MPI-IO

Chris Brady
Heather Ratcliffe

Warwick RSE

Getting data in and out

- The purpose of MPI-IO is to get data in or out of an MPI parallel program to or from disk
- For primary data representation there are libraries
  - NetCDF
  - HDF5
- Might be easier than writing your own
- But, you might want to if
  - Getting data from, or giving data to another code with a specific format
  - Ultimate performance!

Alternatives

- Send all data to rank 0 and writing normal file
  - Strictly serial
  - Requires rank 0 to have enough memory to store all data (at least for 1 variable)
  - Takes no advantage of special IO hardware in HPC systems

Alternatives

- Write 1 file per rank
  - Performance surprisingly OK
  - Bottlenecks hard with large numbers of files
    - Especially on some systems (Lustre)
    - Sysadmin might seek your death
  - Leaves you with a lot of files to maintain
  - Can't restart easily on different number of processors

“Rules” for IO

- Even the best system is slow compared with compute or communication
- Do as much reduction in code as possible before writing
- Write as little data as possible
- If IO is limiting feature of your code, check if you really need parallelism
  - Might be easier to get workstation with lots of memory

MPI-IO concepts

Concepts

- Almost exactly the same as normal file IO
- You have
  - **Opening (fopen, OPEN)** routines giving you
    - **File handles (FILE*, LUN)** - describe a given file
  - **Position (fseek, POS=)** routines that let you get or set
    - **File pointers** - describe where you are "looking" in a file
  - **Read/write (fread/fwrite, READ/WRITE)** routines
    - Read or Write data at the location of the **file pointer**
  - **Sync (fsync, N/A)** - Flush data from buffers to disk. (Called sync in MPI)
  - **Close(fclose, CLOSE)** routines to close the **file handle**

Concepts

- In MPI-IO there are two **file pointers**
  - Individual pointer - each rank maintains a separate pointer
  - Shared pointer - a file pointer that is held in common across all rank
- You can read or write using either pointer with different routines
- Finally, there is the concept of a **file view**
  - Maps data from multiple processors to representation on disk
  - Deal with later

Note for Fortran

- MPI-IO defines a MPI_Offset type to represent byte offsets in files
- In Fortran this becomes INTEGER(KIND=MPI_OFFSET_KIND)
- Using a simple INTEGER will, at best, fail to compile
- Sometimes it will compile and then crash
- This includes INTEGER literals

Handling files

MPI_File_open

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

- `comm` - Communicator. For some operations, all processors in comm must call the same function
- `filename` - the name of the input/output file
- `amode` - the mode with which to open file (see next slide). Combine modes by bitwise OR (or addition with care)
- `info` - Used to provide additional information to the MPI-IO system. System dependent, so here we just use MPI_INFO_NULL
- `fh` - File handle object

MPI_File_open modes

- MPI_MODE_RDONLY - Read only
- MPI_MODE_RDWR - Read write
- MPI_MODE_WRONLY - Write only
- MPI_MODE_CREATE - Create file if it doesn’t exist
- MPI_MODE_EXCL - Throw error if creating file that exists
- MPI_MODE_DELETE_ON_CLOSE - Delete file when closed (temporary file)
- MPI_MODE_UNIQUE_OPEN - File will not be opened elsewhere (either by your code, or by other systems (backups etc.))
- MPI_MODE_SEQUENTIAL - File will not have file pointer moved manually
- MPI_MODE_APPEND - Move file pointer to end of file at opening

MPI_File_close
MPI_File_sync

```
int MPI_File_sync(MPI_File fh)
```

- **`fh`** - File handle from MPI_File_open

```
int MPI_File_close(MPI_File *fh)
```

- **`fh`** - File handle from MPI_File_open

MPI_File_delete

```
int MPI_File_delete(char *filename, MPI_Info info)
```

- `filename` - Name of file to delete
- info - MPI_Info object holding hints for the file system. These are system dependent. Can be MPI_INFO_NULL

Writing using individual pointers

MPI_File_seek

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

- `fh` - File handle from MPI_File_open
- `offset` - Offset from `whence` in bytes. Can be negative
- `whence` - Where to set the offset from
    - MPI_SEEK_SET - seek from start of the file
    - MPI_SEEK_CUR - seek from current file pointer position
    - MPI_SEEK_END - seek from end of file. Use negative offset to go backwards from end

Collective operations

- Two types of reading and writing operation
  - MPI_File_read/MPI_File_write
    - Non collective
    - Can be called by any processor as desired
  - MPI_File_read_all/MPI_File_write_all
    - Collective
    - Must be called by all processors in the communicator given to **`MPI_File_open`**
    - Generally gives superior performance in HPC
    - Otherwise exactly the same

Write/Read

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

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

- `fh` - File handle from MPI_File_open
- `buf` - Buffer for data to be read from/written to
- `count` - Number of elements of `datatype` to be read/written
- datatype - MPI_Datatype of the elements to be written. Can be a custom datatype
- `status` - Information about state of read/write. Can be MPI_STATUS_IGNORE

Write_all/Read_all

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

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

- `fh` - File handle from MPI_File_open
- `buf` - Buffer for data to be read from/written to
- `count` - Number of elements of `datatype` to be read/written
- datatype - MPI_Datatype of the elements to be written. Can be a custom datatype
- `status` - Information about state of read/write. Can be MPI_STATUS_IGNORE

Write example

```
PROGRAM simple_write

  USE mpi
  IMPLICIT NONE

  INTEGER :: rank, nproc, ierr
  INTEGER :: file_handle
  CHARACTER(len=50) :: outstr

  CALL MPI_Init(ierr)
  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  !Delete the existing file
  CALL MPI_File_delete('out.txt', MPI_INFO_NULL, ierr)
  !Open the file for writing
  CALL MPI_File_open(MPI_COMM_WORLD, 'out.txt', &
      MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, file_handle, ierr)

  !MPI_IO is a binary output format. Have to manually add new line characters
  WRITE(outstr,'(A,I3,A)') "Hello from processor ", rank, NEW_LINE(' ')

  !Write using the individual file pointer
  CALL MPI_File_write(file_handle, TRIM(outstr), LEN(TRIM(outstr)), &
      MPI_CHARACTER, MPI_STATUS_IGNORE, ierr)
  !Close the file
  CALL MPI_File_close(file_handle, ierr)
  CALL MPI_Finalize(ierr)

END PROGRAM simple_write
```

Output on 16 cores

```
Hello from processor    3
```

- Only have a single line of output
- Because all of them are writing using their own individual pointers
  - All pointing to start of file
- Random which processor writes last and ends up being in the file

Fix using MPI_File_seek

```
!MPI_IO is a binary output format. Have to manually add new line characters
WRITE(outstr,'(A,I3,A)') "Hello from processor ", rank, NEW_LINE(' ')

!Get the lengths of all other writes
CALL MPI_Allgather(LEN(TRIM(outstr)), 1, MPI_INTEGER, offsets, 1, &
    MPI_INTEGER, MPI_COMM_WORLD, ierr)

!Calculate this processors offset in the file
my_offset = SUM(offsets(1:rank))

!Move the file pointer to that place
CALL MPI_File_seek(file_handle, my_offset, MPI_SEEK_SET, ierr)

!Write using the individual file pointer
CALL MPI_File_write(file_handle, TRIM(outstr), LEN(TRIM(outstr)), &
    MPI_CHARACTER, MPI_STATUS_IGNORE, ierr)
```

- Use MPI_Allgather to get the lengths of all strings
- Then sum the offsets for ranks lower than current processor
- Use MPI_File_seek to seek to that offset

Output now

```
Hello from processor   0
Hello from processor   1
Hello from processor   2
Hello from processor   3
Hello from processor   4
Hello from processor   5
Hello from processor   6
Hello from processor   7
Hello from processor   8
Hello from processor   9
Hello from processor  10
Hello from processor  11
Hello from processor  12
Hello from processor  13
Hello from processor  14
Hello from processor  15
```

- Now works as expected
- Can do the same using shared pointer

Writing using shared pointers

Shared pointers

- Kept in sync by all processors
- Writing or reading on one processor moves file pointer for all processors
- Only one processor can “own” shared pointer for writing or reading at a time
- Comes with a performance hit
- Intrinsically collective, no non-collective version

Write_shared/Read_shared

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

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

- `fh` - File handle from MPI_File_open
- `buf` - Buffer for data to be read from/written to
- `count` - Number of elements of `datatype` to be read/written
- datatype - MPI_Datatype of the elements to be written. Can be a custom datatype
- `status` - Information about state of read/write. Can be MPI_STATUS_IGNORE

Output now

```
Hello from processor  10
Hello from processor   0
Hello from processor   1
Hello from processor  12
Hello from processor   6
Hello from processor  14
Hello from processor   3
Hello from processor   2
Hello from processor   4
Hello from processor  11
Hello from processor  13
Hello from processor   5
Hello from processor  15
Hello from processor   8
Hello from processor   9
Hello from processor   7
```

- Output is all there, but in random order
- MPI_File_write_shared is on “first come, first served” basis

Write_ordered/Read_ordered

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

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

- `fh` - File handle from MPI_File_open
- `buf` - Buffer for data to be read from/written to
- `count` - Number of elements of `datatype` to be read/written
- datatype - MPI_Datatype of the elements to be written. Can be a custom datatype
- `status` - Information about state of read/write. Can be MPI_STATUS_IGNORE

Output now

```
Hello from processor   0
Hello from processor   1
Hello from processor   2
Hello from processor   3
Hello from processor   4
Hello from processor   5
Hello from processor   6
Hello from processor   7
Hello from processor   8
Hello from processor   9
Hello from processor  10
Hello from processor  11
Hello from processor  12
Hello from processor  13
Hello from processor  14
Hello from processor  15
```

- Output is all there, in rank order
- Processors have to queue up
- Can serialise output, performance penalty

MPI_File_seek_shared

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
```

- **`fh`** - File handle from MPI_File_open
- **`offset`** - Offset from **`whence`** in bytes. Can be negative
- **`whence`** - Where to set the offset from
    - MPI_SEEK_SET - seek from start of the file
    - MPI_SEEK_CUR - seek from current file pointer position
    - MPI_SEEK_END - seek from end of file. Use negative offset to go backwards from end

MPI_File_seek_shared

- Do not have different values for whence or offset on different processors
- Not defined what will happen
- Probably won't be what you want
- Will likely change on different MPI implementation

File views

File view concepts

- The most powerful and useful part of MPI-IO is the **file view**
- This maps data on the current processor to its place in a "global" view of the data
- Does this using MPI custom types
- Since generally mapping a subsection of an array, good match to `MPI_Type_create_subarray`

MPI_File_set_view

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

- fh - File handle from MPI_File_open
- disp - Displacement of view from start of file in bytes
- etype - Primitive type for data in view. Should be shortest datatype being written. MPI_BYTE is acceptable in all cases. Must have same extent on all ranks
- filetype - Type representing layout of data
- datarep - String representing how data should be represented. Usually “native”
- info - MPI_Info object containing hints. Good description at https://www.open-mpi.org/doc/v2.0/man3/MPI_File_set_view.3.php. MPI_INFO_NULL is acceptable

Array subsection

nx x ny array of characters (here 16)

A	E	I	M
B	F	J	N
C	G	K	O
D	H	L	P

1 character per processor

- Split processors up using `MPI_Cart_create` again
- MPI_Type_create_subarray needs
  - Sizes
  - Subsizes
  - Starts

Array subsection

nx x ny array of characters
(here 16)

A	E	I	M
B	F	J	N
C	G	K	O
D	H	L	P

1 character per processor

- For all processors
  - sizes = (nx, ny)
  - subsizes = (1, 1)
- Starts are just coordinates from communicator

Array subsection

```
!Create the MPI Cartesian communicator
CALL MPI_Dims_create(nproc, 2, nprocs_cart, ierr)
CALL MPI_Cart_create(MPI_COMM_WORLD, 2, nprocs_cart, periods, .TRUE., &
    cart_comm, ierr)
CALL MPI_Comm_rank(cart_comm, rank, ierr)
CALL MPI_Cart_coords(cart_comm, rank, 2, coords, ierr)

!Open the file for output
CALL MPI_File_open(cart_comm, 'out.txt', &
    MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, file_handle, ierr)

!Create the type representing a single character on this processor
sizes = nprocs_cart
subsizes = (/1, 1/)
starts = coords !Output character at it’s coordinate in the Cartesian comm
CALL MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_FORTRAN, &
    MPI_CHARACTER, view_type, ierr)
CALL MPI_Type_commit(view_type, ierr)

!Set the view using that type
CALL MPI_File_set_view(file_handle, offset, MPI_BYTE, view_type, 'native', &
    MPI_INFO_NULL, ierr)

!Write the file using a collective write
outstr = ACHAR(rank + ICHAR('A'))
CALL MPI_File_write_all(file_handle, outstr, 1, MPI_CHARACTER, &
    MPI_STATUS_IGNORE, ierr)

!Close the file
CALL MPI_File_close(file_handle, ierr)
```

Output

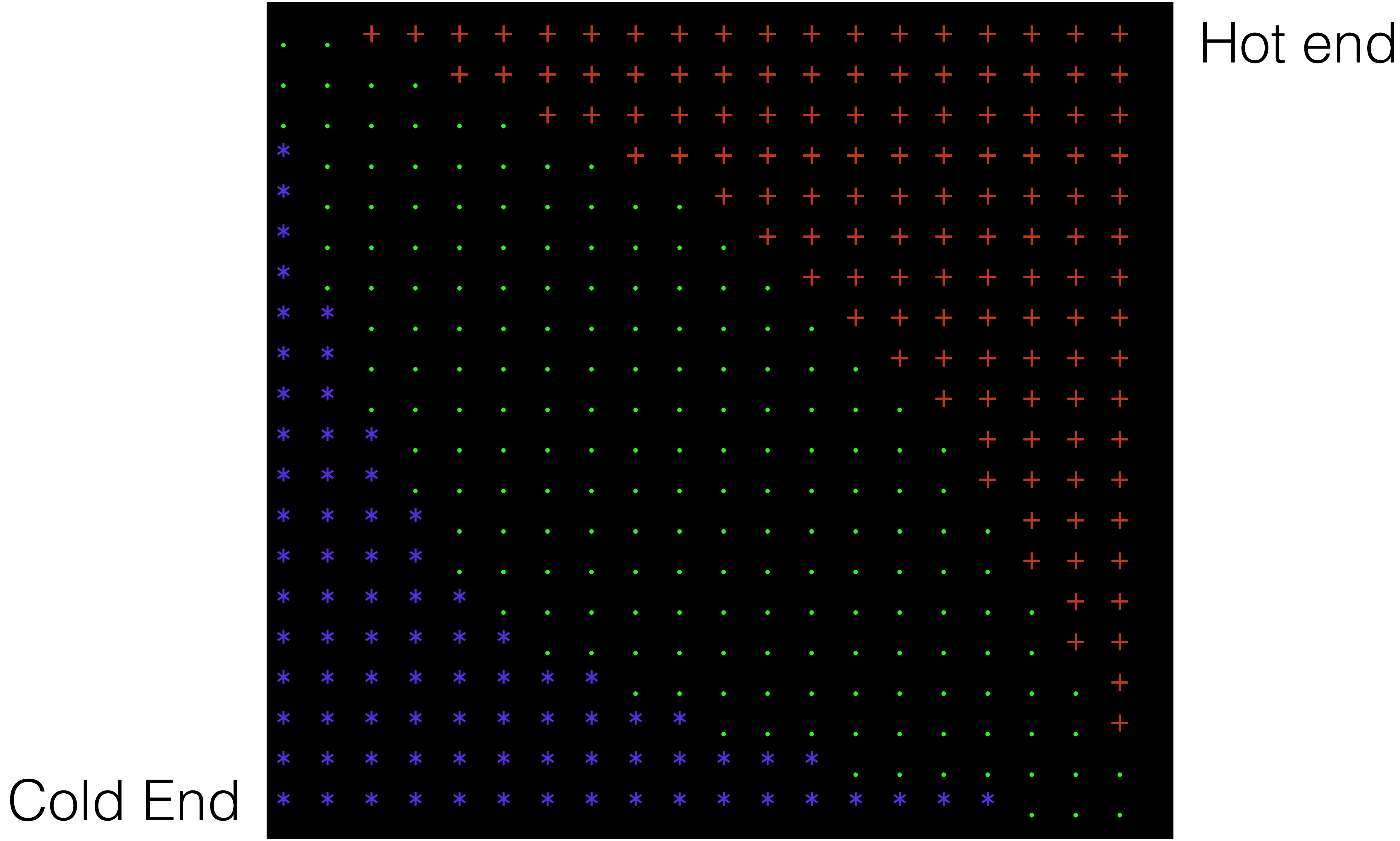
```
AEIMBFJNCGKODHLP
```

```
AEIM
BFJN
CGKO
DHLP
```

- Correct answer once carriage returns put in
- Can get code to write in own carriage returns, but messier

Case Study - MPI-IO

Result



Case study

- Had solution to heat equation that worked on multiple processors
- Uses MPI Types for sending and receiving
- Now change to write output file rather than display to screen
- Same general approach as for characters
- Is now a complication that we have guard cells that we don't want to write into the file

Types for case study IO

```
//Now create the types used for MPI_IO
//First, represent the main array without it's guard cells
sizes[0] = nx_local + 2; sizes[1] = ny_local + 2;
subsizes[0] = nx_local; subsizes[1] = ny_local;
starts[0] = 1; starts[1] = 1;

create_single_type(sizes, subsizes, starts, &type_no_guard);
```

- Create type representing local array shorn of guard cells
- sizes = (nx_local + 2, ny_local + 2)
- subsizes = (nx_local, ny_local)
- starts = (1, 1)

Types for case study IO

```
//Now represent the part of the global array that is represented
//on this processor
sizes[0] = nx; sizes[1] = ny;
subsizes[0] = nx_local; subsizes[1] = ny_local;
//Minus 1 because rest of code used Fortran like 1 based arrays
//MPI ALWAYS uses C style 0 based
starts[0] = x_cell_min_local -1; starts[1] = y_cell_min_local - 1;

create_single_type(sizes, subsizes, starts, &type_subarray);
```

- Create type representing local subsection of global array. Does not include ghost cells!
- sizes = (nx, ny)
- subsizes = (nx_local, ny_local)
- starts = (x_cell_min_local-1, y_cell_min_local-1)
- "-1" in starts because we're using 1 based arrays and we want an offset

Opening the file

```
MPI_File_delete("out.dat", MPI_INFO_NULL);
MPI_File_open(cart_comm, "out.dat", MPI_MODE_WRONLY + MPI_MODE_CREATE,
    MPI_INFO_NULL, &file_handle);
```

- Exactly as in the simple code
- Delete the old file
- Open the new one for creation

Writing the data

```
//Subroutine to write the output file
//Notice that this is called on all cores
//unlike the output to screen
void output_to_file(grid_type * data)
{
  MPI_File_set_view(file_handle, offset, MPI_FLOAT, type_subarray,
      "native", MPI_INFO_NULL);
  MPI_File_write_all(file_handle, data->data, 1, type_no_guard,
      MPI_STATUS_IGNORE);

  //Shift the offset by the amount of data written
  offset = offset + (nx * ny * sizeof(float));
}
```

- Here, we're only opening the file once, but writing to it every output cycle
- Not a very general approach, but works here

Writing the data

```
//Subroutine to write the output file
//Notice that this is called on all cores
//unlike the output to screen
void output_to_file(grid_type * data)
{
  MPI_File_set_view(file_handle, offset, MPI_FLOAT, type_subarray,
      "native", MPI_INFO_NULL);
  MPI_File_write_all(file_handle, data->data, 1, type_no_guard,
      MPI_STATUS_IGNORE);

  //Shift the offset by the amount of data written
  offset = offset + (nx * ny * sizeof(float));
}
```

- Note that “type_subarray” is used in MPI_File_set_view
- “type_no_guard” is used in MPI_File_write_all to clip off the guard cells before writing
- Works just like using MPI types when sending and receiving
- Data is reshaped to match

Writing the data

```
//Subroutine to write the output file
//Notice that this is called on all cores
//unlike the output to screen
void output_to_file(grid_type * data)
{
  MPI_File_set_view(file_handle, offset, MPI_FLOAT, type_subarray,
      "native", MPI_INFO_NULL);
  MPI_File_write_all(file_handle, data->data, 1, type_no_guard,
      MPI_STATUS_IGNORE);

  //Shift the offset by the amount of data written
  offset = offset + (nx * ny * sizeof(float));
}
```

- Note that offset is incremented by “nx * ny * sizeof(float)” each time
- This means that the next output is written after the current one
- Can’t just rely on file pointer, because MPI_File_set_view resets it

Reading the file

- File is a normal binary file can be read by Python/Matlab, whatever
- But for testing purposes, want ASCII art back
- Almost exactly the same
- Create same types (in theory, don't need the guard cells for visualising or their associated types, but imagine that you're restarting your code rather than visualising)
- Just MPI_File_read_all rather than MPI_File_write_all

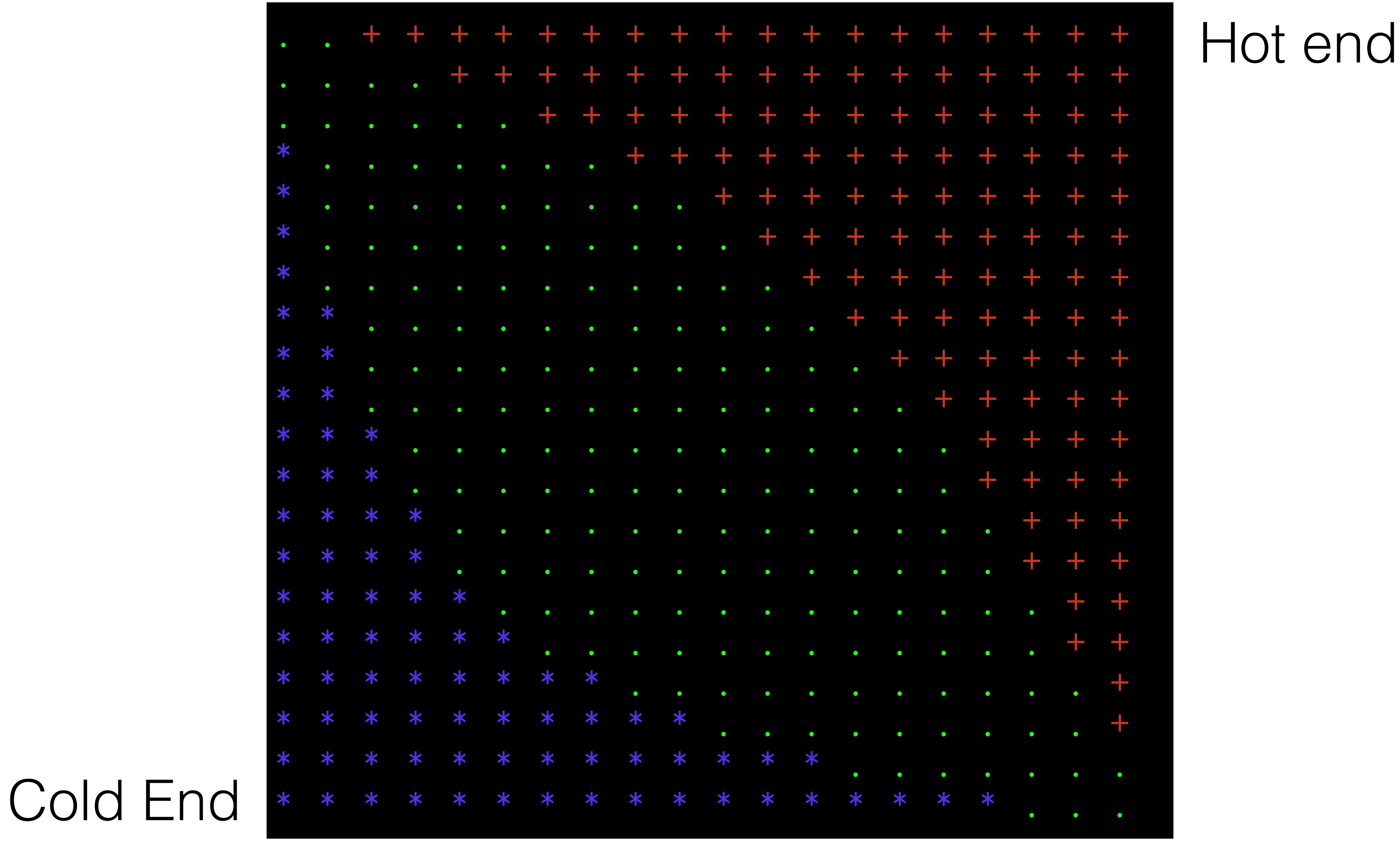
Reading the data

```
//Subroutine to write the output file
//Notice that this is called on all cores
//unlike the output to screen
void input_from_file(grid_type * data)
{
  MPI_File_set_view(file_handle, offset, MPI_FLOAT, type_subarray,
      "native", MPI_INFO_NULL);
  MPI_File_read_all(file_handle, data->data, 1, type_no_guard,
      MPI_STATUS_IGNORE);

  //Shift the offset by the amount of data written
  offset = offset + (nx * ny * sizeof(float));
}
```

- Very, very nearly identical to writing code
- Run “input_from_file” every time to get data back from the file
- Then use the old visualisation routines

Result



Notes

- File reading code can be run on different number of cores to file writing code
- All works seamlessly
- Doesn’t keep any information indicating that array was *ever* split up
- If you want that information then have to write it into your file yourself