Input/Output

Issues with MPI and I/O

MPI standard imposes no requirements on the I/O capabilities of an MPI implementation

There are many systems that allow each process full access to stdin, stdout, and stderr

On other systems, only process 0 in MPI_COMM_WORLD has access to stdout and stderr

This creates difficulties in writing portable programs that will run on any system

Coding and debugging difficult if some processes cannot access stdout and stderr, as diagnostic output will not be produced

In general, even if access to stdout and stderr is allowed for all processes, if the number of processors is very large, communication and synchronization problems may result

In the rest of the lecture we will develop some useful functions which get around this problem, and which you may find convenient to use in your programs

We want to write some useful input/output functions, mainly to provide ourselves with program development tools.

We want to see output from all processes

Our I/O functions will turn I/O operations into collective operations with one of the processes designated as the “I/O process”

For output, I/O process will collect data from all other processes and print it out

For input, the I/O process will read in data and broadcast it to the other processes

Since these will be collective operations, each I/O function will take a communicator argument

Another purpose of our functions will be to organize the output

Even on systems which allow all processes access to stdout, there is usually no reason for the output to be synchronized. This can be inconvenient.

Example: output of a typical “Hello World” program running on many processes that executes:

code:

```
printf(“Process %d saying hello\n”, my_rank);
```

Example output:

```
  Process 4 saying hello
  Process 2 saying hello
  Process 0 saying hello
  Process 5 saying hello
  Process 3 saying hello
  Process 1 saying hello
```

The order will usually be different each time the program is executed, as there is no default mechanism to synchronize the printf output.

When debugging, we usually prefer ordered, reproducible output.

Cached attributes in communicators

We want flexibility in choosing which process will be the I/O process

We want to define a new communicator just to handle I/O

We somehow want to attach the information about which process is the I/O process to the communicator

This procedure of associating additional information with the communicator is called attribute caching.

Communicator consists of:
- group (of processes)
- context
- cached attributes

Each implementation of MPI is supposed to provide several predefined attributes. They contain useful information which can be extracted.

We can define our own attributes as needed.

One of the predefined MPI attributes should be MPI_IO
The content of this attribute is the rank of the process that can carry out language standard I/O (eg. fopen, fprintf etc.)

BUT

1. If no process can do I/O, attribute content will be predefined to MPI_PROC_NULL

2. If every process can do I/O, it will be MPI_ANY_SOURCE

3. If some processes can but others cannot then different processes can have different attribute content.
   Processes which can carry out I/O are required to have attribute content equal to their rank.
   Processes which cannot should have attribute content equal to the rank of some process that can.

Case 3 can easily happen in grid computing, where you may have a situation where your processes are running on machines which differ a great deal from each other (for example some don't have hard drives you can write to etc.).
4. MPI_IO does not indicate which process has access to standard input

Attribute value is extracted with MPI_Attr_get

MPI_Attr_get

```
int MPI_Attr_get ( MPI_Comm comm, int keyval, void *attr_value,
        int *flag )
```

comm - communicator to which attribute is attached

keyval - key value (identifies attribute) Eg. MPI_IO

attr_value - attribute value (if operation successful, check flag)
   void* is Pointer to Void, special type of pointer that can point to any data type

flag - true (i.e. nonzero) if attribute successfully extracted, false if not (for example it will fail when the attribute looked for is not defined

Need to use MPI_IO to pick an “I/O Process” which will work properly

```
int* mpi_io_ptr;
int io_rank;
int flag;

MPI_Attr_get(MPI_COMM_WORLD, MPI_IO, &mpi_io_ptr, &flag);

if (flag == 0) {
/* Attribute not cached.  Not MPI compliant */
io_rank = MPI_PROC_NULL;
} else if (*mpi_io_ptr == MPI_PROC_NULL){
io_rank = MPI_PROC_NULL;
} else if (*mpi_io_ptr == MPI_ANY_SOURCE) {
/* Any process can carry out I/O */
/* Use process 0 */
io_rank = 0;
} else {
/* Different ranks may have been returned on different processes,
so get min */
MPI_Allreduce(mpi_io_ptr, &io_rank, 1, MPI_INT,MPI_MIN,MPI_COMM_WORLD);
/* process *mpi_io_ptr can now be chosen as the I/O process */
```

Note that we have possibly changed the value of MPI_IO attribute here

Now, let’s see how we can create our own communicator attribute

This is done with two steps:

MPI_Keyval_create - we need to create the key (something like MPI_IO in previous example)

MPI_Attr_put - actually define the value of the attribute

MPI_Keyval_create

```
int MPI_Keyval_create(
        MPI_Copy_function *copy_fn,
        MPI_Delete_function *delete_fn,
        int *keyval,
        void *extra_state )
```

copy_fn - determines what happens to the attribute when the communicator it is attached to is copied. Good default value: MPI_DUP_FN which just copies the attribute exactly

delete_fn - determines what happens to attribute when communicator deleted. Good default value: MPI_NULL_DELETE_FN which does nothing extra when communicator deleted with MPI_Comm_free

keyval - key value for future access (integer). This is the output and the most important variable.

extra_state - pointer to pass additional variables to copy_fn and delete_fn functions as needed. Good default value: unused void* pointer

Features marked in purple provide a much greater degree of control over attribute creation, but that is not needed for “I/O process” problem

MPI_Attr_put

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void *attr_value )
```

comm - communicator to which attribute will be attached

keyval - key value as returned by MPI_Keyval_create

attr_value - attribute value

This function has no output, it just modifies comm.

Attributes and attribute keys in MPI are process local, as we have seen, and so are the function MPI_attr_put, MPI_Attr_get and MPI_Keyval_create

If we want to create an attribute that has the same content accross all processes in a communicator (as is the case with the “I/O process”), we need to insure this is the case

Example

```
MPI_Comm io_comm; /* Communicator for I/O */
int IO_KEY; /* I/O process attribute key we will need to define */
int* io_rank_ptr; /* Attributes are pointers */
void* extra_arg /* will not be used */

/* Get a separate communicator for I/O functions by duplicating
MPI_COMM_WORLD */

MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);

/* Create the attribute key */
MPI_Keyval_create(MPI_DUP_FN, MPI_NULL_DELETE_FN, &IO_KEY, extra_arg);

/* Allocate storage for attribute content */

io_rank_ptr = (int*) malloc(sizeof(int));

/* Set the attribute content */
*io_rank_ptr = 0; /* 0 for simple cases will work, or can identify I/O
process with more systematic procedure */

/* Cache the attribute with io_comm */
MPI_Attr_put(io_comm, IO_KEY, io_rank_ptr);
```

Example cont.

```
/* ... */
/* retrieve the data we cached */
int flag;
int* io_rank_attr;

/* Retrieve the I/O process rank */

MPI_Attr_get(io_comm, IO_KEY, &io_rank_attr, &flag);
/* If flag == 0, something went wrong, there is no attribute cached */
if ((flag != 0 ) && my_rank == *io_rank_attr))
printf (“Greetings from the I/O Process! \n);
```

Caching an I/O Process Rank

From P. Pacheco, PP with MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include "mpi.h"
#include "cio.h"

/* BUFSIZ is defined in stdio.h */
char          io_buf[BUFSIZ];

/* Key identifying IO_Attribute */
int           IO_KEY = MPI_KEYVAL_INVALID;

/* unused */
void*         extra_arg;

static int*   error_buf;
static int    error_bufsiz = 0;
```

```
/*****************************************************************/
/* Attempt to identify a process in io_comm that can be
 *     used for I/O.
 *
 * First see whether program defined io rank has been
 *     cached with either communicator.  If this fails
 *     try MPI defined io rank.
 *
 * Return values:
 *     1.  0: rank of I/O process cached with io_comm.
 *     2.  NO_IO_ATTR: couldn't find processor that could
 *         carry out I/O.  MPI_PROC_NULL cached with
 *         io_comm.
 *
 * Notes:
 *     1.  This is a collective operation, since function
 *         Copy_attr may use collective comm.
 *     2.  Only possible values cached are a valid process
 *         rank in comm2 or MPI_PROC_NULL.  (MPI_ANY_SOURCE
 *         won't be cached.)
 */
```

```
int Cache_io_rank(
        MPI_Comm   orig_comm        /* in     */,
        MPI_Comm   io_comm          /* in/out */) {

    int retval;   /* 0 or NO_IO_ATTR */

    /* Check whether IO_KEY is defined.  If not, define */
    if (IO_KEY == MPI_KEYVAL_INVALID) {
        MPI_Keyval_create(MPI_DUP_FN,
            MPI_NULL_DELETE_FN, &IO_KEY, extra_arg);
    } else if ((retval = Copy_attr(io_comm, io_comm,
                IO_KEY)) != NO_IO_ATTR) {
        /* Value cached */
        return retval;
    } else if ((retval = Copy_attr(orig_comm, io_comm,
                IO_KEY)) != NO_IO_ATTR) {
        /* Value cached */
        return retval;
    }
```

```
/* Now see if we can find a value cached for MPI_IO */
if ((retval = Copy_attr(orig_comm, io_comm,
                MPI_IO)) != NO_IO_ATTR) {
    /* Value cached */

    return retval;
} else if ((retval = Copy_attr(io_comm, io_comm,
                MPI_IO)) != NO_IO_ATTR) {
    /* Value cached */

        return retval;
    }


    /* Couldn't find process that could carry out I/O */
    /* Copy_attr has cached MPI_PROC_NULL             */
    return NO_IO_ATTR;

}  /* Cache_io_rank */
```

```
/*****************************************************************/
/* Get attribute value associated with attribute key KEY
 *     in comm1, and cache with comm2 IO_KEY
 *
 * KEY can be either IO_KEY or MPI_IO.
 *
 * Return values:
 *     1.  0:  valid attribute successfully cached.
 *     2.  NO_IO_ATTR:  Couldn't find process that could
 *         carry out I/O.  MPI_PROC_NULL is cached with
 *         comm2.
 */
int Copy_attr(
        MPI_Comm   comm1    /* in     */,
        MPI_Comm   comm2    /* in/out */,
        int        KEY      /* in     */) {

    int   io_rank;
    int   temp_rank;
    int*  io_rank_ptr;
    int   equal_comm;
    int   flag;
```

```
MPI_Attr_get(comm1, KEY, &io_rank_ptr, &flag);

if (flag == 0) {
    /* Attribute not cached with comm1 */
    io_rank_ptr = (int*) malloc(sizeof(int));
    *io_rank_ptr = MPI_PROC_NULL;
    MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
    return NO_IO_ATTR;
} else if (*io_rank_ptr == MPI_PROC_NULL) {
    MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
    return NO_IO_ATTR;
} else if (*io_rank_ptr == MPI_ANY_SOURCE) {
    /* Any process can carry out I/O.  Use */
    /* process 0                          */

    io_rank_ptr = (int*) malloc(sizeof(int));
    *io_rank_ptr = 0;
    MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);

    return 0;
}
```

```
/* Value in *io_rank_ptr is a valid process  */
/* rank in comm1.  Action depends on whether */
/* comm1 == comm2.                           */
MPI_Comm_compare(comm1, comm2, &equal_comm);

if (equal_comm == MPI_IDENT) {
    /* comm1 == comm2.  Valid value already */
    /* cached.  Do nothing.                 */
    return 0;
```

```
    } else {
        /* Check whether rank returned is valid */
        /* process rank in comm2                */
        Get_corresp_rank(comm1, *io_rank_ptr,
            comm2, &temp_rank);

        /* Different ranks may have been returned */
        /* on different processes.  Get min       */
        if (temp_rank == MPI_UNDEFINED)
            temp_rank = HUGE;
        MPI_Allreduce(&temp_rank, &io_rank, 1, MPI_INT,
            MPI_MIN, comm2);

        io_rank_ptr = (int*) malloc(sizeof(int));
        if (io_rank < HUGE) {
            *io_rank_ptr = io_rank;
            MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
            return 0;
        } else {
            /* No process got a valid rank in comm2 */
            /* from Get_corresp_rank                */
            *io_rank_ptr = MPI_PROC_NULL;
            MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
            return NO_IO_ATTR;
        }
    }
}  /* Copy_attr  */
```

```
/*****************************************************/
/* Determines whether the process with rank rank1 in
 *     comm1 is a valid rank in comm2.
 * If it is, it returns the rank in *rank2_ptr.  If it
 *     isn't it returns MPI_UNDEFINED.
 *
 */
void Get_corresp_rank(
        MPI_Comm   comm1       /* in  */,
        int        rank1       /* in  */,
        MPI_Comm   comm2       /* in  */,
        int*       rank2_ptr   /* out */) {

    MPI_Group  group1;
    MPI_Group  group2;

    MPI_Comm_group(comm1, &group1);
    MPI_Comm_group(comm2, &group2);

    MPI_Group_translate_ranks(group1, 1,
            &rank1, group2, rank2_ptr);

}  /* Get_corresp_rank */
```

```
/*****************************************************************/
/* Check whether IO_KEY is valid.  If it is, attempt to
 *     access it.  If it isn't attempt to define it from
 *     MPI_COMM_WORLD.
 *
 * Return values:
 *     1.  0: Valid rank returned.
 *     2.  NO_IO_ATTR:  Unable to find rank.
 */
int Get_io_rank(
        MPI_Comm io_comm      /* in  */,
    int*     io_rank_ptr  /* out */) {

int*  temp_ptr;
int   flag;
```

```
if (IO_KEY == MPI_KEYVAL_INVALID) {
    MPI_Keyval_create(MPI_DUP_FN,
        MPI_NULL_DELETE_FN, &IO_KEY, extra_arg);
} else {
    MPI_Attr_get(io_comm, IO_KEY, &temp_ptr, &flag);
    if ((flag != 0) && (*temp_ptr != MPI_PROC_NULL)) {
        *io_rank_ptr = *temp_ptr;
        return 0;
    }
}
```

```
    if (Copy_attr(MPI_COMM_WORLD, io_comm, MPI_IO)
            == NO_IO_ATTR) {
        return NO_IO_ATTR;
    } else {
        MPI_Attr_get(io_comm, IO_KEY, &temp_ptr, &flag);
        *io_rank_ptr = *temp_ptr;
        return 0;
    }

}  /* Get_io_rank */
```

Cscanf

Cscanf which we will define will be the a kind of collective scanf. We pick similar names to remind ourselves they do something similar.

In Cscanf, the I/O process will just read in a line of data as a string and broadcast it to all processes

Cscanf is general, the number of arguments it can be called with will vary, as we may want to scan different numbers of variables
Each process will make use of stdarg macros to parse the input line

vsscanf will be used - can read data from a string into a variable length argument list

Cscanf

```
/* Prompt for input, read one line and broadcast.
 *
 * Return values:
 *     1. 0:  input read
 *     2. NO_IO_ATTR:  no rank cached with IO_KEY.
 *
 * Notes:
 *     1. Prompt is significant only on IO_process
 */
int Cscanf(
        MPI_Comm  io_comm  /* in  */,
        char*     prompt   /* in  */,
        char*     format   /* in  */,
                  ...      /* out */) {

    va_list  args;
    int      my_io_rank;
    int      root;
```

```
if (Get_io_rank(io_comm, &root) == NO_IO_ATTR)
    return NO_IO_ATTR;
MPI_Comm_rank(io_comm, &my_io_rank);

/* Read in data on root */
if (my_io_rank == root) {
    printf("%s\n", prompt);
    gets(io_buf);
}

    /* Broadcast the input data */
    MPI_Bcast(io_buf, BUFSIZ, MPI_CHAR, root, io_comm);

    /* Copy the input data into the parameters */
    va_start(args, format);
    vsscanf(io_buf, format, args);
    va_end(args);

    return 0;
} /* Cscanf */
```

Example usage

```
ret_val = Cscanf(MPI_COMM_WORLD, "Enter an int, a float, and a string",
        "%d %f %s", &ival, &fval, &sval);
ret_val = Cprintf(MPI_COMM_WORLD, "MPI_COMM_WORLD read:","%d %f %s",
        ival, fval, sval);
```

Writing to stdout - Cprintf

Copy output data into string and gather each process’s string onto the I/O process, which prints them

```
/*****************************************************************/
/* Prints data from all processes.  Format of data must
 *     be the same on each process.
 *
 * Return values:
 *     1.  0:  data printed
 *     2.  NO_IO_ATTR:  no rank cached with IO_KEY
 *
 * Notes:
 *     1.  Title is significant only on root.
 */

int Cprintf(
    MPI_Comm  io_comm  /* in */,
    char*     title    /* in */,
    char*     format   /* in */,
              ...      /* in */) {

int         q;
int         my_io_rank;
int         io_p;
int         root;
MPI_Status  status;
va_list     args;
```

```
if (Get_io_rank(io_comm, &root) == NO_IO_ATTR)
    return NO_IO_ATTR;
MPI_Comm_rank(io_comm, &my_io_rank);
MPI_Comm_size(io_comm, &io_p);

/* Send output data to io_process */
if (my_io_rank != root) {
    /* Copy the output data into io_buf */
    va_start(args, format);
    vsprintf(io_buf, format, args);
    va_end(args);
    MPI_Send(io_buf, strlen(io_buf) + 1, MPI_CHAR,
        root, 0, io_comm);
} else { /* my_io_rank == root */
    printf("%s\n",title);
    fflush(stdout);
    for (q = 0; q < root; q++) {
        MPI_Recv(io_buf, BUFSIZ, MPI_CHAR, q,
            0, io_comm, &status);
        printf("Process %d > %s\n",q, io_buf);
        fflush(stdout);
    }
```

```
/* Copy the output data into io_buf */
va_start(args, format);
vsprintf(io_buf, format, args);
va_end(args);
printf("Process %d > %s\n",root, io_buf);
fflush(stdout);

for (q = root+1; q < io_p; q++) {
    MPI_Recv(io_buf, BUFSIZ, MPI_CHAR, q,
        0, io_comm, &status);
        printf("Process %d > %s\n",q, io_buf);
        fflush(stdout);
    }
    printf("\n");
    fflush(stdout);
}

    return 0;
} /* Cprintf */
```

Similarly, a function Cerror_test can be called across all processes to collectively check if an error has occurred on any one process

If error has occured, the function will print a message on the I/O process, then shut down the whole program with MPI_Abort

Example: say you are allocating a large chunk of memory with malloc at roughly the same time on all processes. Cerror would give you a way to check if the allocation was successful across all processes

What if standard input (stdin) is not available?

Symptom: program crashes after attempting to read from stdin

Several options to deal with situation:

- Read from files, other than stdin

- self-initialization i.e. put the data needed in your source code

- use command line arguments (most implementations allow this) Very useful if input is short

Command line arguments example

```
#include <stdio.h>
#include mpi.h

/* Header file for our I/O library */
#include “cio.h”

int main(int argc, char* argv[]){
MPI_Comm io_comm;
int i;

MPI_Init(&argc, &argv);
/* Set up communicator for I/O */
MPI_Comm_dup(MPI_COMM_WORLD, &io_comm)
Cache_io_rank(MPI_COMM_WORLD, io_comm);

for(i=0,i<argc;i++)
Cprintf(io_comm,””,”argv[%d]=%s”,i,argv[i]);

MPI_Finalize();

}
```

Output

```
mpicc test.c -o argtest
mpirun -np 2 argest hello, world
or on SHARCNET
sqsub -q mpi --test -n 2 -r 1m -o test.txt argest hello, world

Output:

Process 0 > argv[0] = /home/yourname/argtest
Process 1 > argv[0] = /home/yourname/argtest

Process 0 > argv[1] = hello,
Process 1 > argv[1] = hello,

Process 0 > argv[2] = world
Process 1 > argv[2] = world
```

File I/O

So far, all discussion has been about “simple” streams like stdin, stdout, stderr

With file I/O, there is the possibility of multiple input and output streams.

Key issue with file I/O is data mapping.

Example problem:

Large array of data has been distributed on d disks, and we wish to access this array in a program with p processes.

In general p will not be same as d, and data will have to be redistributed when it is read in

Details of implementing this is a subject of intensive research

In case of d=1, we can just modify our collective I/O functions so that they take a file parameter.

Cprintf would become Cfprintf, with file only open on I/O process

If only some processes can have access to standard streams, but all can read from a file then we can just set the I/O process to be zero in this case, for example.

Assuming file writing is possible by all processes, and that each process is writing to its own separate file, then can use standard C functions (eg. fprintf) on all of them i.e. there is no need for collective communications

```
sprintf(filename, “file.%d”,my_rank);
my_fp = fopen(filename, “w”);
fprintf = (my_fp, “greetings from proceess %d! \n”,my_rank);
flocse (my_fp);
```

It is essential in this case to make sure that each process opens a file with a different file name, unless one is certain that each process is accessing its own independent disk

Why do we have to write our own input/output collective functions?

Why are they not a standard part of MPI and implemented as functions, i.e. MPI_Cprinft etc. ?

It is because there is no consensus on the mechanics of I/O on parallel systems, so the MPI standard imposes no requirements on the I/O capabilities of an MPI implementation.

Unfortunately, this means programmers have no guaranteed I/O interface, so they have to develop their own, as we have done here

Updates to the MPI standard (MPI 2.0 and its possible successors) may have these functions included