Info objects in MPI

- We have skipped over Info objects by using `MPI_INFO_NULL`
- Sometimes they are useful hints to MPI implementations as to what to do
- Info objects are collections of key-value pairs
- MPI_Info_create(* info)
- MPI_Info_free(* info)
- MPI_Info_set(info, key, value)

Why use Info object

- Reduce number of arguments to function calls
- Functions that interact with systems in a non-uniform way
- If you don't need specific arguments or options you can leave them blank
- Examples include hints on opening files (more shortly), where to create new MPI tasks using MPI_Comm_spawn, ordering of operations on RMA windows

Parallel IO

- The final major addition to the MPI-2 standard is parallel IO
- MPI-1 provided no support for this
- Usually rank 0 carried out all the IO and then used functions like MPI_Bcast and MPI_Scatter to distribute the data to other nodes
- Portable from serial code but not very scalable to large datasets or large numbers of MPI tasks
- Allows for use of complex IO libraries for data formats such as NetCDF or HDF5

Parallel IO

- One method to get around this is for each MPI task to read/write from its own files
- This enables parallel transfer of data
- Allows each process to use IO libraries
- Several disadvantages
  - Hard to keep all the files together when moving from system to system
  - May have to combine the files to be used as input to another program
  - May need to run future calculations on exactly the same number of cores
  - Poor performance from small IO chunks
- NOTE: all MPI-IO is done as binary read/write. You can't edit the files using vim.

Opening and closing files

- Use the `MPI_File` datatype
- MPI_File_open(comm, fname, mode, info, *file)
- A collective operation across the communicator
- Can use `MPI_COMM_SELF` if you only want to open the file on a single process
- MPI_File_close(*file)
- Again collective. Ensures all processes close the file correctly.
- File value is set to `MPI_FILE_NULL` to allow user to find bad file accesses

Reading and writing

- MPI provides basic read/write functions
- MPI_File_read(file, buf, count, type, status)
- MPI_File_write(file, buf, count, type, status)
- Like printf/scanf, these advance the file pointer

Using file pointers

- Using the normal C library we use a file pointer to tell us where in the file we are currently looking.
- Calls to fread/fscanf move that pointer along through the file in a linear fashion
- Can use functions like `fseek` and `rewind` to move around in the file

```
FILE *fp = fopen("foo", "r");
for(i=0;i<100;i++)
    fgets(buf, 1000, fp);

fseek(fp, -200, SEEK_CUR);
rewind(fp);
```

Using file pointers

- There is a similar function for moving the file pointer in MPI
- Each MPI task has its own pointer to the file. It is not a single shared pointer.
- MPI_File_seek(file, offset, whence);
- The offset is of type `MPI_Offset` which is an integer type that is long enough to hold the largest filesize.
- The third argument can be set to one of three values
  - MPI_SEEK_SET
  - MPI_SEEK_CUR
  - MPI_SEEK_END

Using file pointers

```
insize = N/size;
in = malloc(insize * sizeof(int));
offset = rank * insize * sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "intfile",
  MPI_MODE_RDONLY, MPI_INFO_NULL, &fp);

MPI_File_seek(fp, offset, MPI_SEEK_SET);
MPI_File_read(fp, in2, insize, MPI_INT, &stat);
MPI_File_close(&fp);
```

Using offsets

- As opposed to moving the file pointer on each task we can use the explict-offset functions
- This is especially useful in multi-threaded applications as MPI_File_read advances the file pointer
- MPI_File_read_at(file, offset, buf, count, type, status)
- MPI_File_write_at(file, offset, buf, count, type, status)
- These functions leave the file pointer unchanged

Non contiguous IO

- So far we have just read contiguous sub-chunks of a file on each task
- MPI-IO really shines when we use non-contiguous accesses
- The data in the file might be interleaved so we want to undertake a bulk read/write at the file system level but each individual task is only doing small updates

File Views

- By default each MPI task can see all of the file
- Each one can read/write any value and on opening each file pointer is set to offset 0 — the start of the file
- Sometimes we might want to restrict what data each task can see
- These are called *file views*
- Tasks can only access data visible in their view. All other data is skipped over

File Views

- File views are set using MPI_File_set_view
- We use a triplet of *displacement*, *etype* and *filetype* to specify the view
- Displacement is the distance into the file before the view starts. Used to skip over headers etc. It is always measured in bytes
- Etype is the extent of the datatype stored in the file
- It can be a basic or defined datatype
- All offsets are declared in units of etype

File Views

- Filetype describes the "chunk" size within the file
- This must be the same as etype or a multiple of etype
- The file view starts after the displacement and continues with multiple copies of the filetype
- The default file view created when the file is first opened has displacement 0 and both etype and filetype set to `MPI_BYTE`

File Views

```
/* Create a type of 10 integers */
MPI_Type_contiguous(10, MPI_INT, &ltype);

/* Resize the type so it is 40 ints long */
ext = 40 * sizeof(int);
MPI_Type_create_resized(ltype, 0, ext, &ftype);
MPI_Type_commit(&ftype);

/* Note the per-rank extra displacement */
MPI_File_set_view(fp, DISP + rank*10*sizeof(int),
    MPI_INT, ftype, "native", MPI_INFO_NULL);

MPI_File_read(fp, buf, N, MPI_INT, &stat);
```

Collective IO

- MPI provides collective IO functions MPI_File_read_all and MPI_File_write_all
- There are also read/write_at_all versions of the functions
- May need to modify the types or view to fit
- Can provide major performance improvement

Collective IO

- See `mpi-matrix-write.c` for details
- 4 MPI tasks, each with a sub-block of a $2N \times 2N$ matrix
- Want to write out the full matrix with a single collective call
- Use MPI_Type_create_subarray to represent each local $N \times N$ matrix as a sub-array of the whole matrix
- Use the displacement to start each task writing in the correct place

Non-blocking IO

- There are non-blocking versions for all the non-collective MPI read/write functions
- MPI_File_iread, MPI_File_iread_at etc.
- Replace the status object with a request object
- Then use the usual MPI_Test, MPI_Wait functions to see if the IO has completed

Non-blocking IO

- Partial support for non-blocking collective IO, sometimes called split-collective IO
- To use a split-collective first call (say) MPI_File_read_all_begin
- Do some other work not using the data
- Then call a matching MPI_File_read_all_end to complete the IO
- No other actions can be made on the file pointer between the begin and end calls

Shared File Pointer

- Up to now each MPI task has been using its own local file pointer for parallel IO operation
- There is also a pointer that is shared by all MPI tasks in the communicator on which the file was opened
- A call to the shared pointer from any task will read/write data and advance the shared pointer
  - MPI_File_read_shared
  - MPI_File_write_shared
  - MPI_File_seek_shared
- Also collective operations on the shared pointer
  - MPI_File_read_ordered
  - MPI_File_write_ordered
- Actions carried out as if done in rank order

File Portability

- Different systems may have different data representations — sizes of datatypes, endianess etc
- Files written on one may not be readable on another
- Can improve portability by using the representation option in the file view
  - native — store the data directly as it is in memory
  - internal — a representation consistant within that implementation even if run on different platforms
  - external32 — write all data in IEEE 32-bit big-endian format
- Moving down the options will probably decrease performance