it modifies the pointer associated with fh so that subsequent read/write operations will access the correct portion of file

MPI_File_get_size

```
int MPI_File_get_size(
  MPI_File mpi_fh,
  MPI_Offset *size
);
```

mpi_fh - file handle

size - size of file in bytes

```
/* parallel MPI read with arbitrary number of processes*/
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int myrank, numprocs, bufsize, *buf, count;
    MPI_File thefile;
    MPI_Status status;
    MPI_Offset filesize;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    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 + 1;     /* 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);
printf("process %d read %d ints\n", myrank, count);
MPI_File_close(&thefile);
MPI_Finalize();
return 0; }
```

MPI_File_write_at

```
int MPI_File_write_at(MPI_File mpi_fh,MPI_Offset offset,void *buf,int
count,MPI_Datatype datatype,MPI_Status *status
);
```

Same as MPI_File_write, except for offset, which specifies where write should begin

```c
#include "mpi.h"
#include <stdio.h>
/* example of parallel MPI write into a single file */
#define BUFSIZE 100
int main(int argc, char *argv[])
{
    int i, myrank, buf[BUFSIZE];
    char filename[128];
    MPI_File thefile;
    MPI_Offset offset;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    sprintf(filename, "testfile");
    for (i = 0; i < BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    MPI_File_open(MPI_COMM_WORLD, filename,
         (MPI_MODE_WRONLY | MPI_MODE_CREATE),
         MPI_INFO_NULL, &thefile);
    MPI_File_set_view(thefile, 0, MPI_INT, MPI_INT, "native",
MPI_INFO_NULL);
    offset = myrank * BUFSIZE;
    MPI_File_write_at(thefile, offset, buf, BUFSIZE, MPI_INT,
             MPI_STATUS_IGNORE);
    MPI_File_close(&thefile);
    MPI_Finalize();
    return 0; }
```

MPI_File_write_ordered

```
int MPI_File_write_ordered(MPI_File mpi_fh,void *buf,int count,
MPI_Datatype datatype,MPI_Status *status);
```

Same arguments as simple write, but this is a collective operation

Order of access to file determined by rank of process in the group involved

Other features of MPI-2 which we will not go into due to limited time

- Process creation and management

- Remote Memory Access (RMA) which can be thought of as one-sided communication

- more collective operations

- more flexibility in defining your own operations