NTNU – Trondheim
Norwegian University of
Science and Technology

The advection equation using six-function MPI

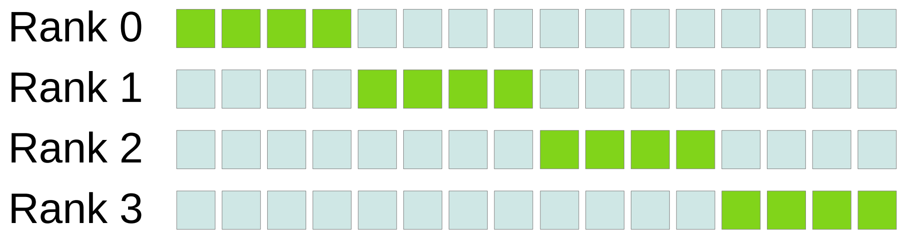
Jan.Christian.Meyer@ntnu.no

The example code archive

- I put another copy of the sequential advection code into “02_advection_sequential”
- It’s pretty much the same thing as before, but I have
  - Divided the code into (hopefully) aptly named functions, and
  - Increased the problem size in order to make it run for a little bit
  - If you want to adjust the size yourself, mind that different sizes of output files require a modification to the plotting script, the number of elements to plot is hardcoded in there
- The remaining two directories contain
  - A partial version that can only initialize the program and save its state
  - A complete version that includes the numerical solver loop

We need to split up the work

- We have a long, linear array, and some number of workers to employ
  - Let’s draw 4, just as an example
- Here’s a popular, but not-so-good solution
  - Allocate full array for everyone, but just work on part of it



  - Add up everyone’s partial solutions at the end

Why would anyone do that?

- It’s really simple to work out array indices when everyone has the same coordinate space
  - You’ll see in a minute
- It’s still not a great idea, though
  - It limits the maximum problem size to the amount of memory 1 rank can allocate
- I am not going to say that it’s bad in every context
  - Small problems are also worth solving
- I <u>am</u> going to say that it’s an impediment to scalability
  - So there

Another way to split work

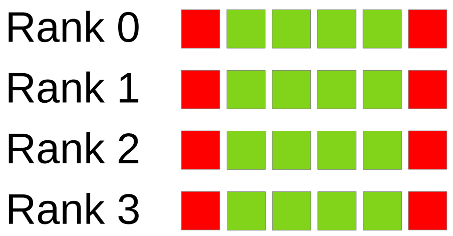
- Divide the problem size by the rank count, and allocate separate parts

Rank 0	🟩🟩🟩🟩	Global elements 0-3, local indices 0-3
Rank 1	🟩🟩🟩🟩	Global elements 4-7, local indices 0-3
Rank 2	🟩🟩🟩🟩	Global elements 8-11, local indices 0-3
Rank 3	🟩🟩🟩🟩	Global elements 12-15, local indices 0-3

- We get to concatenate these when saving the state of the entire domain

From the boundary line

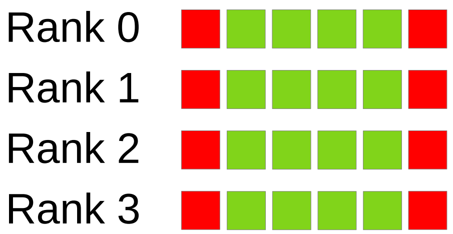
- The way we allocated/indexed the array in the serial version, we added 2 extra points for the boundary condition
  - ...and gave them indices -1 and N…
- Let's do that everywhere here, too



- Only U(-1) by rank 0 and U(4) by rank 3 will *actually* represent the problem domain's boundary
- We'll have use for the others as well

Initialization

- Since we’ve split the coordinates,
  - Rank 1 must know that its element 0 is global element 4
  - Rank 2 must know that its element 0 is global element 8
  - *Etc.*



- One possible solution:

```
int_t my_origin = rank * (N / size);
```

- This requires all parts to be equally large

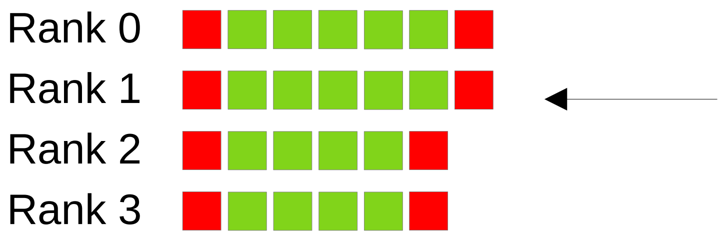
Initialization

- What if the domain size isn't divisible by the number of ranks?
- There are three schools of thought:
  - Stop the program, and demand a particular problem-size / rank count relationship
  - Give the last rank less work (either in a smaller allocation, or padding out the domain data with zeros at the end)
  - Give 1 extra element to a suitable subset of the ranks

Initialization

This week's example code goes for the last option:

```
local_sizes = malloc ( size * sizeof(int_t) );
for ( int_t r=0; r<size; r++ )
    local_sizes[r] = (int_t)( N / size ) + ((r<(N%size)) ? 1:0);
```



(Example: size 18 problem with 4 ranks)

- The result is that every rank gets an array with the others' subdomain sizes in it
  - Per the illustration, local_sizes would contain [5, 5, 4, 4]

Initialization

- Armed with the knowledge of how big the preceding problem parts are, each rank can calculate what its own origin index corresponds to globally

```
int_t my_origin = 0;
for ( int_t i=0; i<rank; i++ )
    my_origin += local_sizes[i];
```

- A small amount of extra typing, but the code only has to run once, and it's very short
- Now that we can calculate x-positions from the indices, we can plug in the function that creates the initial advection state
- Each rank can set up its part separately

PROTIP:

A picture tells a 1000 words

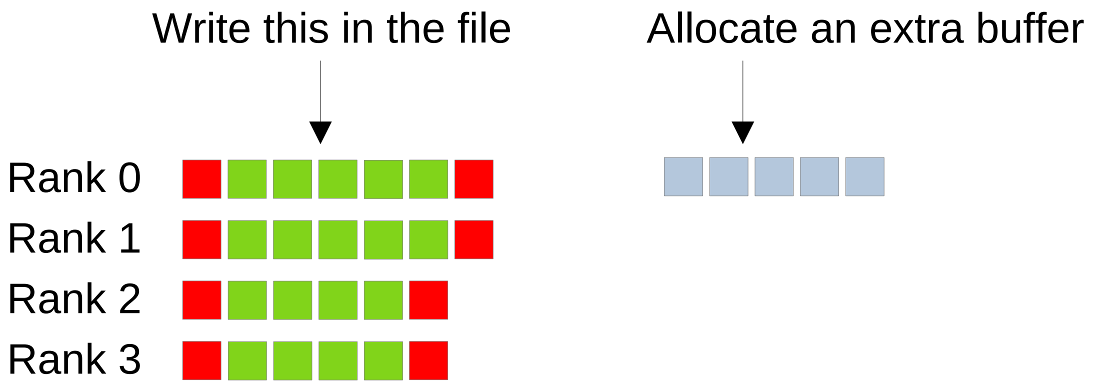
- The first thing I do when starting a parallel program is to invent a way to draw pictures of the global state
  - Parallel programs run in a mish-mash order that can be different every time you launch the program, so debugging with print statements gets messy
  - It’s more feasible if you make every rank write in a separate file, but that still makes it hard to see the interplay between them
- When doing this, it’s important to make double-triple sure that your visualization actually matches the program state
  - Bugs that create inaccurate pictures come back to haunt you later

Saving global state

- The files we used in the sequential code are just a long list of floating point numbers stored in binary
- To make that again, we'll need to concatenate the numbers from all ranks, in rank order
- We can nominate rank 0 to be our "I/O-master", who can collect all the parts and put the file together

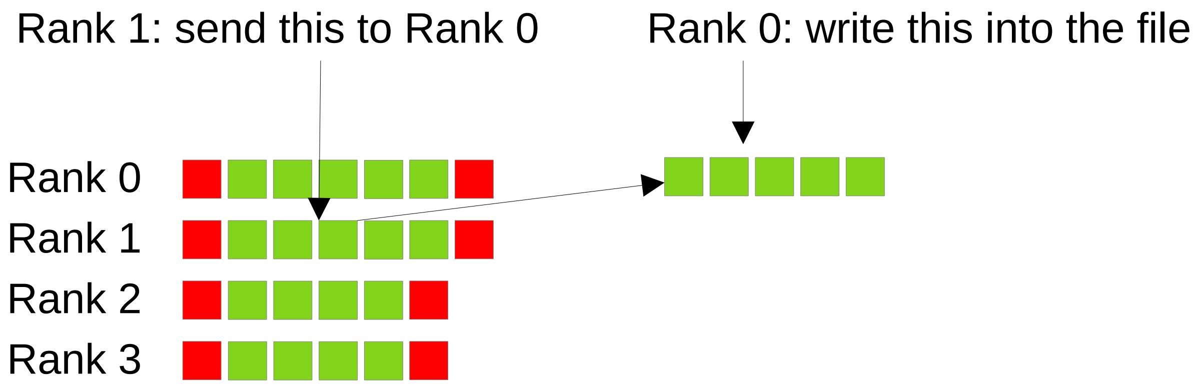
Rank 0 needs an extra buffer

- Because of the way we partitioned, rank 0 will always have (one of) the biggest sudomains, so we can use its size
- Step 1:



Rank 0 needs an extra buffer

- Rank 0 can now loop over the remaining ranks, and receive their sub-domains in the buffer
- Step 2:



- Steps 3 and 4 are just like step 2

This is a bottleneck

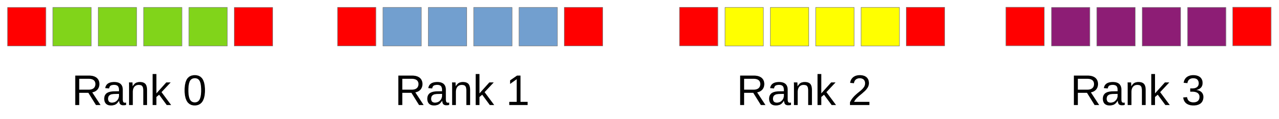
- We’re serializing the execution, rank 0 has to wait for all the ranks in turn, and do something sequential for them
- Another way would have been to let the ranks take turns to open the file and append to it
  - Still sequential, but with less communication
- We could make each rank save its own file, and concatenate them after the program has finished
  - Parallel, but creates more logistics afterwards
- Yet another would be to have everyone write at the same time
  - But we’re only doing 6-function MPI today
- Saving doesn’t happen on every iteration anyway

The example code archive

- This is the state of the code in the subdirectory “03_init_and_cleanup”
- It divides the problem and makes allocations
- It initializes all the arrays
- It saves the global initial state in a file
- It releases all the arrays again

Adding the solver

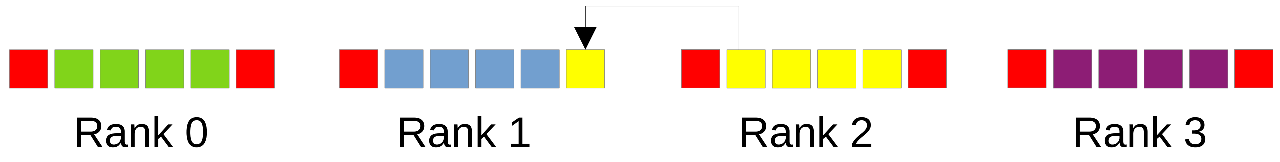
- If we draw the arrays of the ranks side-by-side, we can see an issue with the numerical method:
  - The boundaries are at ranks 0 and 3, but the calculation needs two neighbor values everywhere



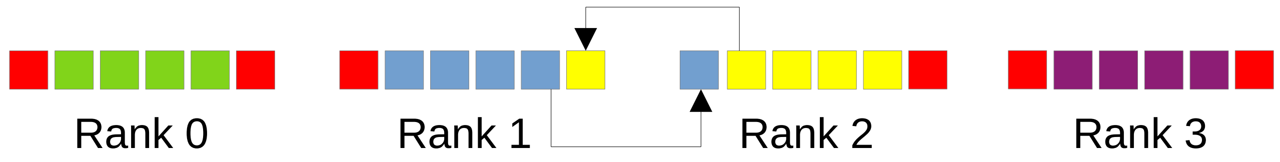
NTNU – Trondheim
Norwegian University of Science and Technology

The value of good neighbors

- The last value by rank 1 needs the first value from rank 2 as a neighbor
  - We can send it a copy



- The first value by rank 2 needs the last value from rank 1 also
  - We can send copies it in the other direction as well



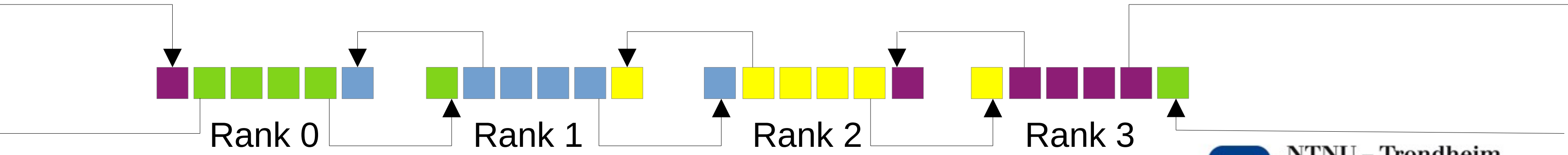
NTNU – Trondheim
Norwegian University of Science and Technology

Border exchange

- This operation is common, and it is called a ***border exchange***
- The artificial extra-points are often called ***ghost points***, together they are often referred to as a subdomain's ***halo***
- Since we're using a periodic boundary, border exchange takes care of that too, as long as we connect the first and last ranks:

```
left_neighbor  = ( rank + size - 1 ) % size;
right_neighbor = ( rank + size + 1 ) % size;
```

- Adding the extra 'size' here is just because moduli of negative numbers aren't a thing in C



NTNU – Trondheim
Norwegian University of Science and Technology

There is actually one more small problem lurking here, but we will deal with it when we talk about modes

Adding the solver

- When we’ve taken care that all the surroundings of the solver are as it expects, it can simply be used the way it was
- Only difference is that its loop has to go from 0 to the rank’s subdomain size, instead of to N
- That’s the code, let’s see if it runs any faster…