NTNU – Trondheim
Norwegian University of
Science and Technology

Cartesian communicators & Parallel I/O

Jan.Christian.Meyer@ntnu.no

Today’s topic

- Today, we will look at
  - How to arrange ranks in a rectangular grid
  - How to write data in one file simultaneously (*i.e.* parallel I/O)
  - How to simultaneously write rectangular data in one file
- There isn’t “really” a connection, **BUT**
  - Rectangular data sets often go hand-in-hand with rectangular grids of ranks
  - Writing array slices in a file is the kind of operation you really don’t want to fix up yourself with creative indexing
  - Most of us would be debugging that sort of maneuver until our grandchildren graduate

Cartesian communicators

- These are like a special case of the graph thing from last time, but it’s such a common case
  - They arrange ranks into a regular array with neighbors to the left/right (1D), up/down (2D), in/out (3D), *etc. etc.*
- The index/edge lists for this kind of graph become so regular that it would be a waste to write them out
  - Each rank has 2 direct neighbors in each direction
  - My neighbor’s neighbor is my own 2nd neighbor
  - ...and so on

Creating a Cartesian communicator

- Like the graph communicator, it starts from another communicator, and just arranges all the ranks
- It needs some lists of integers as well:

```
int MPI_Cart_create (
        MPI_Comm old_communicator,
        int number_of_dimensions,          ← Length of the next few lists
        const int dims[],                  ← Size in each of n directions
        const int periods[],               ← Yes/no (1/0): wrap the edges?
        int reorder,                       ← Same as for graph comms.
        MPI_Comm *new_communicator         ← Result
);
```

Ingredients of a 2D communicator

- Say that we have 12 ranks
  - That can work out to a 4x3 grid, for example
- Let it end at the edges of the grid, and keep the rank numbering

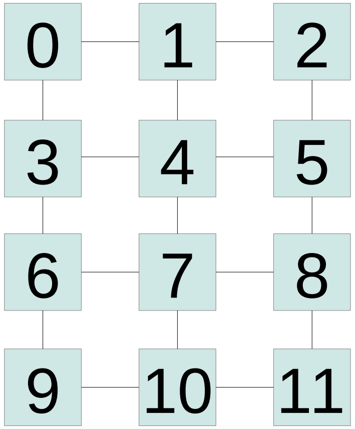
  ndims = 2

  dims[2] = { 4, 3 }

  periods[2] = { 0, 0 }

  reorder = 0
- Here's what we get:

  (typically)



0	1	2
3	4	5
6	7	8
9	10	11

Periodicity

- If we do this instead

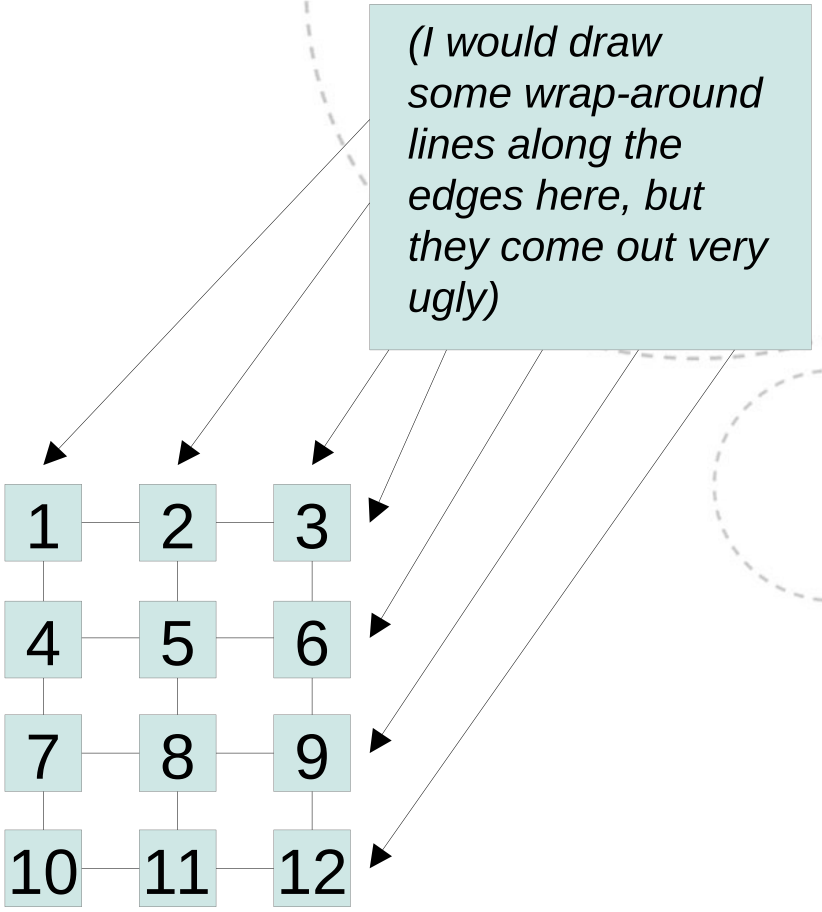
  ndims = 2

  dims[2] = { 4, 3 }

  periods[2] = { 1, 1 }

  reorder = 0

  – 3 will also be left-neighbor of 1
  – 10 will be up-neighbor of 1
  – 2 will be down-neighbor of 11
  – 7 will be right-neighbor of 9
  – …
  – You get the picture



1 2 3
4 5 6
7 8 9
10 11 12

(I would draw some wrap-around lines along the edges here, but they come out very ugly)

Dimensions of the grid

- When you start the program, all you have is a count of how many ranks are in the world comm.
  - For 2D, we’ll want to factor it into AxB with A and B each as close to the square root of the rank count as possible
  - For 3D, we’ll want to factor it into AxBxC with A, B, C as close to the cube root of the rank count as possible
  - For 4D, … and so on, and so forth
- You can write this by hand if you must
  - You don’t have to, MPI_Dims_create does it:

```
int MPI_Dims_create (
    int number_of_nodes,          ← rank count
    int number_of_dimensions,     ← how many directions do we need?
    int dimensions[]              ← result: array of rank counts in each direction
);
```

Finding my place

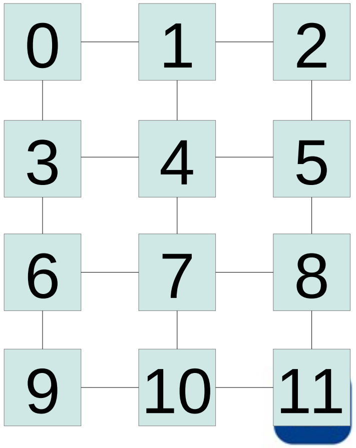
- A rank can find its position in the grid

```
int MPI_Cart_coords (
        MPI_Comm cartesian_communicator,    ← our grid
        int rank,                           ← me
        int dims,                           ← nr. of dimensions
        int coords[]                        ← result: list of coordinates
);
```

- In our example,

```
int coords[2];
MPI_Cart_coords ( cart, 7, 2, coords );
```

returns the 2D pair { 2, 1 } in ‘coords’



0	1	2
3	4	5
6	7	8
9	10	11

Finding my neighbors

- By choosing one of the directions and an offset

  (positive or negative)

  we can find out who’s at our own coordinates plus/minus the offset:

```
int MPI_Cart_shift (
    MPI_Comm cartesian_communicator,        ← the grid
    int direction,                          ← which axis to shift
    int displacement,                       ← how far to shift
    int *rank_source,                       ← rank to recv from
    int *rank_destination                   ← rank to send to
);
```

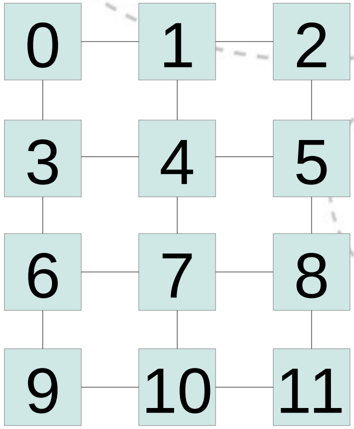
- The idea with the “source” and “destination” is that sending/receiving local data there will shift everyone’s view of global data by “displacement” positions

The example again

- Assuming that this is periodic,

  direction = 1 (x-axis, y is 0)
  displacement = 1 (immediate higher neighbors)

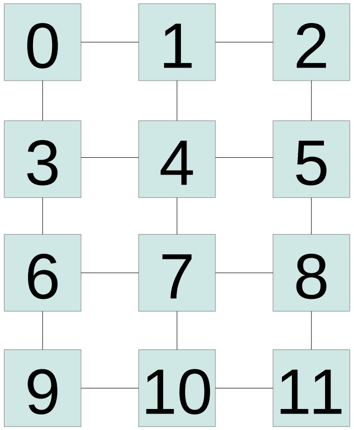
  gives
  source = 6, destination = 8
  on rank 7



0 1 2
3 4 5
6 7 8
9 10 11

- If you put them into source and dest arguments in an MPI_Sendrecv, distributed data will move from left to right by 1 grid position, along the x-axis

One nugget of wisdom



- If you have a non-periodic cartesian communicator, the neighbors that are “off-grid”

  (*e.g.* the right neighbor of rank 5)

  will come back as MPI_PROC_NULL from Cart_shift
- If you feed MPI_PROC_NULL into a communication call where it expects a rank, the matching operation will simply not be carried out
  - Using this mechanism can save you from writing extra code that tests whether or not a rank is at the edge of the communicator when you’re doing border exchanges / boundary conditions
  - Statements that begin with “if (coords[0] == 0)” and similar usually aren’t necessary

Storing distributed data

- Ken Batcher (who worked at Goodyear Aerospace at the time) has semi-famously said:

  “A supercomputer is a device for turning compute-bound problems into I/O-bound problems”

- This is painfully true, suppose that we
  - Have some parallelizable computation for 1000x1000x1000 points
  - Parallelize all the computation perfectly
  - Regularly write the 1000x1000x1000 points to a file, one point at a time

- Knowing Amdahl’s law, we can tell that the total won’t be able to run faster than the file-saving mechanism

Standard I/O is sequential

(kind of)

- It doesn’t *have* to be, but tradition has made it so
  - I can personally remember saving data on magnetic tape
  - It had to wind the tape forward to read the next data element
  - This means that in order to insert something in the middle of a file, you have to push the whole rest of the file further forward by copying it there
- Operating systems mostly still treat files this way
  - We have system calls to open files for reading/writing at the start
  - We have system calls to read/write “the next n bytes”
  - We have system calls to *append* another n bytes
  - <stdio.h> offers very little in the way of random access

Sequential file systems

- Even if you allow random access in files, the vast majority of computers will only have a single large storage medium to write it with anyway
- If the O/S floods it with simultaneous write requests, it will go through them one at a time
- Hence, the file systems we format the media with also tend to behave with a tape-like memory model

Parallel file systems

- Large computing clusters divide the implementation of storing things into *data* and *metadata* operations
  - Data operations store blocks of 1s and 0s as the contents of the file
  - Metadata operations store the file system's hierarchy of directories and file names
- Parallel arrays of separate disks handle both
- If we can do one inexpensive metadata operation to open a file, its content blocks don't have to reside on the same physical disk
  - Hooray, they can be written separately and simultaneously!

MPI_IO facilitates this

- All ranks can open the same file simultaneously
- They can each set a *view* of the file, creating a window where they can write without touching other ranks’ data
- They can all write within their own views at the same time

- If you time parallel I/O on your own computer, you probably won’t get much speedup
  - because of the single disk thing
- If you can write and debug parallel I/O on your own computer, you can get substantial speedup when running the same program on a machine with parallel disk arrays (and the right file system)

NTNU – Trondheim
Norwegian University of
Science and Technology

Opening and closing

- MPI’s own file handles are needed
  - They’re called MPI_File
- We can open them with a collective call:

```
int MPI_file_open (
    MPI_Comm communicator,
    const char *filename,      ← Text string with the file name
    int access_mode,           ← Combination of flags from MPI
    MPI_Info info,             ← Additional hints about the file (not oblig.)
    MPI_File *result           ← The open file handle
);
```

- We can close them too:

```
int MPI_File_close ( MPI_File *handle );
```

Access mode flags

- These are the placeholders for the usual “r” (read), “w” (write), “a” (append) and such that go into the system’s native ‘fopen’ call
  - MPI_MODE_CREATE makes a new file if it didn’t exist from before
  - MPI_MODE_WRONLY signals that we’ll only be writing to it
  - MPI_MODE_RDONLY signals that we’ll only be reading from it
  - MPI_MODE_RDWR allows both
  - MPI_MODE_APPEND signals that we’ll be adding data at the end
  - *Etc. etc.*
- The value of the access mode argument can be a logical-or combination of appropriate flags

Be careful with over-writing

- Regular fopen in write mode will annihilate an already existing file, and start a new one with the same name
- MPI_File_open with write access will not
  - Since we're expecting to put pieces of the file in at arbitrary moments in the future, it would be hard to say which rank should be responsible for obliterating the previous contents
  - Very bad if it happens after another rank already wrote something
  - Waiting for all the ranks to arrive at File_open introduces some of the synchronization requirement we're hoping to eliminate in the first place
- Remove your parallel files between runs to make sure that they only reflect your most recent output
  - If you 'overwrite' a larger file with a smaller one, it'll have a tail full of values that you didn't generate on this particular run

Explicit positioning

- If you can work out the positions for each data chunk yourself, you can divide the file size yourself too

```
int MPI_File_write_at (
    MPI_File filehandle,        ← File to write
    MPI_Offset offset,          ← Where to write in it (different on each rank)
    const void *buffer,         ← What to write in it (your local data part)
    int count,                  ← How many elements?
    MPI_Datatype type,          ← What type of elements?
    MPI_Status *status          ← Status object that we can ignore as usual
);
```

- You are in charge of ensuring that no two ranks write in the same position
  - If they do, the result can be anything

Views

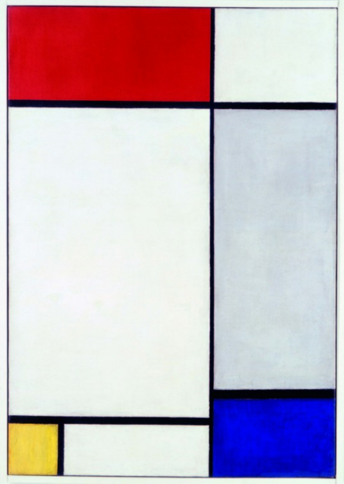
- We can also restrict the area a rank will write in to shape it like an MPI_Datatype

They’re just address layouts, as discussed previously:

```
int MPI_File_set_view (
    MPI_File *filehandle,              ← File to work on
    MPI_Offset *displacement,          ← Where to start (in case it has a header)
    MPI_Datatype element_type,         ← What type of things to read/write
    MPI_Datatype file_layout,          ← What region of the file to access
    const char *representation,        ← Low-level details – “native” takes O/S default
    MPI_Info info                      ← Additional file information we won’t need
);
```

Putting it to use

- Here is Dutch abstract artist Pieter Mondriaan’s 1927 painting *“Composition with Red, Yellow, and Blue”*
- Regardless of how you feel about modern art, it does contain a variety of rectangular subdomains
- We can approximate it with MPI
  - Today’s example code runs only with 7 ranks: one for each region
  - Approximate coordinates of the picture’s regions are hard-coded by eye measure
  - Each rank allocates a local array and fills it with the right color
  - In the end, they collectively write their regions into a file containing 2D image data



Step 1:

Look up coordinates and color

- There's an array of 5 colors, encoded as RGB triplets
- There's an array of 7 sections, each containing
  - y/x origin coordinates
  - height/width size of the section
  - Index for one of the 5 colour triplets
- Each rank
  - Picks the section indexed by its rank
  - Allocates a matching (local) memory region
  - Fills it with the right colour triplet

Step 2:
Commit data types

- There’s one for contiguous 3-tuples of bytes
  - To represent one pixel
- There’s one for a contiguous array of pixels
  - To represent the locally stored part of the whole
- There’s one for a matching sub-array of pixels as indexed into the global coordinate space
  - To create file views that delimit where each rank will write its data

Step 3:

Combine the whole array

- After setting up the coordinate translation, all that’s left to do is to issue a collective write call, and the file comes together
- It comes out as a file ‘composition.data’, which just contains the global pixel array
- There’s a Makefile target that adds a header, to make it a viewable (PPM) image

```
make composition.ppm
```

Why are we doing this again?

- Cartesian data splits add
  - more multidimensional indexing
  - more neighborhood-problems to figure out
  - more debugging and headaches
- There are two really good reasons to bother
  - 1D data splitting of 2D data limits your global problem size by the biggest 1D section that can fit on a rank
  - Cartesian subdomains have the shortest possible border circumferences for their size

MPI in summary

- We have covered
  - Point-to-point communication (modes, non-blocking, persistent)
  - Border exchanges (in 1D and 2D)
  - Collective operations (barriers, broadcasting, reduction, scatter, gather)
  - Derived data types (structured, contiguous, indexed, vectors, subarrays)
  - Communicators (arbitrary groups, graphs, Cartesian)
  - Parallel I/O (explicitly indexed, views)
- All in all, this toolkit is enough to let you write some programs that are *almost* entirely parallelized
- That's probably enough for now
  - We can go from the distributed memory of MPI to the shared memory of pthreads next