NTNU – Trondheim
Norwegian University of Science and Technology

MPI-IO

Guest lecture for TDT4200

Jan Christian Meyer 11.09.2024

Our plan for today

- Introduction to the world's simplest image format
  - because we need some data to work with
- A short review of derived types in MPI
  - because we need some work to do on the data
- A demonstration of how to handle files in parallel
  - because that's what MPI-IO does

The world's simplest image format

- PGM (Portable Gray Map) files have the following structure:
  - The file begins with one line of text, which contains
    `P5 <width> <height> <max>`
  - *width* and *height* indicate the size of the image
  - pixel values should go from 0 through *max*
  - The rest of the file is just an array of pixel values in row-order
- If we just let *max=255*, then pixels fit in (unsigned) bytes
  - Here is an example:

```
#include <stdio.h>
int main () {
    FILE *out = fopen ( "my_image.pgm", "wb" );
    fprintf ( out, "P5 640 480 255\n" );
    for ( int i=0; i<640*480; i++ ) fputc ( (i/480)%256,
out );
    fclose ( out );
}
```

Why are we doing this?

- It’s a super-duper easy way to get a visual result
  - You now know how to create greyscale graphics in every programming langue where you can write bytes in a stream

- Sadly, not every image viewer handles pgm files
  - *eog, display, gimp,* and many others do, though
- Luckily, *ImageMagick* handles it just fine
  - After a quick
    `convert my_image.pgm my_image.jpg`
    or similar, you can inspect it with practically any viewer

Derived types review

- When moving data with MPI, the MPI_Datatype is usually MPI_INT, MPI_DOUBLE, MPI_CHAR, …
- You can make your own data types from combinations of these built-in primitive ones:

```
MPI_Datatype my_type;
MPI_Type_contiguous ( 32, MPI_INT, &my_type );
MPI_Type_commit ( &my_type );
```

  - Hey presto, now we have a data type that contains 32 consecutive integers
  - It’s always polite to clean up after yourself:

```
MPI_Type_free ( &my_type );
```

Derived types can be parts of derived types

```
MPI_Datatype my_other_type;
MPI_Type_contiguous ( 64, my_type, &my_other_type );
MPI_Type_commit ( &my_other_type );
```

- Now we (also) have a type to contain 64*32=2048 consecutive integers
- If you're making some big, complicated type from many different sub-types, you only have to `commit` and `free` those you actually plan to use

NTNU – Trondheim
Norwegian University of Science and Technology

What's the point?

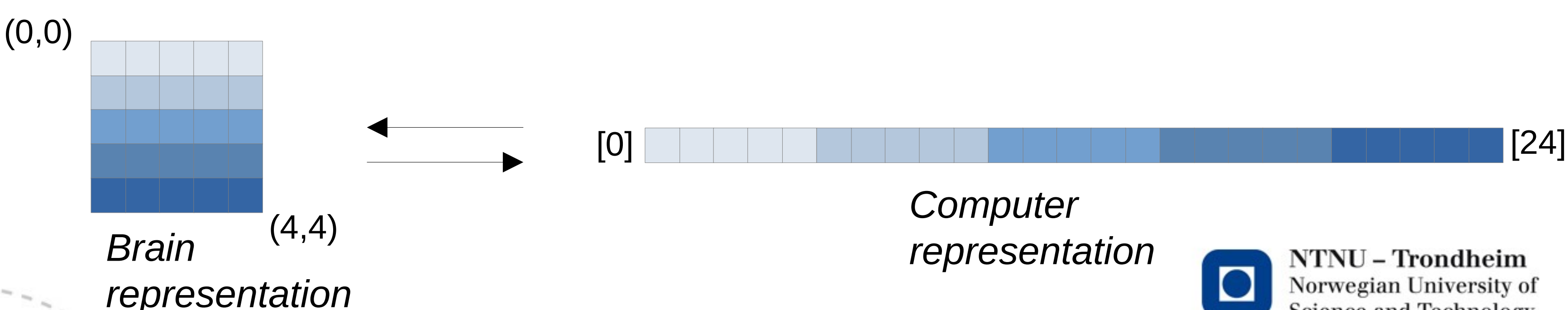
- MPI_Type_contiguous is not really so fascinating:

```
MPI_Send ( buf, 1, my_other_type, dst, 0, MPI_COMM_WORLD );
```

  is pretty much the same thing as

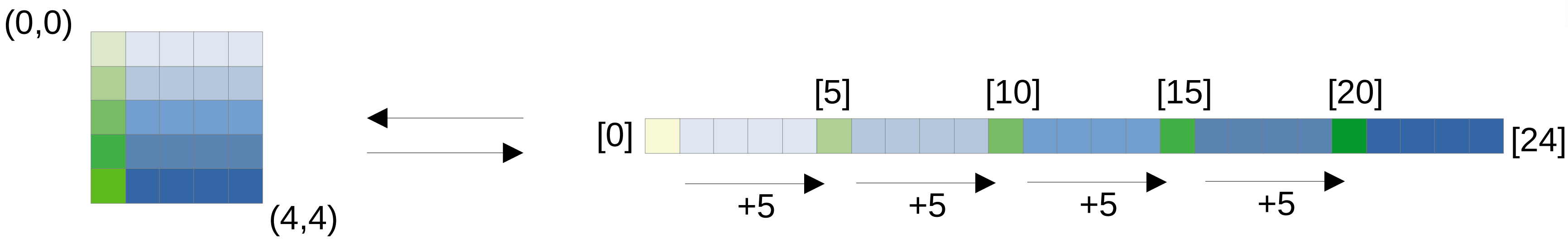
```
MPI_Send ( buf, 2048, MPI_INT, dst, 0, MPI_COMM_WORLD );
```

- Things get more interesting when we can put space between the elements in a type
- Consider this 2-dimensional array (and its memory layout)



Accessing a column

- Rows are easy, their elements are consecutive in memory
- If we want a column of data instead, it's spread out with spaces in between:



- *MPI to the rescue:*

```
MPI_Type_vector ( 5, 1, 5, MPI_INT, &column );
```

  - If you send this type from the buffer address of (0,0), it will transmit the first column
  - If you send from the address of (0,2), it will transmit the third
  - It's just a recipe for sending evenly spaced elements

What's in a vector type?

- A *count*, *blocklength*, and *stride*:

```
MPI_Type_vector ( 5, 1, 5, MPI_INT, &column );
```

Count
How many blocks to send?

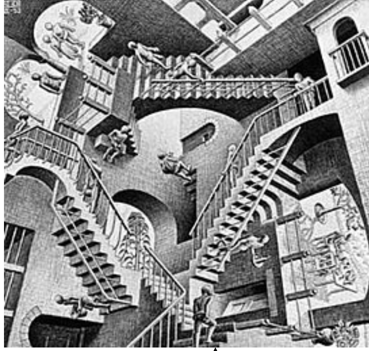
Blocklength
How many elements in each block?

Stride
How many elements are there between the beginnings of blocks?

- This will suffice to create an example.
  - I'm afraid it will be a poorly programmed example
  - Please bear with me, I'm just trying to make one specific point

Demonstration time

Code in today's archive: 01_transfer



- We're using a pgm file with M.C. Escher's famous 1953 litograph "Relativity" in it
  - I have made the pgm file square (1330x1330 pixels) for simplicity
- Our first version of the code uses two ranks only
  - Rank 0 and 1 both allocate an array
  - Rank 0 reads the image into its array
  - Rank 0 sends the image to rank 1, line by line
  - Rank 1 writes a copy of the image it received into a file
- For now, it's just an elaborate way to copy a file
  - All I want is to pass our array through a pair of Send/Recv calls

We can use vectors for lines

Code in the archive: 02_vectors

- This is the same as before, but we've replaced

```
MPI_Send (
    &IMAGE(y,0), image_size[1], MPI_UINT8_T, dest, tag, comm
);
```

with a vector that contains image_size[1] consecutive bytes

```
MPI_Type_vector ( 1, image_size[1], 1, MPI_UINT8_T, &image_line );
...
MPI_Send ( &IMAGE(0,0), 1, image_line, dest, tag, comm );
```

- Splendid, but that's just an *even more* elaborate way to make a copy…?

Derived type magic

Code in the archive: 03_transpose

- Let's try to commit *different* vector types on ranks 0,1

R0: `MPI_Type_vector ( 1, image_size[1], 1, MPI_UINT8_T, &image_line );`

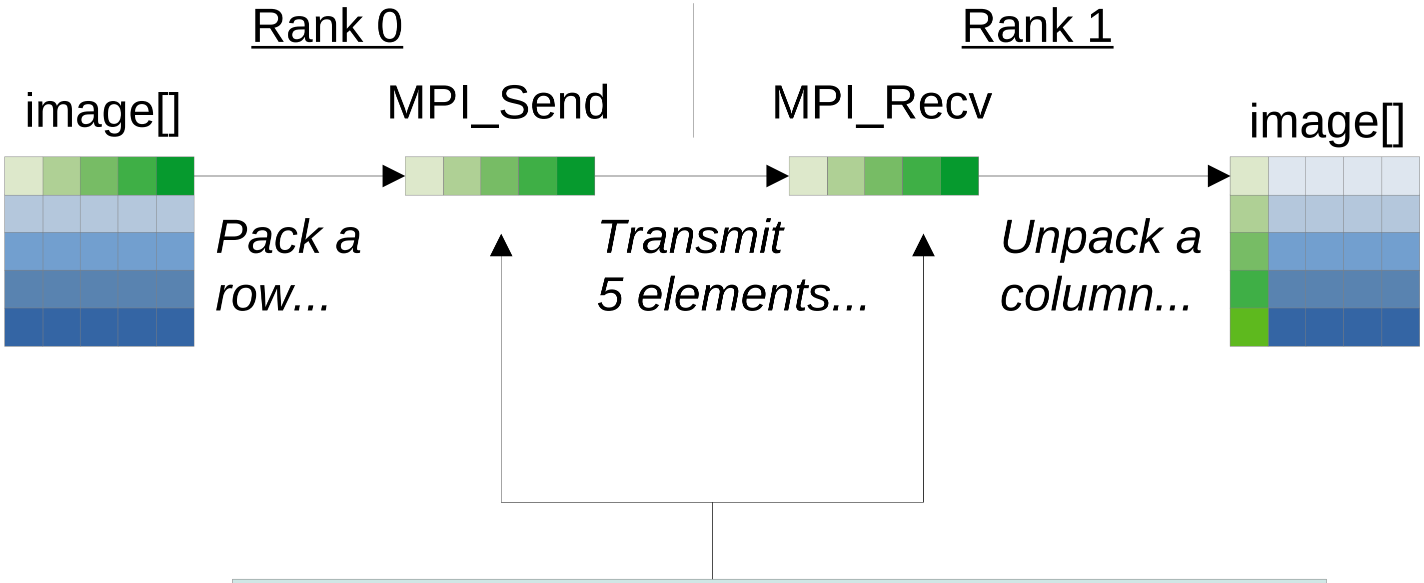
R1: `MPI_Type_vector ( image_size[0], 1, image_size[1], MPI_UINT8_T, &image_line );`

  - To rank 0, 'image_line' means a row
  - To rank 1, 'image_line' means a column
  - Since the image is square, rows and columns contain the same # of bytes

- *Abracadabra*, rank 1's copy of the image has been flipped!
  - What is happening?

How it works

- A derived type is nothing more than a recipe for how to space array elements in memory:



Sending and receiving only requires these two buffers to be equally long

This is not a hack

it's a feature

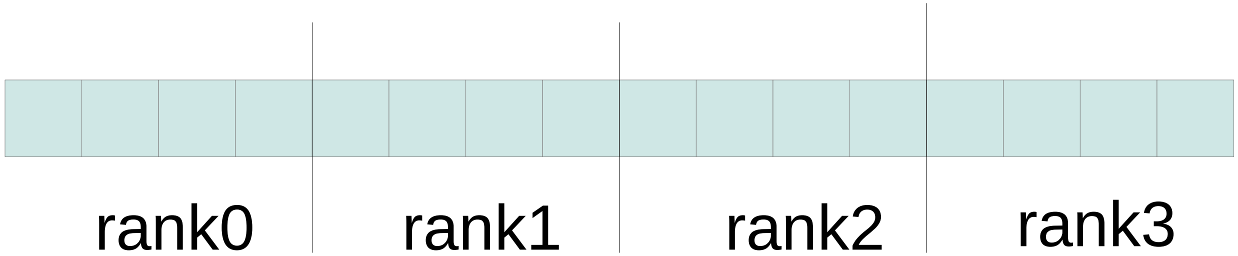
- If you want to do the same thing without any MPI_Datatype:
  - Rank 0 must
    - Allocate an extra array with space for a line
    - Write a loop that copies a row into that array
    - Send the array to rank 1
    - Clear out the extra array after sending
  - Rank 1 must
    - Allocate an extra array with space for a line
    - Recv the array from rank 0
    - Write a loop that copies data out of the array and into a column
    - Clear out the extra array after receiving and copying the contents
- The whole idea with MPI_Datatype is to save you from having to sort everything into (and out of) linear orders by hand

Standard I/O is sequential

- Files have a lot in common with arrays in memory
  - They start at some index 0, and then you find your data some number of bytes offset from there
  - Arguably, they *are* arrays in memory, it's just that they're in a slow type of memory
- The POSIX standard I/O library kind of expects it the memory be a roll of magnetic tape
  - (FILE *) tracks a 'position' in the file you open
  - fwrite, fread, and friends write at the present location
  - fseek lets you do random access, but with the expectation that the tape robot has to wind the tape to the correct location
- Magnetic disk drives have almost the same limitation
  - They just seek a lot faster

Parallel I/O

- In random-access memory, we get things to go faster by distributing parts of arrays:



- This would be nice to have for files also
  - Otherwise, your parallel computation will just go serial when you want to save the result
  - That creates a bottleneck, *cf.* Amdahl's law

MPI-IO

- MPI has collective operations (similar to bcast, reduce, *etc.*) that allow us to do this
- They require their own type of file handles, because the regular ones come with just 1 ‘current position’
- MPI_File has no such limitation
  - MPI_File_open
    MPI_File_close

    do exactly what you would expect them to, but they manipulate parallel-friendly file handles

A simple case in 1D

Code in the archive: 04_array_split

- This program demonstrates opening, closing, and simultaneously writing equally large data lumps in a file
- MPI_File_write_at_all is collective
  - There is also an MPI_File_write_at which doesn't require everyone to call it at once, but still admits simultaneous reading/writing in different parts of the file
- The programmer is in charge of making sure that reading/writing doesn't create dependencies
  - Let two ranks write in the same place at your own peril

NTNU – Trondheim
Norwegian University of Science and Technology

Beware of zombie results

- With sequential I/O, opening a file for writing removes the previous version entirely, and starts a fresh file
- MPI-IO can't do that, because no rank can know for sure that another one hasn't already written in a shared file by the time it opens the handle
- Consequence:
  - If you run your program once and produce 10MB of data in a new file, it'll work as you expect
  - If you run the same program again, but now you only write 5MB into the same file, the last 5MB in that file will be "leftovers" from the first run
  - Delete your files between runs to avoid confusion

NTNU – Trondheim
Norwegian University of Science and Technology

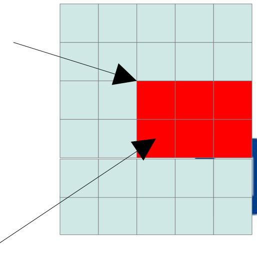
Views

- To support us with calculating offsets and sizes that don't overlap, MPI file handles let you set a *view* that restricts where each rank will read/write
- It's just an association between an MPI_File and an MPI_Datatype
  - If we commit different layouts as data types on each rank, they can each have different windows into the file
  - Handy for writing non-contiguous data (such as our column vectors from before) without having to write loops that write a little, jump a little forward, write a little more...

NTNU – Trondheim
Norwegian University of Science and Technology

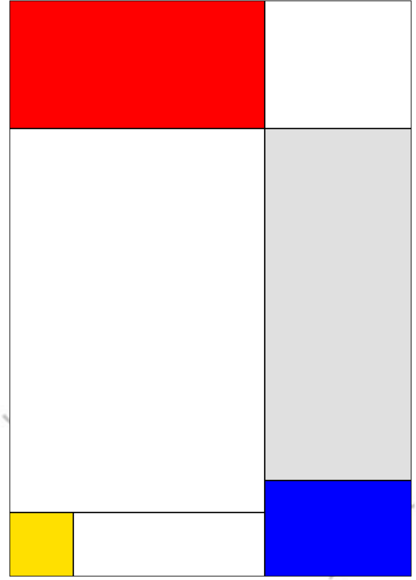
Subarrays

- MPI_Type_create_subarray lets you define ranges in multi-dimensional data without having to lay it out sequentially
- Subarrays have 4 ingredients:
  - Number of dimensions (N)
  - Size of the entire (global) array we're slicing (N coordinates)
  - Size of the slice (N coordinates)
  - Origin of the slice (N coordinates)



Slightly fancier case in 2D

Example in code archive: 05_array_split_2D



- The example program runs with exactly 7 ranks
  - Bad practice, but it's just an example
- Each of them defines a subarray for one of seven color sections in a 2D surface
  - The results come out as another PPM image, like this one *(it's a facsimile of a famous painting)*
- The code is a little more complicated than our previous examples
  - I strongly recommend that you try to write a *sequential* program that produces the same result
  - That will make it abundantly clear why MPI-IO is the way it is

A note about speedup

- MPI-IO support on your laptop is most useful for developing and debugging correct code
- 4 cores writing in parallel on a machine like that probably won't get any speed improvement
  - Your file system probably only covers 1 physical storage device
- Parallel I/O actually does give you speedup on a large parallel cluster, though
  - Its file system is distributed across lots of physical storage devices