Parallel Computing Systems and Applications
MPI — Extras

Salvatore Filippone

salvatore.filippone@uniroma2.it

Getting/setting environment parameters
Compile or run time:

```
#define MPI_VERSION 3
#define MPI_SUBVERSION 1

int MPI_Get_version(int *version, int *subversion)

int MPI_Get_library_version(char *version, int *resultlen)
int MPI_Get_processor_name(char *name, int *resultlen)
```

Environmental Management

Allocating (special-purpose) memory:

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
int MPI_Free_mem(void *base)
```

Example:

```
float (* f)[100][100];
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

Environmental Management

Error handling. Predefined error handlers:

MPI_ERRORS_ARE_FATAL
MPI_ERRORS_RETURN

User-defined error handlers:

```
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);

int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
                       *comm_errhandler_fn, MPI_Errhandler *errhandler)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)

int MPI_Win_create_errhandler(MPI_Win_errhandler_function
                           *win_errhandler_fn, MPI_Errhandler *errhandler)
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
int MPI_File_create_errhandler(MPI_File_errhandler_function
                *file_errhandler_fn, MPI_Errhandler *errhandler)
```

More on error handling

```
int MPI_Add_error_class(int *errorclass)
int MPI_Add_error_code(int errorclass, int *errorcode)
int MPI_Add_error_string(int errorcode, const char *string)

int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
```

Timers:

```
double MPI_Wtime(void)
double MPI_Wtick(void)
```

Startup & friends

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize(void)
int MPI_Initialized(int *flag)
int MPI_Finalized(int *flag)

int MPI_Abort(MPI_Comm comm, int errorcode)
```

Dynamic Process Creation

It is possible to create ("spawn") MPI processes dynamically. Given the enormous variety of execution environment it is impossible to give general advice, but it is possible to use the MPI_COMM_WORLD attribute

    MPI_UNIVERSE_SIZE

to make informed decisions

```
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
       MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
       int array_of_errcodes[])

int MPI_Comm_get_parent(MPI_Comm *parent)
```

Dynamic Process Creation

```c
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
  int world_size, universe_size, *universe_sizep, flag;
  MPI_Comm everyone;  /* intercommunicator */
  char worker_program[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&world_size);
  if (world_size != 1)
    error("Top heavy with management");
  MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                    &universe_sizep, &flag);
  if (!flag) {
    printf("This MPI does not support UNIVERSE_SIZE. How many\n\
processes total?");
    scanf("%d", &universe_size);
  } else universe_size = *universe_sizep;
  choose_worker_program(worker_program);
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
        MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,MPI_ERRCODES_IGNORE);
  MPI_Finalize();
  return 0;
}
```

```c
/* worker */
#include "mpi.h"
int main(int argc, char *argv[])
{
  int size;
  MPI_Comm parent;
  MPI_Init(&argc, &argv);
  MPI_Comm_get_parent(&parent);
  if (parent == MPI_COMM_NULL) error("No parent!");
  MPI_Comm_remote_size(parent, &size);
  if (size != 1) error("Something is wrong with the parent");
  /*
   * Parallel code here.
   * The manager is represented as the process with rank 0 in (the remote
   * group of) the parent communicator. If the workers need to communicate
   * among themselves, they can use MPI_COMM_WORLD.
   */
  MPI_Finalize();
  return 0;
}
```

Parallel I/O

From the MPI standard document

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.
The significant optimizations required for efficiency (e.g., grouping [47], collective buffering [7, 15, 48, 52, 58], and disk-directed I/O [43]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks).
The I/O environment described in this chapter provides these facilities. Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

The MPI I/O subsystem is powerful, but very complicated because it imposes a *structure* on the files, instead of using just byte sequence. Use it when necessary.

The MPI I/O interface

file An MPI file is an ordered collection of typed data items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.

displacement A file displacement is an absolute byte position relative to the beginning of a file.

etype An etype (elementary datatype) is the unit of data access and positioning.

filetype A filetype is the basis for partitioning a file among processes and defines a template for accessing the file

view A view defines the current set of data visible and accessible from an open file as an ordered set of etypes.

```c
int MPI_File_open(MPI_Comm comm, const char *filename, int amode,
      MPI_Info info, MPI_File *fh)
int MPI_File_close(MPI_File *fh)
int MPI_File_delete(const char *filename, MPI_Info info)
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
int MPI_File_get_amode(MPI_File fh, int *amode)
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
      MPI_Datatype filetype, const char *datarep, MPI_Info info)
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, const void *buf,
      int count, MPI_Datatype datatype, MPI_Status *status)
```

Generalized Requests

Used to implement new non-blocking operations

```
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
   MPI_Grequest_free_function *free_fn,
   MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
   MPI_Request *request)
typedef int MPI_Grequest_free_function(void *extra_state);
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
int MPI_Grequest_complete(MPI_Request request)
```